Fundamentos de Programación con la STL

Antonio Garrido Carrillo

eug

© ANTONIO GARRIDO CARRILLO
© UNIVERSIDAD DE GRANADA
© Fotografías y cubierta: ANTONIO GARRIDO CARRILLO
FUNDAMENTOS DE PROGRAMACIÓN
ISBN: 978-84-338-5918-1.
Edita: Editorial Universidad de Granada.
Campus Universitario de Cartuja. Granada.
Edición digital. 2016.

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 Lenguajes de programación 1

1.1 Esquema funcional de un computador 1

1.1.1 Funcionamiento de un computador 3

1.2 Codificación de la información 5

1.2.1 Sistemas de numeración 5

1.2.2 Sistema binario en los computadores 7

1.2.3 Representación de datos simples 8

1.3 Lenguajes de programación 12

1.3.1 El lenguaje C++ 14

1.3.2 Desarrollo de software 15

1.3.3 El proceso de traducción 16

1.4 Problemas 19

2 Introducción a C++ 21

2.1 Un programa básico 21

2.1.1 Escritura del fichero fuente 22

2.2 Datos y expresiones 23

2.2.1 Variables 24

2.2.2 Literales 25

2.2.3 Operaciones 26

2.2.4 Operador de asignación 28

EDITORIAL UNIVERSIDAD DE GRANADA

2.3 Entrada/Salida básica 30
2.3.1 Operadores de E/S estándar 31
2.3.2 Ejemplo 32
2.3.3 Entrada de datos. Algunos detalles 34
2.4 Funciones de biblioteca 35
2.5 Inicialización de variables y constantes 37
2.6 Problemas 38

3 La estructura de selección 43
3.1 Introducción 43
3.2 Operadores relacionales. Selección simple 44
3.2.1 Operadores aritméticos y relacionales 46
3.3 Selección doble *if/else* 47
3.3.1 Anidamiento 47
3.3.2 Formato de escritura 49
3.3.3 El operador ternario "?:" 49
3.4 Expresiones booleanas. Operadores lógicos 50
3.4.1 Evaluación en corto 52
3.5 Selección múltiple *switch* 53
3.6 Booleanos en C. Algunos detalles 54
3.7 Representación gráfica 56
3.8 Problemas 56

4 La estructura de repetición 59
4.1 Introducción 59
4.2 Bucles *while* 59
4.3 Bucles *for* 63
4.3.1 Nuevos operadores: incremento y decremento 65
4.3.2 Nuevos operadores de asignación 67
4.3.3 El operador coma 67
4.3.4 Ejemplos 67
4.4 Bucles *do/while* 71
4.5 Estructuras de control anidadas 72
4.6 Declaración y ámbito de una variable 75
4.6.1 Ocultación de variables 76
4.7 Programación estructurada y sentencia *goto* 76
4.7.1 Sentencias continue y break 77
4.8 Problemas 77

EDITORIAL UNIVERSIDAD DE GRANADA eug

5 Funciones . . . 81
5.1 Introducción 81
5.2 Funciones 82
5.2.1 Un programa con funciones. Declaración y definición . . . 84
5.2.2 Funciones *void* . . . 91
5.2.3 Paso por valor y por referencia . . . 93
5.2.4 Función *main*. Finalización del programa . . . 97
5.3 Diseño de funciones 98
5.3.1 Variables globales . . . 100
5.3.2 Funciones y programación estructurada . . . 101
5.3.3 Especificación de funciones . . . 104
5.3.4 Pruebas . . . 106
5.3.5 Errores . . . 106
5.4 Problemas 107

6 Vectores de la STL . . . 109
6.1 Introducción 109
6.2 Vectores 110
6.2.1 Declaración . . . 110
6.2.2 Acceso al contenido . . . 112
6.2.3 Funciones y tipo vector . . . 117
6.2.4 Búsqueda con vectores de la STL . . . 118
6.2.5 Algoritmos clásicos de ordenación . . . 121
6.3 Vectores dinámicos 132
6.3.1 Eficiencia y vectores dinámicos . . . 136
6.4 Matrices 137
6.4.1 Declaración . . . 138
6.4.2 Matrices rectangulares . . . 139
6.4.3 Matrices no rectangulares . . . 141
6.5 Problemas 143

7 Cadenas de la STL . . . 145
7.1 Introducción 145
7.2 Tipo string 146
7.2.1 Declaración . . . 146
7.2.2 Literal de tipo string . . . 146
7.2.3 El tipo *string* como contenedor de *char* . . . 147
7.2.4 E/S de objetos string . . . 147
7.2.5 Asignación de cadenas . . . 150
7.2.6 Operaciones relacionales . . . 151
7.2.7 Otras operaciones frecuentes . . . 152
7.2.8 Funciones y tipo string . . . 156

EDITORIAL UNIVERSIDAD DE GRANADA

7.3 El tipo char: codificaciones 156
7.3.1 Código ASCII: Extensiones 157
7.3.2 Unicode 160
7.3.3 Editor de texto y codificación usada 162

7.4 Problemas 162

8 Estructuras y pares 163

8.1 Introducción 163
8.1.1 Tipos de datos definidos por el usuario 164

8.2 Estructuras 164
8.2.1 Declaración de variables 165
8.2.2 El operador punto 165
8.2.3 Copia y asignación de estructuras 167
8.2.4 Ordenar estructuras 170

8.3 Pares con la STL 175
8.3.1 El tipo *Pair* 175
8.3.2 Operaciones con *pair* 177
8.3.3 Diccionarios 178

8.4 Problemas 183

9 Recursividad 185

9.1 Introducción 185
9.1.1 Recursión directa e indirecta 186

9.2 Ejemplos de funciones recursivas 186
9.2.1 Factorial de un número entero 186
9.2.2 Suma de los elementos de un vector 188
9.2.3 Listado de los elementos de un vector 189

9.3 Gestión de llamadas: la pila 189
9.3.1 La pila del sistema 190
9.3.2 Pila y recursividad 192
9.3.3 Gestión de la memoria 192

9.4 Diseño de funciones recursivas 193
9.4.1 Implementación de funciones recursivas 195
9.4.2 Descomposición en números primos 199

9.5 Recursivo vs iterativo 201
9.5.1 Recursión de cola 202

9.6 Ejemplos de funciones recursivas 203
9.6.1 Sucesión de Fibonacci 203
9.6.2 Torres de Hanoi 203
9.6.3 Ordenación por selección 205
9.6.4 Ordenación por mezcla 206

9.7 Problemas 208

EDITORIAL UNIVERSIDAD DE GRANADA

10 Introducción a flujos de E/S 211
10.1 Introducción 211
10.2 Flujos de E/S 212
10.2.1 Flujos y buffers 212
10.2.2 Flujos globales predefinidos 214
10.3 Operaciones básicas con flujos 216
10.3.1 Tamaño finito de los flujos 216
10.3.2 Estado de los flujos 218
10.3.3 Otras funciones de entrada útiles 223
10.4 Flujos asociados a ficheros 225
10.4.1 Apertura y cierre de archivos 226
10.4.2 Paso de flujos a funciones 230
10.5 Flujos y C++98 232
10.5.1 Apertura con *cadena-C* 233
10.5.2 Reset del estado con *open* 233
10.6 Problemas 233
11 Compilación separada 235
11.1 Introducción 235
11.2 Compilación separada 236
11.2.1 Compilación y enlazado 237
11.2.2 El preprocesador. Ficheros h y cpp 238
11.3 El preprocesador: Múltiples inclusiones de archivos cabecera 240
11.3.1 La directiva *define* 240
11.3.2 Compilación condicional 242
11.3.3 Un ejemplo más complejo 242
11.3.4 Dependencias entre archivos 246
11.4 Bibliotecas 246
11.5 Espacios de nombres 248
11.5.1 Creación de espacios de nombres. Nombres calificados 249
11.5.2 Eliminando la calificación: *using* 251
11.5.3 Espacio de nombres y distintos archivos 251
11.5.4 Evitando *using* en archivos cabecera 252
11.6 Objetos y múltiples ficheros 252
11.6.1 Variables locales a un fichero 253
11.6.2 Constantes globales 254
11.6.3 Funciones locales a un fichero 255
11.7 Problemas 255
EDITORIAL UNIVERSIDAD DE GRANADA

12 C++11/14 257
12.1 Introducción 257
12.2 Inicialización de tipos simples 258
12.2.1 Llaves para tipos simples 259
12.2.2 Deducción del tipo 260
12.3 El tipo *vector* 261
12.3.1 Terminación con >> 261
12.3.2 Bucle para recorrer un contenedor 261
12.3.3 Listas de inicialización 263
12.3.4 Optimización de movimiento de objetos 265
12.4 El tipo *string* 266
12.4.1 El tipo *string* como contenedor 266
12.4.2 Conversiones a/desde tipos numéricos 268
12.5 Estructuras, pares y tuplas 270
12.5.1 Inicialización con llaves 270
12.5.2 Tuplas 271
12.5.3 Pares y tuplas 273
A Solución a los ejercicios 275
A.1 Lenguajes de Programación 275
A.2 Introducción a C++ 275
A.3 La estructura de selección 278
A.4 La estructura de repetición 283
A.5 Funciones 290
A.6 Vectores de la STL 296
A.7 Cadenas de la STL 307
A.8 Estructuras y pares 311
A.9 Recursividad 316
A.10 Introducción a flujos de E/S 319
B Generación de números aleatorios 323
B.1 Introducción 323
B.2 El problema 323
B.2.1 Números pseudoaleatorios 324
B.3 Transformación del intervalo 325
B.3.1 Operación módulo 326
B.3.2 Normalizar a U(0,1) 326
EDITORIAL UNIVERSIDAD DE GRANADA eug

C Ingeniería del software 329
C.1 Introducción 329
C.2 Actividades en el proceso del software 330
C.3 La necesidad de un enfoque de calidad 330
C.4 Paradigmas en el desarrollo del software 331
C.5 Modelo del sistema 332
C.6 Paradigmas de la programación 333
C.6.1 Programación procedimental 333
C.6.2 Programación modular 334
C.6.3 Abstracción de datos 334
C.6.4 Programación orientada a objetos 335
C.6.5 Programación genérica 336
C.6.6 Metaprogramación 336
C.6.7 Programación funcional 337
C.7 Desarrollo de programas 337
C.7.1 Análisis 338
C.7.2 Diseño 338
C.7.3 Implementación 341
C.7.4 Pruebas 341
D Tablas 343
D.1 Tabla ASCII 343
D.2 Operadores C++ 344
D.3 Palabras reservadas de C89, C99, C11, C++ y C++11 346
Bibliografía 349
Índice alfabético 353
EDITORIAL UNIVERSIDAD DE GRANADA
eug

eug
EDITORIAL UNIVERSIDAD DE GRANADA

Prólogo

El objetivo de este libro es presentar los conceptos fundamentales de programación en el contexto del lenguaje de programación C++. A diferencia de otros libros, no se pretende mostrar todos los detalles de este lenguaje, sino enseñar los conceptos más básicos de programación.

Aprender a programar

Comenzar en el mundo de la programación no es una tarea trivial. A pesar de que un programador experimentado probablemente mirará los contenidos del curso convencido de su simplicidad, un estudiante sin ningún conocimiento previo podría encontrar complejos los detalles más insignificantes. Generalmente, la experiencia oculta la dificultad de los detalles de la implementación para enfocar el interés en el diseño de la solución.

En muchas ocasiones, la tarea de plantear un curso para aprender a programar centra su interés en las metas que se quieren conseguir, antes que en los primeros pasos que se deben dar. De hecho, este planteamiento es el que lleva, en muchos casos, a crear cursos iniciales novedosos en los que se pretende que el alumno esté realizando diseños orientados a objetos al mes de comenzar su formación o que realice complejas interfaces de usuario en base a un lenguaje de muy alto nivel. Si realiza un programa en tres líneas que realiza una operación sorprendente, no significa que sepa programar.

Si desea formar a un buen programador, es absurdo intentar conseguir que éste realice un gran diseño a los pocos meses de comenzar su formación. Muchas decisiones de diseño son consecuencia de un largo período de formación y experimentación. Si revisa los programas que durante años ha ido generando un buen programador, seguramente descubrirá una diferencia notable entre los primeros y los últimos. Es posible que los últimos le parezcan obras de arte, pero no olvide que su formación comenzó a un nivel más básico.

Teniendo en cuenta estas consideraciones, el curso que se presenta no es más que un curso básico de programación estructurada. Sin embargo, se ha diseñado pensando en los conocimientos que el estudiante debería tener para abordar otros cursos y materias más avanzadas relacionadas con la informática y la programación.

El lenguaje de programación

El lenguaje seleccionado para desarrollar el curso es C++. La discusión sobre qué lenguaje es ideal para un primer curso ha estado presente desde el comienzo de los estudios de informática. De hecho, ya en el año 1970 se presenta el lenguaje Pascal, especialmente diseñado para aprender a programar.

Los criterios para seleccionar un lenguaje u otro pueden ser muy variados. Por ejemplo, se proponen los lenguajes más usados, los que permiten mayor productividad, o los que está usando la empresa de software que esté de moda en ese momento. Sin embargo, lo más importante debería ser tener en cuenta que el estudiante no tiene ningún conocimiento y que es un primer curso de programación, ya sea para tener algunos conocimientos o para servir como base para otros cursos. Con estas condiciones, la elección es más sencilla.

Un lenguaje bastante usado en los primeros cursos es el lenguaje C. Es un lenguaje de bajo nivel en relación a otros, basado en la programación estructurada. Su simplicidad y su uso generalizado dentro y fuera de ámbito docente lo hace un buen candidato. Sin embargo, en muchos casos se desestima porque resulta demasiado alejado de la sintaxis y los conceptos que se usan en la mayoría de lenguajes de alto nivel actuales.

Por otro lado, lenguajes como Java son una alternativa para proponer una forma docente con la que priorizar conceptos de programación dirigida a objetos. El problema es que un alumno sin ningún conocimiento está lejos de entender esos conceptos, para los que realmente necesita conocer algunas estructuras más básicas que aparecen en la programación estructurada. Estos cursos pueden resultar algo divulgativos en algunos contenidos y más eficaces cuando se dedica un tiempo a olvidar esos diseños dirigidos a objetos y a centrarse en conceptos más básicos como las estructuras de control o la abstracción funcional.

Una solución que podría interpretarse como intermedia es el uso de un lenguaje multiparadigma como C++. Éste plantea distintos paradigmas de programación, por lo que el curso podría desviarse tanto a un modelo como a otro. Por ejemplo, podría plantearse el estudio de los aspectos más cercanos a C, resultando un curso con cierto paralelismo a un curso C primero y Java después.

La versatilidad de este lenguaje es un aspecto que nos hace seleccionarlo para este primer curso, lógicamente, teniendo cuidado de diseñar un curso que seleccione los contenidos necesarios y suficientes para un curso de fundamentos de programación. De hecho, en muchos casos se rechaza como primer lenguaje dada su complejidad, justificándolo incluso diciendo que para usar este lenguaje es necesario conocer el lenguaje C. En este libro, se propone un contenido guiado por aspectos docentes, evitando profundizar en detalles técnicos del lenguaje, seleccionando los conceptos que componen un curso completo, autocontenido y que preparan al estudiante tanto para temas de más bajo nivel —con un lenguaje como C— como para temas de más alto nivel —como en encapsulamiento y la abstracción—. Precisamente, lo que debería ser un curso de fundamentos de programación.

Este enfoque permite poner este complejo lenguaje al alcance del lector, aunque no tenga conocimientos previos, y lo prepara para profundizar y asimilar fácilmente otros conceptos más complejos de éste u otro lenguaje. Por tanto, resulta una referencia especialmente útil para planes de estudios donde quieran actualizar los contenidos desde lenguajes tradicionales —como *Pascal* o C— a un lenguaje con las posibilidades de C++.

Introducción a la programación para todos

La razón principal que ha motivado la creación de este libro ha sido la necesidad de ofrecer un documento práctico que facilite tanto la docencia del profesor como el aprendizaje del alumno en las carreras de ingeniería.

La evolución de los planes de estudios en los últimos 30 años ha desembocado en un modelo de enseñanza en el que el estudiante tiene una gran responsabilidad: por un lado, se han ampliado los

contenidos de la carrera, mientras por otro se han disminuido el tiempo de enseñanza en el aula. Una parte del esfuerzo debería realizarse de forma autodidacta. Los estudiantes deben asimilar una gran cantidad de contenidos en poco tiempo, por lo que resulta muy difícil disponer de un buen material de estudio si se limita únicamente a los apuntes de clase. Por ello, es necesario disponer de algún libro de apoyo.

Este libro intenta compensar esta dificultad para un estudiante de primer curso. Teniendo en cuenta los planes de estudios en el contexto europeo, así como el deseo de muchos lectores para el autoaprendizaje, este libro se convierte en una referencia útil para aprender a programar. No sólo puede usarse en las titulaciones de ingeniería informática, sino en otras ingenierías, en titulaciones de ciencias, en estudios medios, e incluso de forma autodidacta.

Organización del libro

La exposición de los temas se ha realizado pensando en el objetivo docente del libro. Para ello, se desarrollan los contenidos guiados por los conceptos siguiendo una dificultad incremental, sin suponer ningún conocimiento previo del lector. Se evita la exposición de contenidos guiados por el lenguaje, especialmente evitando detalles que no son importantes para la formación general de un programador.

Los temas se abordan con una discusión razonada de los contenidos, presentando los detalles que el estudiante debería conocer, pero también justificando por qué y cómo se utilizan en la práctica. Se intenta evitar la exposición "por recetas", es decir, que el estudiante memorice soluciones en lugar de entender cómo crearlas. El estudiante debería poder crear buenas soluciones, en lugar de conformarse con "programas que funcionan". Las soluciones deberían reflejar una capacidad de diseño que garantice los mejores resultados en cursos avanzados, donde un proyecto de cierto tamaño sólo puede resolverse con los mejores fundamentos de programación.

En este desarrollo, se exponen los temas incluyendo ejemplos ilustrativos. Además, se proponen ejercicios a lo largo del tema, independientes de los problemas finales del tema. El lector debería asimilar los contenidos y ser capaz, en gran medida, de resolver estos ejercicios propuestos como una forma de confirmar que se ha entendido el tema. Además, el libro incluye las soluciones al final para aquellos estudiantes que tengan más dificultades o quieran confirmar o comparar sus soluciones.

Contenidos

Los capítulos se han organizado para que el lector vaya asimilando el lenguaje poco a poco. En este sentido, los temas no pretenden presentar un aspecto del lenguaje para que se conozca completamente antes de pasar al siguiente. Cada tema introduce algunos aspectos adicionales para que el lector empiece a utilizarlos, pero pueden ser completados y asimilados completamente a lo largo de ese y otros capítulos posteriores.

Un repaso rápido por el índice le puede hacer entrever que es un libro organizado de una forma muy diferente a otras referencias. Aunque se trabaja con C++, no aparecen tipos tan básicos como los *vectores-C* o los punteros. En su lugar, se incluyen tipos de la STL que resuelven los mismos problemas eliminando detalles de muy bajo nivel que no son necesarios en un curso de fundamentos, y ofreciendo una interfaz bien desarrollada que facilita al alumno avanzar más fácilmente en cursos posteriores.

En el **capítulo 1** se introduce al lector en el mundo de la programación. Para ello, suponemos que no tiene experiencia anterior, y por tanto incluimos los conceptos más básicos para que entienda lo que es un ordenador, un lenguaje de programación o un programa.

En el **capítulo 2** se abordan las ideas más simples para introducir al lector en el lenguaje C++. El objetivo es que pueda realizar pequeños programas cuanto antes, para conocer de forma práctica

qué quieren decir esos conceptos expuestos en la lección. Se presentan conceptos muy sencillos —como lo que es una variable o un tipo de dato— pero que pueden resultar complicados al ser tan novedosos. Se muestran ejemplos con los primeros algoritmos en este lenguaje, programas muy simples que incluyen sentencias de declaración de datos, expresiones y E/S.

En el **capítulo 3** se introduce la idea de flujo de control y bifurcación en un programa. Ese control de flujo motiva la presentación del tipo booleano junto con los operadores adicionales, así como las sentencias `if-else` y `switch`.

En el **capítulo 4** completamos las estructuras básicas de control con los bucles: `for`, `while` y `do-while`. Los programas ya tienen bastante complejidad como para añadir algunas ideas sobre declaración y ámbito de las variables.

En el **capítulo 5** se introducen las funciones como la herramienta más básica de modularización. En este punto el alumno deber realizar un esfuerzo por empezar a cultivar su habilidad de abstracción, aunque a este nivel se limite a crear abstracciones funcionales. En este tema sólo se presentan los detalles más básicos sobre funciones, de forma que el lector sea capaz de manejar problemas más complejos y de crear soluciones estructuradas. El aumento en la complejidad de los programas permite empezar a conocer el problema de la modularización y la importancia de obtener un buen diseño.

En el **capítulo 6** se presentan los tipos de datos compuestos, concretamente los vectores y matrices. En los temas anteriores el lector debe haber asimilado los conocimientos básicos para desarrollar algoritmos de forma estructurada. En este tema, se avanza en la complejidad de los datos introduciendo nuevos tipos y relacionándolos con los conocimientos adquiridos. El contenido se basa en el tipo `vector` de la STL, por lo que será necesario que el alumno empiece a adaptarse a una nueva interfaz y comience a trabajar con objetos y sus operaciones. Estas nuevas estructuras de datos permiten empezar a introducir algunos algoritmos clásicos, como los de búsqueda y ordenación, que son una base que cualquier programador debería conocer pero que a su vez constituyen ejemplos ideales para empezar a practicar.

En el **capítulo 7** se presentan las cadenas, el tipo `string` de la STL. En gran medida, es una nueva revisión de ideas que se han expuesto en el tema anterior, aunque en este caso se especializa para una secuencia de caracteres en lugar de un vector de un tipo general. Se aprovecha el tema para hablar en mayor profundidad sobre el tipo `char` y la codificación.

En el **capítulo 8** se presentan las estructuras, como una nueva forma de definir tipos de datos compuestos, en este caso heterogéneos. Se muestran los aspectos más básicos y se relacionan con los temas anteriores. Además, se introducen los pares —tipo `pair`— de la STL, un tipo especialmente importante por su uso en otras estructuras de datos, no sólo en este lenguaje.

En el **capítulo 9** se presenta la recursividad, con lo que se completan los conocimientos relacionados con funciones en este curso básico. Aunque la recursividad no parezca más que una simple posibilidad más a la hora de llamar a una función, muchos programadores novatos encuentran dificultades para crear funciones recursivas. Ese nivel extra de abstracción sobre lo que es una función parece un problema para estos programadores. Por eso, el tema se ha retrasado para que el alumno pudiera practicar con funciones antes de plantear esta posibilidad. En este tema, se estudia en detalle el problema de su diseño, acompañándolo con múltiples ejemplos y consejos. Además, se presentan los detalles de la pila y cómo el compilador resuelve las llamadas a función. Esta visión permite entender mejor cómo funciona un programa, facilitando tareas tan importantes como la traza de un programa.

En el **capítulo 10** se estudian los flujos de E/S. Este tema se limita a estudiar los conceptos fundamentales, explicando mejor cómo funcionan los objetos `cin` y `cout`, intensivamente usados durante el curso. En este punto, se revelan algunos aspectos que habían sido esquivados en los temas anteriores, haciendo que el estudiante pueda controlar mejor estos objetos y ampliando sus posibilidades mediante la introducción de flujos asociados a ficheros.

En el **capítulo 11** se estudia la compilación separada. Aunque se han presentado sólo fundamentos del lenguaje, evitando muchos tipos y detalles, el contenido de los temas anteriores ya permite al estudiante abordar problemas de cierta complejidad. Tal vez la solución no sea la más óptima, puesto que se conoce sólo una parte del lenguaje; a pesar de eso, a este nivel, el estudiante está preparado para crear verdaderos proyectos que basados en programación estructurada configuren una buena solución a un problema. La compilación separada permite que puedan crearse estas soluciones con un buen diseño y una forma eficaz de distribuir el código.

Finalmente, en el **capítulo 12** se completa el curso con contenidos referidos a C++11/14. Desde el punto de vista de los conceptos no aporta mucho más a los temas anteriores. Pero desde el punto de vista práctico no podemos olvidar que los futuros programadores de C++ trabajarán en el estándar actual.

Se ha dejado en un tema extra porque en un curso de fundamentos el lenguaje C++98 es suficiente, lo que hace innecesario complicarlo más. De hecho, mi recomendación es comenzar de esta forma. Una vez que están asentadas las bases de la programación, resulta mucho más sencillo añadir las novedades de C++11. Las secciones del tema están ordenadas conforme al curso, por lo que un profesor interesado en el nuevo estándar podría añadirlas de forma relativamente sencilla a las distintas partes del curso.

Además, se incluyen varios apéndices que completan los temas anteriores:

- En el **apéndice A** aparecen las soluciones a los ejercicios propuestos a lo largo del libro. Para que el lector asimile más fácilmente los contenidos, se proponen ejercicios a la largo de los temas, de forma que el estudio de éstos se realice de una forma más interactiva. Cuando se propone un ejercicio, el lector debería intentar resolverlo proponiendo su solución, tras lo que debería consultar la solución propuesta en este apéndice.
- En el **apéndice B** se muestran las herramientas más básicas que podemos usar para la generación de números aleatorios. Son las mismas que puede encontrar en C. En un curso de introducción a la programación pueden resultar muy útiles, ya sea para crear algunos programas que lo necesitan —por ejemplo, pequeños juegos— como para generar secuencias de datos para pruebas.
- En el **apéndice C** se realiza una discusión sobre la ingeniería del software. Con los conocimientos adquiridos, el lector está preparado para resolver problemas con cierta envergadura. En este sentido, resulta conveniente empezar a introducir algunos conocimientos sobre ingeniería del software, así como insistir sobre algunos aspectos relativos a la calidad del software desarrollado. La ingeniería del software es un tema que daría para escribir no uno, sino varios libros. En este apéndice, sólo se pretenden exponer algunos conceptos básicos sobre lo que un programador, aunque sea novato, debería conocer.
- En el **apéndice D** se presentan algunas tablas de interés para un programador de C++.

Finalmente, se presenta la bibliografía, donde aparece la lista de referencias que se han usado para la creación de este libro o que resultan de especial interés para que el lector amplíe conocimientos.

Sobre la ortografía

El libro se ha escrito cuidando la redacción y la ortografía; no podría ser de otra forma. Se ha intentado cuidar especialmente la expresión y la claridad en las explicaciones. Como parte de este cuidado, es necesario incluir en este prólogo una disculpa por no respetar todas y cada una de las normas que actualmente establece la Real Academia Española (RAE) sobre el español.

En primer lugar, es difícil concretar un lenguaje claro y duradero en el contexto de la informática, especialmente porque cambia e incorpora nuevos conceptos continuamente. Unido al retraso que requiere el análisis y la decisión por parte de la RAE para incluirlos en el diccionario de la lengua española, siempre podemos encontrar alguna palabra que se adapta desde el inglés y no está aún

aceptada. Por ejemplo, usaremos la palabra *unario* como traducción de *unary*. Otros palabras que usamos de forma natural cuando comentamos código en C++ pueden ser *booleano, token, buffer, precondición, endianness, log, argumento, preincremento, prompt,* etc.

Por otro lado, es habitual que el programador trabaje con documentación técnica que está escrita en inglés. Esto hace que un uso de palabras similares en español sea una forma de facilitar la lectura rápida y cómoda. Por ejemplo, la palabra *indentación* en lugar de *sangrado*, *casting* en lugar de *moldeado* o *array* en lugar de *arreglo*.

Además, es probable que haya alguna otra palabra. Por ejemplo, la palabra *sumatoria* que encuentro en la *RAE* como *sumatorio*. ¿Por qué la hacen masculina si *suma* es femenina? Que yo recuerde, todos mis profesores y compañeros la han llamado y la llaman *sumatoria*.

Sin duda, mis cambios son un error, pero desde un punto de vista técnico prefiero no darle excesiva importancia. Es más importante superar la dificultad de los contenidos técnicos, dejando cuestiones de la lengua en un segundo plano.

A pesar de todo, he querido ajustarme en lo posible a lo que nos dice la RAE; por respeto al estupendo y complejo trabajo que realizan, pero sobre todo por respecto a nuestra lengua materna, el principal legado común que nuestros ancestros nos han dejado.

Una vez certificado mi respeto a la lengua y a la RAE como institución, tengo que disculparme por introducir alguna palabra u ortografía que no es actual. Esta disculpa se refiere, especialmente, al uso de la tilde diacrítica en el adverbio *sólo* —note la tilde— y los *pronombres demostrativos*. En mi humilde opinión, la distinción gráfica con respecto al adjetivo *solo* y los *determinantes demostrativos*, respectivamente, justifican claramente su uso. Si quiero priorizar la legibilidad, la claridad, el énfasis de lo que comunico, creo que su uso está justificado.

Por supuesto, seguramente habrá otros errores ortográficos o gramaticales; éstos son fruto de mi torpeza e ignorancia, por lo que pido humildemente disculpas.

El entorno de desarrollo

El entorno de desarrollo es otra de las discusiones que se plantean repetidamente en los primeros cursos de programación. La decisión incluye no sólo el lenguaje, sino el sistema operativo, los entornos de desarrollo —*IDE*— e incluso otras herramientas.

En nuestro caso, la elección del lenguaje C++ nos abre un amplio abanico de posibilidades. Lo más sencillo es aconsejar software libre que permite a cualquier usuario comenzar sin ningún coste. Una opción ideal, por tanto, es el compilador de la *GNU* que está disponible para múltiples plataformas.

Por otro lado, nuestro interés es centrar el esfuerzo en los conceptos de programación. No es conveniente distraer al estudiante con detalles técnicos sobre gestión de proyectos antes de que sea necesario. Por eso, lo más recomendado es usar algún tipo de entorno de desarrollo que simplifique todas estas tareas.

En nuestro caso, hemos usado con éxito el entorno *codeblocks*, gratuito y disponible para distintas plataformas, incluyendo las habituales *Windows* y *GNU/Linux* tan habituales entre los estudiantes. Por supuesto, cualquier otro entorno similar podría ser válido, aunque esta decisión se sale de los objetivos de este libro. Cada profesor deberá escoger el entorno ideal basado en un compilador que siga el estándar C++, por lo que no tendrá ningún problema para llevar a la práctica las lecciones de este libro.

Agradecimientos

En primer lugar, quiero agradecer el trabajo desinteresado de miles de personas que han contribuido al desarrollo del software libre, sobre el que se ha desarrollado este material, y que

constituye una solución ideal para que el lector pueda disponer del software necesario para llevar a cabo este curso.

Por otro lado, no puedo olvidar a las personas que me han animado a seguir trabajando para crear este documento. El trabajo y tiempo que implica escribir un libro no viene compensado fácilmente si no es por las personas que justifican ese esfuerzo. En este sentido, quiero agradecer a mis alumnos su paciencia, y recordar especialmente a aquellos, que con su esfuerzo e interés por aprender, han sabido entender el trabajo realizado para facilitar su aprendizaje.

Finalmente, quiero agradecer el apoyo de los compañeros que han usado este material como guía en sus propias clases, ya que con ello me han animado a completarlo y a terminar este libro.

A. Garrido.
Mayo de 2016.

1 Lenguajes de programación

Esquema funcional de un computador 1
Codificación de la información 5
Lenguajes de programación 12
Problemas 19

1.1 Esquema funcional de un computador

En este apartado vamos a conocer, de forma muy simplificada, qué es y cómo funciona una computadora. Una definición simple es la siguiente (véase la figura 1.1):

Definición 1.1 — Computadora. Un *computador* es una máquina diseñada para aceptar un conjunto de datos de entrada, procesarlos[a], y obtener como resultado un conjunto de datos de salida.

[a] Por procesar datos entendemos la realización de operaciones aritméticas y lógicas con ellos.

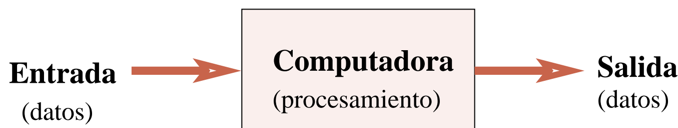


Figura 1.1
E/S de datos de una computadora.

Por otro lado, debemos tener en cuenta que con una computadora podemos realizar distintas tareas. Para ello, no sólo podemos introducir datos para procesar, sino también las instrucciones que indican cómo se procesan. Normalmente estas instrucciones están previamente almacenadas en la computadora, por lo que el usuario sólo necesita interactuar con ella por medio de la entrada y salida (E/S) de datos.

Definición 1.2 — Programa. Denominamos *Programa* al conjunto ordenado de instrucciones que indican a la computadora las operaciones que se deben llevar a cabo para realizar una determinada tarea.

Definición 1.3 — Hardware. Denominamos *Hardware* a la parte física del sistema, es decir, el conjunto de dispositivos, cables, transistores, etc. que lo conforman.

Definición 1.4 — Software. El término *Software* se usa para denominar a la parte lógica, al conjunto de programas, por tanto a la parte intangible del sistema. En el contexto de la ingeniería informática lo usaremos de forma más general, ya que le asociaremos no sólo los programas, sino toda la información asociada a su desarrollo y mantenimiento.

En la figura 1.1 hemos representado una computadora (*hardware*) que contiene los programas (*software*), en la que se introducen una serie de datos (*entrada*) para obtener unos resultados (*salida*).

Podemos realizar un esquema un poco más elaborado de una computadora, como el que se presenta en la figura 1.2.

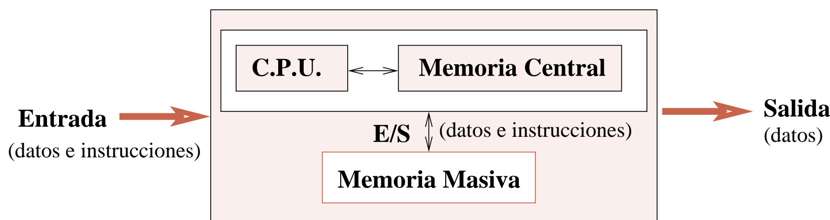


Figura 1.2
Esquema simplificado de un computador.

En este esquema hemos diferenciado, principalmente, dos componentes:

- *Unidad central (Memoria principal y C.P.U.)*. Corresponde al computador propiamente dicho. Por tanto, es la que se encarga de recibir los datos, de procesarlos y de devolver los resultados. Las dos partes fundamentales que la componen son:
  - La *memoria principal*. Donde se encuentran las instrucciones y datos que se están procesando. La memoria se puede considerar como una tabla compuesta por miles de casillas numeradas (*0,1,2,...*) en las que se almacena un número de un determinado rango (supondremos un número del rango 0-255). Cada una de las casillas es una *posición de memoria* [1]. Por tanto, nuestras instrucciones y datos están *codificados* como una secuencia de números.
  - La *C.P.U.*. Sigue las instrucciones y realiza el procesamiento que determina el software. Podemos dividirla en dos componentes:
    - La *unidad de control*. Se encarga de ejecutar las instrucciones de los programas. Para ello, controla el estado de los dispositivos, y es capaz de mandar señales de control para que actúen.
    - La *unidad aritmético-lógica*. Realiza las operaciones aritméticas y lógicas. Este dispositivo dota a la CPU de capacidad para realizar cálculos (operaciones básicas).

- *Memoria Masiva*. Corresponde a dispositivos con alta capacidad de almacenamiento y su misión es la de almacenar tanto datos como programas. Realmente son externos al computador y su relación con éste es, como habíamos indicado, de E/S de datos e instrucciones. Sin

[1] También se usa *palabra de memoria*.

embargo, dada su importancia y papel en los sistemas, se puede considerar como parte del computador.
Esta memoria es de gran tamaño, más barata y más lenta que la memoria principal. Observe que la memoria principal es la que usa directamente la CPU, por lo que conviene que sea rápida. De hecho, es mucho más rápida que la memoria masiva.

Si lee las especificaciones de los dispositivos que usa habitualmente, descubrirá que parecen bastante más complejos que la descripción anterior. Por ejemplo, puede encontrarse con varios núcleos, con memoria caché intermedia, *GPU*, discos externos de estado sólido, etc.

Tenga en cuenta que el esquema presentado nos permite disponer de un modelo funcional, dando sentido a las discusiones que se presentan en los siguientes temas, donde por ejemplo hablaremos de la ejecución de instrucciones por parte de la *CPU*, el almacenamiento en memoria de un dato o el salvado a dispositivos externos. La descripción de la arquitectura de un ordenador moderno se sale de los objetivos de este curso.

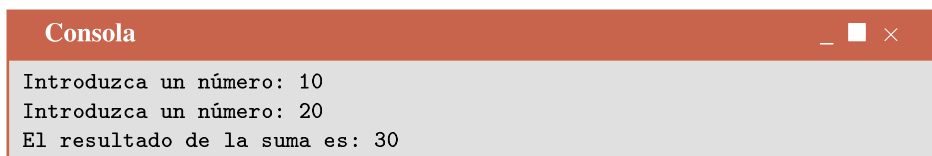
1.1.1 Funcionamiento de un computador

Para entender completamente el esquema indicado, es necesario describir el funcionamiento de la unidad central, es decir, cómo se relacionan los programas, los datos, la memoria y la CPU.

Consideremos un programa muy simple, llamado sumar, que se limita a pedir dos números al usuario y escribir el resultado de la suma. Para ello, se utiliza el teclado como dispositivo de entrada y la pantalla como dispositivo de salida. El programa realiza los siguientes 7 pasos:

1. Operación de salida de un mensaje *"Introduzca un número:"*.
2. El usuario introduce un número.
3. Operación de salida de un mensaje *"Introduzca un número:"*.
4. El usuario introduce un número.
5. Operación de suma entre el primer y segundo número introducidos.
6. Operación de salida de un mensaje *"El resultado de la suma es:"*.
7. Operación de salida del resultado de la suma.

El aspecto final que podrían tener en pantalla estos mensajes es:



La pregunta es: ¿Cómo ha llevado a cabo todo esto el ordenador?

En primer lugar, el usuario ha indicado al sistema que debe realizar la tarea que indica el programa *sumar*. A esta acción la denominamos *ejecutar* o *lanzar* el programa *sumar*. Para ejecutarlo, el sistema debe llevar el programa —que puede estar almacenado en la memoria masiva— hasta la memoria principal. Es decir, se *carga* el programa en memoria principal. Recordemos que el computador propiamente dicho lo componen la CPU y la memoria principal, por ello, cualquier programa que se ejecute deberá estar cargado en ésta.

El programa está compuesto por una secuencia de números que codifican todas las instrucciones a realizar. En el ejemplo anterior, los 7 pasos que hemos indicado. Ahora bien, este conjunto de instrucciones —que ejecutará la CPU— tiene la desventaja de que está limitado a unas acciones muy simples, ya que la complejidad de implantarlas en los circuitos hardware del procesador es muy alta[2]. Por lo tanto, para realizar una instrucción que puede parecer simple, el programa la tiene

[2]A pesar de que los procesadores actuales ya poseen, en un solo chip de pocos centímetros cuadrados, hasta cientos de millones de transistores.

dividida en pequeños pasos de operaciones que sabe llevar a cabo la computadora. Por ejemplo, piense que hay una persona que sólo entiende las instrucciones:

- Dar paso.
- Girar a la derecha.
- Girar a la izquierda.

y usted tiene que darle instrucciones para llegar hasta algún lugar. Aunque el trayecto resulte sencillo, probablemente se descompondrá un muchas instrucciones del tipo anterior.

En nuestro ejemplo, podemos pensar en las instrucciones del paso quinto, donde se suman dos números. La configuración de la memoria podría ser algo como lo que se presenta en la figura 1.3 (Para simplificar el ejemplo, supongamos que tanto una instrucción como un dato numérico se *almacenan* en una posición de memoria).

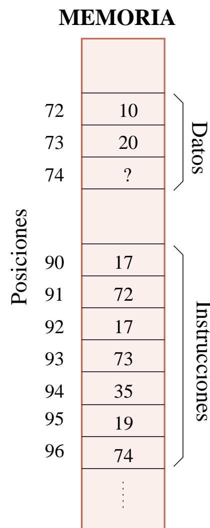


Figura 1.3
Datos e instrucciones de la suma.

Podemos observar que se reservan las posiciones *72,73,74* para almacenar el primer dato, el segundo y el resultado, respectivamente. Las instrucciones se encuentra en las posiciones *90-96*. Por supuesto, existen otras previas para realizar los primeros 4 pasos y otras para los últimos. En este gráfico, presentamos únicamente el conjunto de instrucciones que son necesarias para llevar a cabo la suma.

La situación actual es que hemos leído el dato "10" (está en la posición *72*) y el dato "20" (está en la posición *73*). Todavía no hemos guardado nada en la posición *74*, por lo que en principio desconocemos el número contenido. Además, estamos justo antes de realizar la suma, así que la siguiente instrucción a realizar está en la posición *90* (utilizaremos *CONTADOR*[3], para indicar la posición por la que vamos).

Una descripción simplificada del comportamiento del computador para ejecutar una instrucción del programa es:

1. La CPU obtiene el número de la posición *CONTADOR* —el 17— que corresponde a una instrucción.
2. Decodifica el significado del código 17, que indica que debe llevar el dato de la posición de memoria a la CPU. Carga la dirección de memoria a la que se refiere (dirección 72).

[3]De *Contador del Programa*.

3. Ejecuta la instrucción. Por tanto, ya tiene el dato 10 —que acaba de obtener de la dirección 72— en la CPU.

Ahora *CONTADOR* vale 92, que corresponde a la siguiente instrucción. Se vuelve a repetir el proceso:

1. La CPU obtiene el número de la posición *CONTADOR* —otro 17— que corresponde a una instrucción.
2. Decodifica el significado del código 17, que indica que debe llevar el dato de la posición de memoria a la CPU. Carga la dirección de memoria a la que se refiere (dirección 73).
3. Ejecuta la instrucción. Por tanto, ya tiene el dato 20 —que acaba de obtener de la dirección 73— en la CPU.

Ahora CONTADOR vale 94, que corresponde a la siguiente instrucción. Se repite:

1. La CPU obtiene el número de la posición *CONTADOR* —ahora 35— que corresponde a una instrucción.
2. Decodifica el significado del código 35, que indica que debe realizar la suma de los datos en la CPU.
3. Ejecuta la instrucción. Por tanto, ya tiene el dato 30 en la CPU.

Finalmente, repite el proceso para la última instrucción —el número 19— que indica que debe llevar el dato resultante desde la CPU a la memoria.

Note que esta discusión se ha realizado de manera informal. Por supuesto, podríamos haber descrito este funcionamiento usando un conjunto de instrucciones más cercano a un caso real, e incluso introduciendo más detalles sobre el sistema. Sin embargo, el objetivo es que el lector entienda el funcionamiento evitando detalles menos importantes. Si está interesado en más detalles, puede consultar múltiples libros, por ejemplo, Prieto[24].

En este punto, lo más relevante es que entienda hasta qué punto las instrucciones que sabe ejecutar una *CPU* son simples. Además, la descripción de cada uno de los pasos es muy compleja. Imagine que tiene que especificar las instrucciones de uno de los programas que habitualmente ejecuta en sus dispositivos. La tarea es tan complicada que sería prácticamente imposible describirla en términos de las instrucciones de la *CPU*.

1.2 Codificación de la información

Un ordenador puede usar y almacenar datos muy complejos, por lo que es necesario codificarlos y manejarlos de alguna forma. Teniendo en cuenta que el ordenador está compuesto por un conjunto de circuitos que funcionan con corrientes eléctricas; ¿Cómo podemos llegar desde éstas hasta los datos que habitualmente podemos usar en un programa?

1.2.1 Sistemas de numeración

El sistema de numeración más usado es el conocido sistema decimal, en el que consideramos un conjunto de 10 dígitos: del 0 al 9. Estamos habituados a escribir un número como una secuencia de dígitos decimales, separando la parte entera de la decimal con un punto[4]. En este sistema, se utiliza la base 10 ($B=10$) para representar cualquier número. Por ejemplo, el número 275.3 en base 10 representa la cantidad:

$$2 \cdot 10^2 + 7 \cdot 10^1 + 5 \cdot 10^0 + 3 \cdot 10^{-1}$$

[4]Realmente, la normativa internacional establece el uso de la coma para separar la parte decimal, aunque también acepta el uso anglosajón del punto como separador. En nuestro caso, nos acostumbraremos al punto, ya que es de obligado uso en el lenguaje.

También podemos seleccionar otra base para representar un número. Por ejemplo, podemos seleccionar base 8 —octal— de forma que los únicos dígitos que tendríamos serían *{0,1,2,3,4,5,6,7}*. En este caso, el número 275.3 —en base octal— representa la cantidad:

$$2 \cdot 8^2 + 7 \cdot 8^1 + 5 \cdot 8^0 + 3 \cdot 8^{-1}$$

Note que este cambio de base sólo significa un cambio de notación (usamos menos dígitos) nunca una pérdida de capacidad. Los números que habitualmente escribimos se pueden representar también usando otros sistemas con el mismo significado.

Las operaciones que solemos hacer en base decimal también se pueden hacer el cualquier otra base. Por ejemplo, cuando contamos desde el cero en adelante, lo que hacemos es ir incrementando en uno el dígito menos significativo. Cuando ese dígito alcanza su máximo valor, añadir uno significa hacerlo cero e incrementar en uno el siguiente dígito. Esa misma operación se haría igual en otra base.

Es interesante destacar que también podemos usar bases mayores de 10. Por ejemplo, podemos seleccionar el sistema de base 16 (*hexadecimal*), donde tenemos 16 dígitos. Como son más de 10, no tenemos representación gráfica para los últimos 6, así que usamos las primeras letras del alfabeto como dígitos (*A*-*F*). Por ejemplo, el 10 en base hexadecimal es el 16 en decimal, o si al 10 en hexadecimal le quitamos uno, nos da el *F* en base hexadecimal (15 en decimal).

La base 12 es otra posibilidad, aunque de poco interés para la informática, aparece en múltiples referencias argumentando que sería un mejor sistema para la sociedad. El sistema *duodecimal* o *docenal* usa 12 caracteres, del 0 al 9, más dos para representar el 10 y el 11. Por ejemplo, las letras *A* y *B*. Incluso hay una propuesta para crear dos dígitos (un 2 rotado para el 10 y un 3 rotado para el 11). Si está interesado, consulte internet donde encontrará más detalles.

De la misma forma, podemos seleccionar el sistema binario (base 2) donde únicamente tenemos dos dígitos: el 0 y el 1. Este sistema es fundamental para nosotros, pues la información acabará representada como secuencias de estos dos dígitos.

Un ejemplo de algunos números en las bases más relevantes comentadas se presenta en la tabla 1.1.

Tabla 1.1
Ejemplos notación decimal/binaria/octal/hexadecimal

Decimal	binario	Octal	hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
...	...	...	...
8	1000	10	8
9	1001	11	9
10	1010	12	A
...	...	...	...
14	1110	16	E
15	1111	17	F
16	10000	20	10

Ejercicio 1.1 Calcular el valor del número hexadecimal *F3A.C9* en base decimal.

Presentar los ejemplos de base 2, 8 y 16 no es casual. El sistema binario es la base sobre la que se apoya la tecnología desarrollada para fabricar nuestros computadores, y por tanto es fundamental para aprender a programar. Por otro lado, aunque la octal y hexadecimal parezcan menos relevantes, resultan muy convenientes para poder manejar de manera cómoda la base binaria. Es muy simple pasar de base 2 a las otras bases y a la inversa, ya que 8 y 16 son potencias de la primera.

1.2.2 Sistema binario en los computadores

Al nivel de los circuitos electrónicos, resulta fácil representar la información de un dígito binario. Por ejemplo, podemos usar dos estados[5] (hay conducción o no) para representar los dígitos binarios 0 y 1. Un dígito binario se denomina *bit*, y representa la cantidad mínima de información.

Si consideramos que nuestro ordenador usa el sistema binario, podemos detallar aún más el esquema de la memoria que antes comentábamos, indicando que realmente se compone de un conjunto de dígitos binarios (normalmente, del orden de miles de millones de bits). Estos bits se agrupan en pequeños grupos de igual tamaño, cada uno de ellos considerado una posición de memoria. Por ejemplo, podemos agruparlos en grupos de 8 bits, así, en cada posición de memoria se pueden guardar únicamente valores que van desde el *00000000* (cero en decimal) al *11111111* (255 en decimal).

Con esta codificación, el conjunto de instrucciones *17, 72, 17, 73, 35, 19, 74* que se presentaban en la figura 1.3, son realmente una secuencia de dígitos binarios. Si consideramos grupos de 8 bits en cada posición de memoria, esos valores se almacenan como:

```
00010001 01001000 00010001 01001001 00100011 00010011 01001010
```

Además, para la computadora las direcciones de memoria también se codifican como números binarios, las señales de control a los dispositivos para que realicen una operación —por ejemplo, una suma de dos números binarios— también son valores de ceros y unos. Como vemos, el interior de un ordenador es un "mundo de bits".

Medidas de memoria

El tamaño de la memoria puede llegar a ser muy grande; es necesario incluir múltiplos del bit para simplificar la forma de expresar estos valores. Por ejemplo, un *byte*[6] corresponde a 8 bits. Algunas de estas medidas son:

- *Kibibyte* (KiB)= 2^{10} Bytes = 1024 Bytes $\approx 10^3$ Bytes
- *Mebibyte* (MiB)= 2^{20} Bytes= 1048576 Bytes $\approx 10^6$ Bytes
- *Gibibyte* (GiB)= 2^{30} Bytes $\approx 10^9$ Bytes
- *Tebibyte* (TiB)= 2^{40} Bytes $\approx 10^{12}$ Bytes

El uso de estas medidas es muy habitual. Por ejemplo, actualmente la memoria principal ya alcanza tamaños del orden del Gibibyte, así como los sistemas de almacenamiento masivo del orden de varios Tebibytes.

Finalmente, es importante precisar que estas medidas se suelen encontrar en múltiples referencias como *Kilobyte*, *Megabyte*, *Gigabyte* y *Terabyte*, respectivamente. Estos fueron los términos que inicialmente —y durante décadas— se han usado en informática. Sin embargo, el hecho de que generaba confusión con el sistema internacional (donde kilo es 10^3, mega 10^6, etc.) se estableció

[5] Piense, por ejemplo, en una bombilla que representa un cero cuando está apagada o un uno cuando está encendida.
[6] Corresponde al número de bits necesarios para almacenar un carácter, que generalmente es de 8.

que el término *binary* debería aparecer en los múltiplos basados en potencias de dos. De ahí, surgió **ki**lo-**bi**nary-byte, **me**ga-*bi*nary-byte, etc[7].

1.2.3 Representación de datos simples

Como hemos visto, el sistema utiliza bits para representar la información, de hecho, los utiliza para todo. Resulta especialmente útil conocer la forma en que podemos representar datos que, siendo simples, nos encontraremos en nuestros problemas. Por ejemplo, valores enteros, reales, y mensajes tales como *"Introduzca un número"*.

Representación de enteros

No es posible representar cualquier número entero, ya que el número de dígitos —y por tanto de bits— tendría que ser ilimitado, mientras que la capacidad de memoria para almacenarlos es finita.

Sin embargo, en la práctica el rango de posibles valores enteros que necesitamos manejar para nuestros problemas es, generalmente, limitado. Por ello, podemos establecer un número de bits para su representación, por ejemplo: 32 bits. Un número entero se puede representar con 32 bits usando directamente su representación en base 2. Así, el entero cero corresponde a:

```
00000000 00000000 00000000 00000000
```

Note que esto implica una restricción en el rango de posibles valores[8], ya que el entero más grande que puede representarse es:

```
11111111 11111111 11111111 11111111
```

que correspondería a 4.294.967.295. Por ejemplo, si quisiéramos representar el valor que corresponde a una unidad más, necesitaríamos 33 bits.

Por otro lado, no hemos comentado nada acerca de valores negativos. Si deseamos incluirlos, tendremos que cambiar la representación. La solución más sencilla[9] es reservar un bit (por ejemplo, el primero) para indicar el signo, y usar el resto como magnitud.

De esta forma, el rango de posibles valores cambia, ya que si disponemos de 32 bits, la magnitud se representa con 31, y por tanto, el rango de valores va desde el número -2.147.483.647 al +2.147.483.647.

> **Ejercicio 1.2** ¿Cuál es el rango de valores que puede tomar un entero que, con la representación anterior (1 bit de signo + resto de magnitud), usa dos bytes?

Finalmente, recuerde que en nuestro ejemplo cada posición de memoria la habíamos asociado a un byte, es decir, cada casilla contenía un número del 0 al 255. Con esta representación para los enteros —con 32 bits— un entero necesita 4 posiciones de memoria. Por ejemplo, si tenemos dos enteros consecutivos en memoria a partir de la posición 10, el segundo estará en la posición 14 ocupando también la 15, 16 y 17.

[7] Lo cierto es que en la práctica esta solución ha generado cierta confusión, ya que los programadores siempre hemos "vivido" en el mundo de las potencias de dos, y ahora cuando encontramos la palabra *kilobyte* no sabemos si estamos usando la antigua o nueva nomenclatura.

[8] En la práctica, estas restricciones pueden resolverse, ya que si necesitamos tratar con rangos más amplios, siempre podemos diseñar nuevos tipos de datos (como veremos en temas posteriores).

[9] Existen otras soluciones, como la representación en complemento a 2. Sin embargo, no entraremos en detalles aquí (véase por ejemplo Prieto[24]).

Representación de reales

En este caso la situación es un poco más compleja, ya que la representación directa de un número en base 2 no es válida. El problema es que es deseable poder representar números muy pequeños y muy grandes, por lo que necesitaríamos demasiados bits. Para resolverlo, expresamos el número como:

$$Mantisa \cdot Base^{Exponente}$$

y lo almacenamos dividiendo los bits disponibles en 3 campos:

1. *Signo*: 1 bit.
2. *Característica*: bits para representar el exponente.
3. *Mantisa*: El resto de bits para representar la mantisa.

Por ejemplo, si codificamos usando base 2, cualquier número puede ser escrito como:

$$1.mantisa \cdot 2^{exp}$$

donde indicamos explícitamente el número uno, ya que el número lo normalizamos para que tenga ese valor como parte entera; tenga en cuenta que si tanto la mantisa como el exponente se escriben en binario, sólo hay ceros y unos. Ahora bien, ¿Cómo llevamos un número escrito de esta forma a la memoria del ordenador?

Primero decidimos cuántos bits queremos "gastar" para guardar un número con decimales. Si usamos, por ejemplo, 32 bits para su codificación, podemos establecer 3 campos:

1. *Signo*: 0 indica positivo, 1 negativo
2. *Característica*: 8 bits que guardan el entero positivo $127 + Exp$ (se suma 127 para que también pueda haber negativos).
3. *Mantisa*: 23 primeros bits de $mantisa$ (truncamos el resto).

Con esta codificación, es importante notar que:

- Tenemos un rango limitado. No se puede representar cualquier exponente, ya que sumándole 127 debe caber en 8 bits ($exp \in [-127, 128]$). Por tanto, no son posibles números muy grandes o muy pequeños (cercanos a cero).
- Perdemos precisión. Tenemos que truncar la mantisa, aunque generalmente la cantidad de cifras que nos ofrece —en nuestro ejemplo, 23 bits— es bastante grande para la mayoría de las aplicaciones.

No intente memorizar los detalles de esta representación. No corresponde a este curso este estudio tan detallado a bajo nivel. Lo más importante ahora es darse cuenta de las posibilidades que nos ofrece y las limitaciones con las que nos encontraremos. En la práctica, verá que cuando programamos no estamos pensando en estos detalles.

Representación de caracteres

La computadora maneja dígitos binarios, así que representar caracteres parece menos natural que valores numéricos. Sin embargo, el problema es más simple, ya que el conjunto de posibles caracteres que tenemos que representar es relativamente pequeño.

Para obtener la codificación, establecemos una correspondencia uno a uno entre los elementos a codificar y una secuencia de ceros y unos.

Por ejemplo, imaginemos que queremos codificar las 27 letras del alfabeto. Necesitamos al menos 27 secuencias de ceros y unos. Por simplicidad, hacemos que estas secuencias sean de la misma longitud, y por tanto, necesitamos al menos 5 bits (capaces de generar 32 secuencias distintas). Así, hacemos que la letra `'a'` se corresponda con *00000*, `'b'` con *00001*, etc. De esta forma, podemos guardar en la memoria cualquier secuencia de caracteres compuesta por letras.

Si volvemos al ejemplo de programa que vimos anteriormente, podemos encontrar un mensaje (*"Introduzca un número:"*) que debía aparecer al usuario. Notemos que nuestra codificación de

27 letras no es suficiente para esta secuencia, ya que encontramos otros caracteres como `'I'` (en mayúscula no es lo mismo que `'i'`), dos espacios, el carácter `':'`, el carácter `'ú'`.

Además de eso, es conveniente que todo el mundo se ponga de acuerdo, ya que cuando los ordenadores se intercambien información, se pasarán estas secuencias de bits, que deben interpretar de forma idéntica.

Existen distintos estándares de codificación de caracteres. El más extendido es el *ASCII*[10], que utiliza 7 bits (128 caracteres posibles). En éste, no se incluyen caracteres como las vocales acentuadas u otros que se limitan a ciertas regiones. Para la zona de América y Europa Occidental se propone una extensión [11] a 8 bits que añade todos los caracteres necesarios para estas regiones, por lo que será el que nosotros usemos. De hecho, cuando hagamos referencia al código *ASCII*, supondremos que nos referimos a esta codificación de 8 bits.

Es interesante destacar que en esta codificación se incluyen también:

- *Caracteres de control*. Por ejemplo, existe un carácter de salto de línea, retorno de carro o para generar un pitido.
- *Caracteres especiales*. Aquí incluimos por ejemplo el espacio en blanco, el tabulador, *, &, ?, etc.
- *Caracteres gráficos*. Del tipo $|, \clubsuit, \top, \vdash$, etc. que permiten, por ejemplo, dibujar recuadros en terminales que sólo escriben texto (caracteres *ASCII*).

Como ejemplo, podemos indicar la forma en que se codificaría el siguiente texto:

```
	Hola:
  Bienvenido al curso.
     Antonio.
```

Como la secuencia de números —uno por cada 8 bits— 9, 72, 111, 108, 97, 58, 13, 32, 32, 66, 105, 101, 110, 118, 101, 110, 105, 100, 111, 32, 97, 108, 32, 99, 117, 114, 115, 111, 46, 13, 32, 32, 32, 32, 32, 32, 65, 110, 116, 111, 110, 105, 111, 46, 13.

Como sabemos, estos números se almacenarán en la memoria como secuencias de bits. Por tanto, realmente serán códigos en binario, en nuestro caso, bytes. Serían los siguientes (separando las líneas):

```
00001001 01001000 01101111 01101100 01100001 00111010 00001101

00100000 00100000 01000010 01101001 01100101 01101110 01110110
01100101 01101110 01101001 01100100 01101111 00100000 01100001
01101100 00100000 01100011 01110101 01110010 01110011 01101111
00101110 00001101

00100000 00100000 00100000 00100000 00100000 00100000 01000001
01101110 01110100 01101111 01101110 01101001 01101111 00101110
00001101
```

Es interesante notar la codificación de los caracteres de control. Por ejemplo, antes de la letra `'H'` (*ASCII* 72) sólo hay un carácter 9 (tabulador), mientras que antes de la letra `'A'` (*ASCII* 65) hay varios espacios (*ASCII* 32). Después del punto final (*ASCII* 46), se incluye también un salto de línea (*ASCII* 13).

Finalmente, es interesante destacar que el código *ASCII* contiene las letras del alfabeto en orden alfabético y consecutivas. Por ejemplo, el carácter `'a'` tiene el valor 97, el `'b'` el 98, hasta

[10] American Standard Code for Information Interchange.
[11] Se denomina *Latín-1*, del estándar *ISO 8859-1*.

el `'z'` que tiene el 122. Esto significa, que si queremos ordenar datos de tipo carácter de forma alfabética, podemos "aprovechar" que sus códigos ya están ordenados. Así, si el ordenador sabe ordenar números, también caracteres. Sin embargo, deberemos tener en cuenta que:

- Las letras mayúsculas y minúsculas se encuentran separadas. Mientras las minúsculas se encuentran en el rango $[97,122]$, las mayúsculas están en el rango $[65,90]$.
- Los caracteres especiales (con acentos, las letras ñ,Ñ), están en la parte extendida de la tabla. Por tanto, están después de la letra *'z'*.

Por este motivo alguna vez el lector se puede encontrar con algún listado de palabras que, supuestamente se encuentran ordenadas, pero que realmente sitúan las mayúsculas antes, y las que contienen caracteres especiales en posiciones erróneas. Normalmente, es consecuencia de haber usado programas que usan esta codificación[12].

Más adelante (véase sección 7.3.1 en la página 157) volveremos al tema de la codificación *ASCII*, dando más detalles de cómo está diseñada y qué efectos tiene en nuestros programas.

Incompatibilidad de representaciones

Después de lo expuesto, es lógico considerar que las representaciones son incompatibles. Aún así, es recomendable que el lector se pare un instante para insistir y reflexionar sobre ello.

Por ejemplo, imagine que a partir de la posición de memoria *100*, almaceno la secuencia de 7 caracteres *"123.456"*. Por tanto, en la posición 100 encuentro el carácter '1' (*ASCII* 49, en binario *00110001*), hasta la posición 106 en la que encuentro el carácter '6' (*ASCII* 54, en binario *00110110*). En la figura 1.4 se presenta gráficamente esta situación.

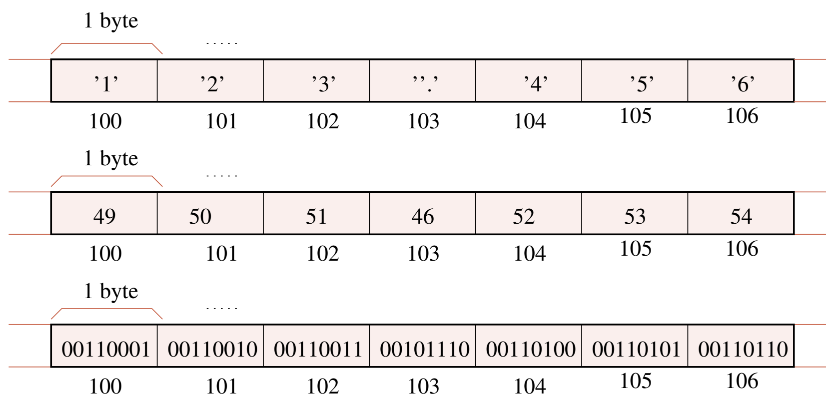


Figura 1.4
Representación de la secuencia de caracteres *"123.456"*.

Esta codificación es distinta a la de un número real, ya que para éste, se podrían usar los 32 bits iniciales (por ejemplo, las posiciones 100 a 103). En la figura 1.5 se presenta gráficamente cómo la decodificación de una secuencia de caracteres como un número real da lugar a un resultado erróneo.

De igual forma, es incompatible la codificación de un número entero y un real, a pesar de que puedan ocupar el mismo espacio (por ejemplo, 32 bits). Por lo tanto, el ordenador debe conocer exactamente la codificación que usa para sus datos, de forma que si en una posición almacenó un dato de un tipo, cuando quiera interpretar el contenido de ésta, deberá usar la codificación establecida para ese tipo.

[12]No es tan extraño encontrarlos, ya que es muy simple y es válida para el lenguaje inglés que no contiene esos caracteres especiales.

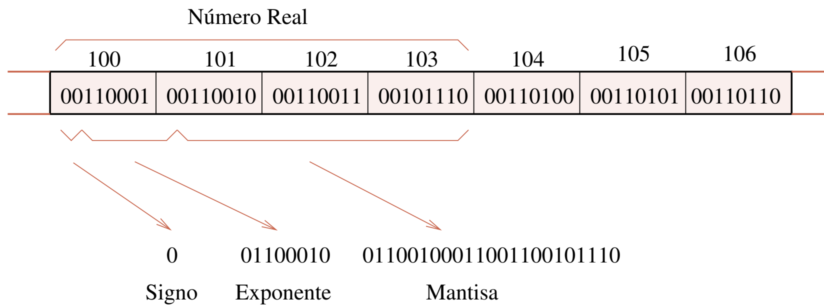


Figura 1.5
Error de decodificación de una secuencia como número real.

1.3 Lenguajes de programación

Seguro que el lector conoce programas que realizan tareas complejas y que, seguramente, se componen de millones de instrucciones. Lógicamente, el problema de desarrollar la secuencia de instrucciones que incluye uno de estos programas parece algo difícil de llevar a cabo —si no imposible— por una persona.

En este caso, la distancia entre el problema y la solución parece muy grande (véase figura 1.6). Es necesario establecer métodos para poder pasar de uno a otro.

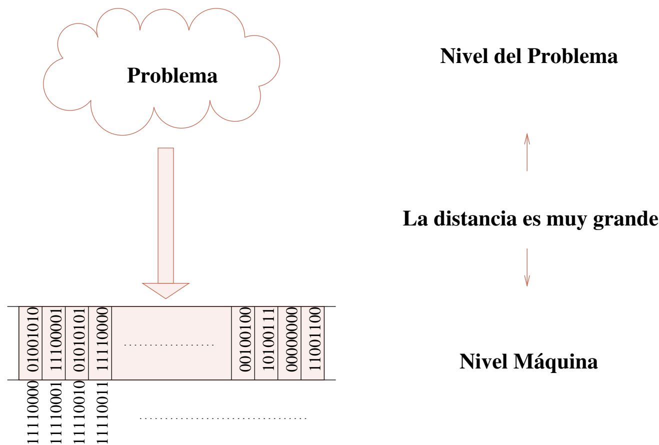


Figura 1.6
Del problema al programa.

Definición 1.5 — Lenguaje máquina. Denominamos *lenguaje máquina* al código (conjunto de instrucciones) que entiende directamente un procesador.

Por ejemplo, cuando mostrábamos el funcionamiento de un computador, el número 17 era una instrucción para llevar un dato de la memoria al procesador, el número 35 indicaba sumar, o el 19 llevar desde la CPU a la memoria. Estas 3 son instrucciones que entiende el procesador y por tanto, son parte del lenguaje máquina. Este lenguaje tiene serios inconvenientes:

- El repertorio de instrucciones es *limitado y simple*. Cualquier tarea, independiente de su dificultad, se debe descomponer en un conjunto de instrucciones de este repertorio. Imagine una persona que, usando un lápiz, sólo sabe obedecer cuatro instrucciones:
    1. *Levanta lápiz*.
    2. *Baja lápiz*.
    3. *Avanza*. Se mueve 1 milímetro en la dirección actual.
    4. *Gira*. Cambia la dirección actual girando 1 grado a la derecha.

  Y usted quiere darle las instrucciones necesarias para realizar cierto retrato. Sin duda, determinar todas estas instrucciones es una tarea difícil y laboriosa.
- El repertorio de instrucciones *depende del procesador*. Distintos fabricantes diseñan procesadores que difieren en las instrucciones y codifican las instrucciones de distinta forma. Por ejemplo, imagine que en el ejemplo anterior, usamos un procesador que entiende el 17 como realizar una suma, en lugar de entender que se quiere llevar un dato al procesador. O incluso, entiende el 35 como una división, que no era parte del conjunto de instrucciones del primer procesador.

Debido a la dificultad de este lenguaje, debemos crear otros métodos para obtener las instrucciones máquina que necesita un ordenador para realizar una tarea. La solución está en los *lenguajes de alto nivel*. Son lenguajes más potentes, ya que contienen más instrucciones y más complejas. Por ejemplo, imagine que para el problema del dibujante, creamos un lenguaje de mayor nivel, incluyendo:

1. Linea x. Pinta una línea de longitud x milímetros.
2. Girar x. Cambia la dirección actual girando x grados.

Ahora resulta mucho más fácil realizar dibujos con líneas rectas. Por ejemplo, podemos dibujar un cuadrado con lado 10 con:

1. Linea 10. Girar 90.
2. Linea 10. Girar 90.
3. Linea 10. Girar 90.
4. Linea 10.

Ahora bien, este conjunto de instrucciones no las entiende directamente el dibujante (no está en su repertorio). Antes de dárselo al dibujante, tenemos que “traducirlo”. Así, la instrucción *Linea x* se traduce en $x+2$ instrucciones de su repertorio: *Bajar Lápiz*, *Avanza* (x veces), *Levantar Lápiz*.

Esta misma idea es la que se aplica para la programación de ordenadores, para los que se crean *lenguajes de alto nivel*, más potentes, junto con programas *traductores*. El programador escribe en alto nivel, aunque este programa no lo entiende directamente el procesador. Para poder ejecutar el programa, un programa traductor lo transforma del lenguaje de alto nivel al lenguaje máquina.

La distancia entre el problema y la solución (programa en lenguaje de alto nivel) es menor, ya que ahora se dispone de más instrucciones y de mayor potencia.

Además, como muestra la figura 1.7, una ventaja muy importante es que es posible disponer de la solución en diferentes máquinas. Note que si disponemos de un programa traductor en dos sistemas totalmente distintos (por ejemplo, tienen dos lenguajes máquina diferentes), podemos desarrollar una única solución —en el lenguaje de alto nivel— que se traducirá en distintas versiones dependiendo del sistema destino.

Definición 1.6 — Portabilidad. Un sistema software es *portable* si es válido (se puede traducir y ejecutar) en distintos sistemas, sin que sea necesaria ninguna modificación.

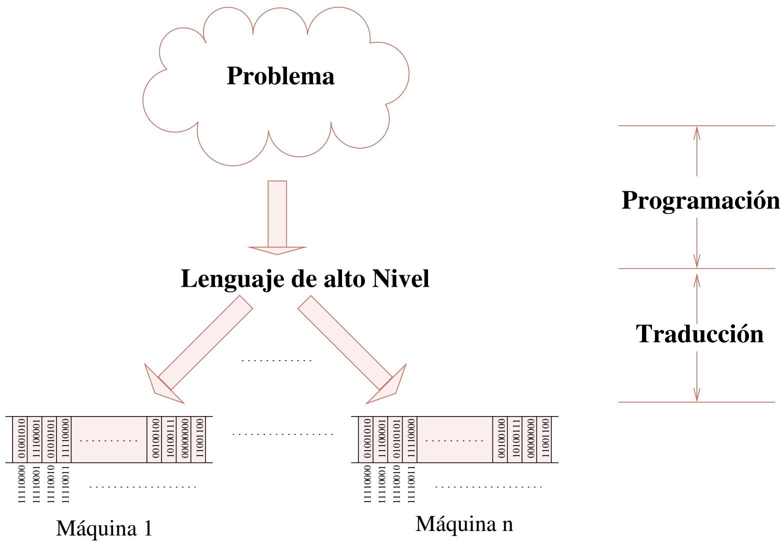


Figura 1.7
Lenguaje de Alto nivel.

1.3.1 El lenguaje C++

La historia de los lenguajes de alto nivel es muy larga; tiene más de medio siglo. Durante este tiempo han aparecido múltiples lenguajes, incorporando nuevas cualidades y especializándose para los problemas y las metodologías que iban surgiendo.

En 1972, se crea el lenguaje C como el lenguaje para el desarrollo de programas en los sistemas *Unix*. Debido a que varios fabricantes empezaron a desarrollar versiones de C que empezaban a sufrir distintas variaciones, se determinó la necesidad de llegar a un acuerdo. En 1989 se crea el estándar *ANSI* de C (conocido por C89).

El hecho de que haya un estándar es fundamental, ya que fabricantes de computadoras que diseñan sistemas muy diferentes pueden ofrecer la posibilidad de usar exactamente el mismo estándar. Así, los programadores pueden escribir programas que se pueden llevar a muchos computadores, a pesar de sus diferencias. Un fabricante crea un sistema y un traductor del estándar a su sistema. A partir de ese momento, todos los programas que seguían el estándar se pueden traducir a ese nuevo sistema.

A principios de los 80, *Bjarne Stroustrup* creó el lenguaje C++ como una extensión de C y evolucionó hasta el estándar de 1998 (conocido como C++98). Posteriormente, aparecieron algunas correcciones hasta llegar al estándar de 2003 (también conocido como C++03).

Finalmente y tras bastantes años, aparece un nuevo estándar en 2011 —conocido como C++11— en el que se han incluido importante modificaciones que lo hacen mucho más completo y potente. De forma similar a como ocurrió con el C++98, también aparecen algunas correcciones y cierta ampliación en el estándar de C++14. A pesar de todo, aquí expondremos fundamentalmente el lenguaje C++98, ya que muchos compiladores y herramientas ya desarrolladas requieren este lenguaje, por lo que un programador de C++ debería conocer las diferencias entre los dos estándares. Además, el código C++98 sigue siendo válido en C++14, por lo que podemos dejar el estudio de las novedades para un tema más avanzado.

El nuevo lenguaje C++ no es una simple modificación de algunos aspectos de C, sino que además, se le ha dotado de nuevas posibilidades, destacando sobre todo la programación dirigida a objetos, la programación genérica o la meta-programación.

Un programa en lenguaje C++, que corresponde al ejemplo anterior de suma de dos números, es el siguiente:

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

int main()
{
    int n1;
    int n2;
    int res;

    cout <<"Introduzca un número: ";
    cin >> n1;
    cout <<"Introduzca un número: ";
    cin >> n2;
    res= n1+n2;
    cout << "El resultado de la suma es: " << res << endl;
}
```

donde no aparecen detalles de *bajo nivel* sobre la máquina donde se ejecuta. De hecho, este programa se puede usar para resolver nuestro problema en muchas máquinas distintas; en cualquiera que tenga un programa traductor de C++ conforme al estándar.

Finalmente, es interesante destacar que los lenguajes evolucionan, e incluso aparecen otros nuevos que se adaptan cada vez mejor a las necesidades y metodologías actuales. Por ejemplo, el lenguaje C también se ha modificado hasta un nuevo estándar (C11) que sigue evolucionando independientemente de C++.

También han aparecido otros, entre los que podemos destacar *Java* y *C#* después de C++, y que también se han beneficiado de las aportaciones que realizó éste.

1.3.2 Desarrollo de software

En principio, el desarrollo de software se refiere a la tarea de crear un programa que resuelva un problema, es decir, consiste en saber cómo pasar del problema al programa en un lenguaje de alto nivel.

Pensemos en un problema sencillo; por ejemplo, supongamos que queremos que un programa lea la siguiente ecuación:

$$a \cdot x + b = 0$$

y escriba la solución. Para resolverlo, debemos indicar al ordenador los pasos que debe seguir para obtener la solución, es decir, el algoritmo que resuelve el problema.

> **Definición 1.7 — Algoritmo.** Un *algoritmo* es un conjunto finito y ordenado de pasos o instrucciones para obtener la solución a un problema.

Lógicamente, cada uno de esos pasos debe ser preciso (no ser ambiguo) y el algoritmo debe acabar en un tiempo finito.

Note que si queremos obtener una solución a nuestro problema de programación, los pasos que se especifiquen en el algoritmo deben poder especificarse en el lenguaje de la solución. Así, podemos decir que el algoritmo consiste en:

1. Leer los valores a, b.
2. Calcular $x = -b/a$
3. Escribir como resultado el valor x.

que en C++ podría escribirse como sigue:

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

```
3
4 int main()
5 {
6     double a;
7     double b;
8     double x;
9
10    cout << "Considere la ecuación ax+b=0." << endl;
11
12    cout << "Introduzca el valor de a: ";
13    cin >> a;
14
15    cout << "Introduzca el valor de b: ";
16    cin >> b;
17
18    x= -b/a;
19    cout << "La solución de la ecuación es: " << x << endl;
20 }
```

donde podemos ver que el primer paso del algoritmo se ha resuelto en las líneas 10-16, el segundo en la 18, y el tercero en la 19.

En otro ejemplo, podemos plantear un problema en el que un programa lee una imagen (obtenida con nuestra última cámara de fotos digital), recorta un trozo seleccionado y la presenta, tras un zoom y un aumento del contraste que nos revela esa persona oculta que no sabíamos distinguir. En este caso, la tarea es muy compleja como para poder escribir, de forma directa, el programa que la resuelve.

El algoritmo necesario para resolver un problema mucho más complejo no es tan fácil de obtener. Para ello, será necesario desarrollar una metodología que simplifique el camino, desde el problema hasta la solución, en términos de un lenguaje.

Por otro lado, el lector puede entender que el objetivo es obtener cualquier programa que resuelva el problema del *usuario*[13]. Sin embargo, la ingeniería del software tiene como objetivo obtener programas de "calidad", es decir, programas fiables (que no fallen), que sean eficientes, y que puedan ser modificados (ya sea para resolver fallos del programa o para adaptar modificaciones del problema).

En el apéndice C (página 329) se presenta una introducción a la ingeniería del software de forma más detallada. A pesar de ello, es importante insistir en este enfoque desde este primer tema, ya que lo arrastraremos a lo largo de todo este libro como el motivo por el que dos programas, a pesar de obtener los mismos resultados, no serán igualmente válidos.

Lógicamente, los problemas que resolvamos irán creciendo en dificultad de forma que no será tan sencillo como plantear un simple algoritmo y realizar la *implementación* (escritura de la solución en un lenguaje de alto nivel). El lector irá aprendiendo a enfrentarse a este tipo de problemas de complejidad creciente conforme estudie los temas, a medida que vamos introduciendo nuevos conceptos y herramientas.

1.3.3 El proceso de traducción

El proceso de traducción desde un lenguaje como C++ a lenguaje máquina lo realiza un programa "externo" previamente disponible. Existen dos tipos de programas para realizar la traducción:

- *Intérpretes*. Realizan una traducción sentencia a sentencia[14], y las van ejecutando.
- *Compiladores*. Realizan una traducción completa desde el lenguaje de alto nivel a lenguaje máquina.

Los intérpretes traducen la primera sentencia a lenguaje máquina, detienen la traducción, ejecutan la sentencia, traducen la siguiente sentencia, la ejecutan, etc.

[13] La persona o sistema que luego usará el programa

[14] Llamamos sentencia —del inglés *sentence*— a cada una de las instrucciones de alto nivel que componen el programa.

Los compiladores traducen todo el programa. Así, no es necesario disponer del compilador cuando necesitemos ejecutar el programa. El programa se traduce una vez y se ejecuta tantas veces como queramos. En nuestro caso, utilizaremos un compilador de C++.

El proceso de traducción lo realiza un compilador. Esta etapa parece muy sencilla, ya que no es necesario que intervenga el programador. Sin embargo, tras escribir un programa en C++, no hemos terminado en absoluto con el trabajo. No nos podemos olvidar que no somos perfectos y el programa seguro que contendrá errores. Hablaremos de dos tipos de errores:

- *Errores de compilación y enlazado*. No hemos escrito correctamente el programa y no corresponde al lenguaje. El compilador genera errores de traducción que el programador debe interpretar para solucionar. Por ejemplo, si después de la línea de `main` no escribimos un carácter —{— de apertura de llaves el compilador genera un error de sintaxis en el momento de la compilación.
- *Errores de ejecución*. Se genera un error cuando el programa ya está traducido y lo estamos ejecutando. Generalmente se deben a errores de programación, es decir, que nos hemos equivocado en los algoritmos que deberían resolver el problema. Por ejemplo, si escribimos correctamente una operación de división de dos valores enteros, se podría generar un error en tiempo de ejecución en caso de que el divisor valga cero[15].
  Algunos autores distinguen además los *errores lógicos*, cuando el programa termina correctamente, pero con una solución equivocada.

Los errores de compilación y enlazado son, con diferencia, los más sencillos de resolver. Ahora bien, para poder resolverlos debemos conocer bien el lenguaje, incluyendo la forma en que realiza la traducción.

Para obtener el fichero ejecutable se realizan dos pasos:

1. *Compilación*. Es la traducción propiamente dicha. Se traduce el fuente —ficheros en lenguaje C++— a ficheros "traducidos", es decir, a ficheros que contienen nuestro código pero en el lenguaje destino.
2. *Enlazado*. Se une el resultado de la compilación con los recursos necesarios para obtener el ejecutable.

Como ejemplo, volvamos al programa fuente de la sección anterior, en el que obtenemos la solución de una ecuación de primer grado. El programa fuente está en formato texto, es decir, está compuesto por una secuencia de caracteres *ASCII* que, supuestamente, sigue las reglas que especifica el lenguaje.

La etapa de compilación obtiene un programa traducido, que denominamos *programa o fichero objeto*. En la figura 1.8 se presenta un esquema de esta traducción. Como entrada, el programa fuente se encuentra almacenado en un fichero que denominamos, por ejemplo, *primer_grado.cpp*. El compilador traduce este fichero, generando un fichero objeto denominado *primer_grado.o*.

Figura 1.8
Del fuente al objeto.

El compilador realiza la traducción leyendo el programa fuente desde el primer carácter al último (en ese orden), es decir, que podemos decir que "lee" el programa desde la primera línea hasta la última.

[15]La operación de división por cero puede provocar un error fatal que detiene el programa.

Durante este proceso, mantiene una serie de tablas de símbolos que contienen información sobre su situación en cada instante, las cosas que ha traducido, y lo que necesita para seguir (podríamos decir que son tablas donde escribe, "en sucio", todo lo que necesita). Las etapas que se siguen son:

1. *Análisis léxico*. Inicialmente tenemos una secuencia de caracteres. El primer paso consiste en convertirla a una secuencia de *tokens*. Por ejemplo, podemos recibir la secuencia de caracteres :
   cin>>a;res=-a/3.14;
   Y el analizador léxico la descompone en algo como:
   "identificador cin", "operador >>", "identificador a", finalizador ;", "identificador res", "operador =", "operador -", "identificador a", "operador /", "real 3.14", "finalizador ;"
2. *Análisis sintáctico*. Comprueba que la secuencia de *tokens* sigue las reglas sintácticas que determina el lenguaje. Por ejemplo, si se está usando el operador suma en una expresión, la sintaxis indica que se escribe un operando, seguido del operador +, y seguido por otro operando. Si el programador escribe dos veces el operador +, la sintaxis no es correcta, y el compilador genera un error indicando que no se esperaba encontrar otro operador.
3. *Análisis semántico*. En esta etapa se comprueba que el programa, que sintácticamente es correcto —está bien escrito— tiene sentido. Por ejemplo, puedo escribir una operación correctamente como *Operando1 Operador Operando2*, sin embargo, no es correcto si lo que estamos es sumando un número entero, con una cadena de caracteres.
4. *Generación de código*. El programa es correcto y por tanto se puede generar el código objeto que le corresponde.
5. *Optimización*. El compilador revisa el código generado para optimizarlo, eliminando operaciones innecesarias, cambiando de orden las operaciones, etc. Por ejemplo, imagine que calcula la expresión $(a+b)*5/(a+b)$. En el código generado, puede descubrir que la expresión $(a+b)$ se calcula dos veces, entonces modifica el código para que sólo se haga una vez. A lo largo de este libro iremos indicando algunos casos en los que el compilador podría aplicar cambios para la optimización.

Finalmente, el programa objeto no es directamente ejecutable y es necesario aplicar un paso más —el *enlazado*— para obtener el resultado final. En esta etapa, se une el resultado de la compilación con otros recursos externos que son necesarios para nuestro programa. Como es de esperar, también podemos obtener errores, denominados de *enlazado*, en este último paso.

En la figura 1.9 se presenta un esquema de esta etapa. Como entrada, el fichero objeto se encuentra almacenado, por ejemplo, en *primer_grado.o*. El *enlazador* une este fichero con otros ficheros y recursos, generando un fichero ejecutable denominado *primer_grado.exe*.

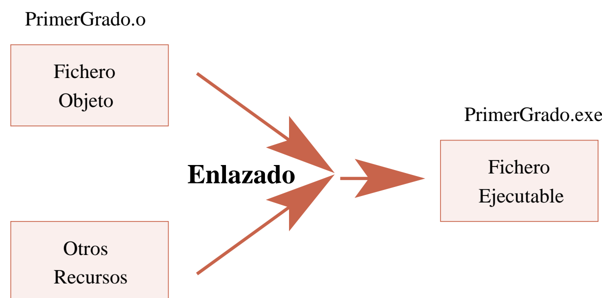


Figura 1.9
Del objeto al ejecutable.

Más adelante, sobre ejemplos concretos, detallaremos esta última etapa. Así mismo, un estudio más amplio sobre este proceso, y que incluye otras posibilidades, se presenta en el tema 11 (página 235).

El preprocesador

En C++, podemos decir que la compilación consiste realmente en dos etapas:

1. *Preprocesamiento*.
2. *Traducción*, que corresponde a la compilación propiamente dicha.

La primera etapa consiste en un procesamiento previo del fichero fuente para que un programa —el preprocesador— analice la existencia de instrucciones "especiales" que se deben llevar a cabo antes de que el código entre al compilador. Este tipo de instrucciones se denominan *directivas* del preprocesador, y se distinguen porque comienzan con una carácter `#` al principio de la línea.

Un ejemplo es la directiva `#include` que va seguida de un nombre de archivo que contiene, igualmente, código C++. Cuando se recorre el código fuente, al llegar a este punto el preprocesador detecta esta orden, localiza el fichero referenciado y, como efecto, la elimina del programa sustituyéndola con el código C++ que contiene el archivo indicado.

Como vemos, realmente no se realiza ningún tipo de compilación (recuerde, análisis léxico, sintáctico, etc.), sino que sólo se sustituye una línea por otras. La directiva `#include` no es más que una instrucción para que realice una operación de edición, sin ningún tipo de traducción. De hecho, si le decimos que incluya un archivo que no es código C++, lo incluiría sin ningún problema, aunque luego provocaría múltiples errores cuando intentara analizar lo que ha incluido. Es decir, es una etapa previa a la compilación, de ahí que se llame *preprocesador* o *precompilador*.

A pesar de esta distinción, cuando hablemos de forma general sobre la compilación nos referiremos a las dos etapas. El lector encontrará más detalles sobre esta directiva y otras en el tema 11 (página 235).

1.4 Problemas

Problema 1.1 Responda con verdadero y falso a las siguientes afirmaciones. En caso de que sea falso, explique brevemente la razón.

- En un esquema simplificado de un computador, los datos se almacenan en la memoria masiva, mientras los programas se guardan en la memoria central.
- Cuando almacenamos un dato en la memoria principal, se localiza en una posición de memoria, por ejemplo, en la posición 193. Esto significa, que si colocamos otro dato justo a continuación de éste, se localizará en la posición 194.
- El espacio de memoria para almacenar el literal cadena de caracteres `"12345.6789"` es del mismo tamaño que para almacenar el número real `12345.6789`.
- Disponer de un estándar o "acuerdo" sobre un lenguaje es una gran ventaja porque nos permite traducir cualquier programa a un lenguaje máquina universal.

Problema 1.2 Considere el programa ejemplo que escribe la solución de una ecuación de primer grado. Sin consultar más detalles sobre el lenguaje, intente deducir cómo se implementaría un programa que lee el radio de una circunferencia y escribe su longitud.
Nota: el producto se especifica con el carácter $*$.

2 Introducción a C++

Un programa básico 21
Datos y expresiones 23
Entrada/Salida básica 30
Funciones de biblioteca 35
Inicialización de variables y constantes 37
Problemas 38

2.1 Un programa básico

En principio, consideraremos que un programa C++ implementa el conjunto ordenado de instrucciones indicadas en el módulo denominado **main**. Por lo tanto, la ejecución de un programa consiste en localizar **main** y seguir todas sus instrucciones (delimitadas por dos caracteres '{','}') hasta que finalice.

El programa más simple posible es el que no hace nada y por tanto se escribiría como:

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

donde vemos que para especificar el módulo **main**, debemos escribir `int main()`. A continuación, las instrucciones del programa (en este caso, ninguna). Más adelante se estudiará en detalle por qué esta línea tiene esta sintaxis. En nuestros primeros programas, simplemente escribiremos esa secuencia para establecer el comienzo del módulo que contiene las instrucciones del programa.

Un ejemplo algo más complejo es el siguiente:

```
1 #include <iostream>  // Para usar cout, endl
2 using namespace std; // Usamos el estándar
3
4 int main()
5 {
6    cout << "¡Bienvenido!" << endl;
7 }
```

Donde podemos observar:

- En la línea 1 y 2 se incluyen los caracteres `//`. El compilador entiende estos caracteres como indicadores de comienzo de comentarios.
  Para poder leer mejor nuestros códigos podemos incluir cualquier comentario en el punto que deseemos del programa. Para ello, tenemos que indicar al compilador que no son parte del programa. Lo podemos realizar de dos formas:
  1. Escribir dos caracteres `//`. Todo lo que sigue a estos caracteres hasta final de línea se ignora. Nos permitirá escribir un comentario en una línea.

2. Escribir los caracteres /∗ (como comienzo) y ∗/ (como fin). Todo lo que se encuentra entre el comienzo y el fin se ignora. Nos permitirá escribir un comentario que ocupe varias líneas.

- En la línea 1, se incluye una directiva del precompilador. En este caso, el precompilador localiza el fichero con nombre **iostream** e incluye —al principio del programa— el código C++ que contiene. En principio, sólo debe tener en cuenta que, al incluir este archivo, permitimos que el programa sea capaz de realizar entradas y salidas, por ejemplo, por el teclado y la pantalla.
- En la línea 2 aparece una línea indicando que se utilizará el espacio de nombres **std** . El estudio de los espacios de nombres se dejará hasta un tema posterior. En principio, podemos interpretarlo como la orden para que el compilador nos permita usar directamente las herramientas que nos ofrece el estándar.
- Línea 4-5. Comienzan las instrucciones que corresponden al programa que se debe ejecutar.
- Línea 6. Indica que se escriba en la salida el mensaje seguido de un final de línea, es decir, termina avanzando a la siguiente línea. En esta línea usamos **cout** y **endl**, que son dos identificadores del estándar. Disponemos de ellos al haber incluido **iostream**, y podemos usarlos directamente al indicar que usamos el espacio de nombres **std**.

Por tanto, para comenzar el curso de programación en C++, consideraremos que un programa tiene la siguiente estructura:

```
[Inclusión de archivos cabecera]
using namespace std;

int main()
{
[Instrucciones]
}
```

donde la parte indicada por corchetes corresponde al código que debemos escribir para implementar nuestra solución, es decir, los pasos que corresponden al algoritmo que resuelve el problema.

Note que en la inclusión de archivos hemos indicado que son de *cabecera*. Esto se debe a que estos archivos contienen información y se escriben especialmente para ser incluidos al comienzo del archivo. De ahí que se denominen de cabecera[1].

2.1.1 Escritura del fichero fuente

Para poder usar el ejemplo anterior tendremos que introducirlo en el ordenador, realizar la traducción y ejecutarlo. Por tanto, en primer lugar tendremos que escribirlo como un archivo fuente.

Como indicamos, este fichero consiste en la secuencia de caracteres que componen el programa. Por tanto, necesitaremos usar un *editor* de texto que permita introducir estos fuentes almacenándolos en formato estándar *ASCII*. Damos un nombre a este fichero, y ya estamos preparados para la compilación.

El nombre del fichero puede ser cualquiera, aunque procuraremos que sea significativo y corresponda al contenido del fichero, es decir, al problema que resuelve. Como extensión usaremos [2] *cpp*, indicando así que contiene código C++. Por ejemplo, podemos nombrar nuestro ejemplo con el nombre *bienvenido.cpp*.

En este punto podemos lanzar el proceso de traducción para obtener otro archivo —en este caso ejecutable— del programa fuente. El nombre de este archivo final puede estar predefinido en

[1] Veremos cómo en sus nombres se puede incluir la extensión .h que proviene del inglés *header*.
[2] Existen otras extensiones estándar, como son *cxx* o *C*.

nuestro compilador o podemos indicar uno nosotros. Por ejemplo, podemos indicar que queremos traducir al archivo *bienvenido.exe*.

Una vez que se traduce, podemos pedir al sistema que se ejecute este archivo, obteniendo como resultado el mensaje *"¡Bienvenido!"*.

Formateo del fichero fuente

Respecto al formato de escritura del fichero fuente, el lenguaje C++ es un lenguaje denominado de *formato libre*, es decir, no es necesario que "formateemos" el fuente de ninguna forma especial. Sólo es necesario tener en cuenta que el analizador léxico deberá obtener los *tokens* correctamente. Para ello, se considera que cualquier carácter espacio, salto de línea, tabulador, etc. —cualquier "espacio blanco"— determina una separación entre *tokens*. Por ello, para el analizador léxico todos ellos son equivalentes, e incluso varios de ellos consecutivos lo son.

Como ejemplo de este comportamiento, podemos reescribir el mismo programa con el mismo resultado si lo reformateamos como sigue:

```
1  #include <iostream>  // Para usar cout, endl
2                       using
3  namespace
4                 std
5
6  ; // Usamos el estándar
7
8  int main
9  (
10 )
11 {
12                        cout
13   <<
14   "¡Bienvenido!"  /* no sirve */  << endl
15                      ;
16 }
```

donde vemos que los espaciados respetan el mismo conjunto de *tokens* e incluso hemos añadido un comentario en medio de una línea que lógicamente se ignorará. Como el lector intuirá, intentaremos escribir nuestros programas con un formato como el primero, que nos permite leer más fácilmente el código.

Es importante que note que todos los espacios que hemos introducido en el programa no son más que una forma distinta de separar los mismos elementos, los mismos *tokens* que componen el programa. Para enfatizar este aspecto, considere el siguiente programa:

```
1 #include <iostream>  // Para usar cout, endl
2 using namespace std; // Usamos el estándar
3
4 int main()
5 {
6    cout < < "¡Bienvenido!" << endl;
7 }
```

que parece prácticamente idéntico al anterior. Sin embargo, observe que en la línea 6 se producen dos tokens "<" (después de `cout`) en lugar de uno sólo "<<". En este caso, se ha producido una secuencia distinta, por lo que el programa ha cambiado. De hecho, el compilador detiene la traducción indicando un error. Más adelante veremos que las dos alternativas no tienen el mismo significado.

2.2 Datos y expresiones

En nuestros programas necesitamos procesar información de entrada para obtener la salida deseada. El sistema debe ser capaz de usar datos —números, mensajes, nombres, valores reales, etc.— y operar con ellos para obtener esos resultados.

El lenguaje C++ nos ofrece distintos *tipos de datos predefinidos*, junto con una serie de operadores sobre ellos, que nos permiten construir expresiones. Algunos de los tipos básicos que ya vienen en el lenguaje y que usaremos intensivamente a lo largo de este curso son:

- `int`: Tipo entero.
- `double`: Tipo en coma flotante (número reales).
- `char`: Tipo carácter (por ejemplo la letra `'a'`, el espacio `' '`, el salto de línea, etc.).
- `bool`: Tipo booleano. Sólo puede tomar dos valores: `false` o `true`.

El que un tipo de dato sea *predefinido* indica que se encuentra ya en el lenguaje, es decir, que el compilador conoce cómo se almacena (sus valores), cómo se escribe (la sintaxis), y qué significa cada una de sus operaciones.

Por ejemplo, si tenemos que hacer un programa que gestione las calificaciones obtenidas por los alumnos de primer curso de una carrera, podemos usar un valor de tipo `int` para indicar el número de alumnos que hay en un grupo, un valor `double` para indicar la calificación obtenida en un examen, un valor `char` para codificar el grupo al que pertenece un alumno o un valor `bool` para indicar si un alumno ha superado la parte práctica de la asignatura.

2.2.1 Variables

Los datos que usa un programa se almacenan en memoria (recuerde los contenidos del tema anterior sobre representación de datos). En C++ no nos tenemos que preocupar de la zona de memoria donde se almacena cada dato, ya que el compilador la asigna de forma automática.

Definición 2.1 — Variable. Una variable es un nombre para una zona de memoria que almacena un valor de algún tipo determinado.

Por ejemplo, imagine que deseamos escribir un programa para calcular la media aritmética de dos valores reales. Podemos considerar los siguientes pasos:

1. Leer el valor del primer número real.
2. Leer el valor del segundo número real.
3. Calcular un nuevo número calculado como la media aritmética.
4. Escribir este nuevo número como resultado del programa.

Para escribirlo en C++ utilizamos variables. Por ejemplo, podemos pensar que el proceso consiste en leemos un número y lo *"guardamos"*, leemos el segundo y lo *"guardamos"*, para calcular la suma *"recuperamos"* el primero y el segundo, para obtener el resultado y *"guardarlo"*. Los conceptos de "*guardar*" y "*recuperar*" se corresponden con los de escribir en memoria o leer de memoria, respectivamente.

En la práctica, lo único que tenemos que hacer es seleccionar el tipo de dato que deseamos y darle un nombre. Por tanto, para poder usar una variable, lo primero que tenemos que realizar es la *declaración de un objeto* en la que informamos del tipo de dato que almacena y del nombre que referencia a ese dato.

Es en la declaración cuando el compilador "detecta" la necesidad del dato, y por tanto cuando "prepara" la zona de memoria donde se localizará. Todo ello, de forma automática.

La forma de una declaración de variable es:

```
<tipo de la variable> <nombre de variable> ;
```

donde el tipo de la variable corresponden a uno de los tipos conocidos por el compilador (por ejemplo, `int` o `bool`) y el nombre de la variable es un identificador que nosotros seleccionamos. Termina con el carácter `';'` que indica al compilador el fin de la declaración.

El nombre de la variable tiene que estar formado por una secuencia de caracteres compuesta por letras del alfabeto inglés (es decir, no son válidos caracteres como vocales acentuadas o la letra `'ñ'`), dígitos o el carácter `'_'`. Además, el nombre no puede comenzar por un dígito.

La declaración es fundamental para el compilador, no sólo porque conoce el nombre de la variable, sino porque tiene que preparar una zona de memoria para manejarla. Por ejemplo, el tipo indica el tamaño de la zona de memoria a reservar. Además, una vez que avancemos en el programa, el compilador conoce el tipo de dato que corresponde a la variable, por lo que podrá interpretar las expresiones que la usen. Encontrar un identificador que no haya sido declarado es un error de compilación.

Ejemplos de declaración son:

```
int valor;
double resultado;
char un_caracter;
bool apto;
```

Ejemplos de declaraciones incorrectas son:

```
int dato          // Falta ;
double 1val;      // Comienza por 1
bool -logico;     // Comienza por -
char una letra;   // Tiene un espacio
```

Finalmente, los identificadores de variables no pueden ser ninguna de las palabras reservadas del lenguaje (ver tabla D.2, página 346). Por ejemplo, la palabra **`int`** está reservada para indicar un tipo entero, por tanto, el usuario no puede usarla para identificar una variable.

> **Ejercicio 2.1** Considere la siguiente lista de identificadores:
> - *hola, primero, 1dato, datonúmero2, año, double, resultado, el_valor_5.*
>
> Indique los identificadores que son válidos en C++.

2.2.2 Literales

Un literal es un valor concreto de un tipo. Por ejemplo,

- Un literal de tipo **`int`** se escribe como un conjunto de dígitos (0-9) consecutivos (precedido con un signo `'-'` para los negativos). Ejemplos: `123, -37, 5.`
- Un literal de tipo **`double`** como un conjunto de dígitos, tal vez precedido con el signo `'-'`, que incluye el carácter punto (por ejemplo, `-1.23` o `5.`). Además, es posible usar la notación científica añadiendo una letra `'e'` y un exponente; por ejemplo, `5.2e-7`.
- Un literal de tipo **`char`** diseñado para almacenar un carácter se escribe con el carácter entre comillas simples. Por ejemplo, `'A'` corresponde al carácter *ASCII* 65, o `' '` (espacio) corresponde al carácter *ASCII* 32.
- Un literal de tipo **`bool`** (booleano) se especifica a través de las palabras **`true`** o **`false`**. En este caso sólo existen estos dos posibles literales.
- Un literal *cadena de caracteres* se escribe como una secuencia de caracteres entre comillas dobles. Por ejemplo, `"Mensaje"`. Note que no es lo mismo un carácter entre comillas simples que entre comillas dobles, puesto que entre dobles no es un carácter, sino una cadena de longitud uno.

Es importante observar que para el caso de los caracteres o las cadenas de caracteres es necesario escribir el carácter entre comillas simples o dobles respectivamente. Sin embargo, este formato no es posible para todos los caracteres. Algunos ejemplos de caracteres que no se pueden expresar de esta forma son:

- No podemos escribir una comilla simple entre comillas simples. El carácter se entendería como "cierra comillas".

- De forma similar, no podemos escribir unas comillas dobles dentro de una cadena de caracteres. El compilador creería que ha terminado la cadena.
- ¿Cómo se escribe el carácter tabulador?
- ¿Cómo se escribe el salto de línea?
- etc.

Para resolver este problema se usa la barra invertida (`'\'`) para indicar al compilador que se desea especificar un carácter especial. En la tabla 2.1 se presentan algunos de estos códigos.

Tabla 2.1
Ejemplos de códigos de barra invertida.

Código	Significado
\'	**Comilla simple**
\"	**Comilla doble**
\n	**Salto de línea**
\t	**Tabulador**
\r	**Retorno de carro**
****	**Barra invertida**

Por ejemplo, los siguientes son literales válidos:

- Caracteres: `'\n'`, `'\t'`
- Cadenas de caracteres: `"\tComienza con tabulador"`

Ejercicio 2.2 Identifique el tipo de los siguientes literales:

- *"X","Hola","","Y',1345,30.0,true,15,"false",12.0,'\"'*

2.2.3 Operaciones

Los tipos de datos de C++ no serían útiles sin un conjunto de operaciones asociadas. Cuando disponemos de un nuevo tipo, no sólo somos capaces de representar la información, sino que podemos operar con ella.

Por ejemplo, para el tipo **`int`**, podemos escribir expresiones con los operadores de suma, resta, producto, división, etc. Con ellos, podríamos escribir la expresión:

$$\frac{a+b}{c}$$

Lógicamente, para escribirlas en un programa tenemos que usar una notación que permita especificarla como una secuencia de caracteres. Los operadores de suma, resta, producto y división se escriben mediante los caracteres $+-*/$. En nuestro ejemplo, podemos escribir la expresión como la siguiente secuencia:

```
(a+b)/c
```

Note que hemos añadido paréntesis, ya que el producto y la división tienen más prioridad que la suma y la resta. Conocer la prioridad de los operadores es fundamental, ya que afecta al resultado de la expresión. Al escribir las expresiones como una secuencia de caracteres es necesario conocer el orden de evaluación. Éste viene predeterminado en el lenguaje (ver apéndice D, página 343). Para modificarlo podemos, como hemos visto, usar paréntesis.

Por ejemplo, si no hubiéramos puesto los paréntesis en la expresión anterior, ésta hubiera correspondido a:

$$a+\frac{b}{c}$$

Además, no sólo tenemos que establecer el orden de estos cuatro operadores, sino de todos los que podría aparecer en la expresión. Por ejemplo, otros operadores con enteros son el menos unario, y el operador módulo (`%`). El orden de evaluación de una expresión con enteros vendrá determinado por el siguiente orden de precedencia:

1. Paréntesis[3].
2. Menos unario. Por ejemplo, en la expresión `a/-b + c` el menos unario sobre `b` se aplica antes que la división o la suma.
3. Producto, división, módulo.
4. Suma, resta.

Además, es necesario conocer el orden de evaluación o *asociatividad* para operadores de la misma precedencia, ya que si aparecen varios operadores de la misma precedencia, el resultado puede ser distinto si evaluamos de derecha a izquierda o al revés. Por ejemplo,

```
a/b*c
```

tiene dos operadores de igual precedencia que darían lugar a distintos resultados dependiendo del orden de evaluación. Como norma general, los operadores binarios se evalúan de izquierda a derecha[4]. Así, en el ejemplo anterior, primero se realiza la división y el resultado se multiplica por `c`. Si quisiéramos que `a` se dividiera por el resultado del producto, tendríamos que añadir paréntesis para darle más prioridad.

Para el caso de los números reales —que hemos visto que podemos almacenarlos con el tipo **`double`**— también existen operadores similares de forma que podemos escribir expresiones con datos de tipo real. Al igual que antes, tenemos casi los mismos operadores —eliminando el de módulo— con la misma precedencia.

Ejercicio 2.3 Supongamos las variables `a,b,c,d` de tipo **`double`**. Escriba en C++ las siguientes expresiones:

$$\frac{a+\frac{b}{cd}}{a+b}$$

$$a+\frac{b+c}{d}+\frac{b}{c}-b$$

División de enteros vs división de reales

Es importante saber que el operador de división puede parecer distinto para enteros que para reales. Realmente hacen lo mismo: la división. Sin embargo, puede sorprender que el resultado parece distinto, ya que en el caso de los enteros no existen los decimales.

Cuando dividimos un entero entre otro, obtenemos como resultado otro entero. Por tanto, el operador corresponde a la división entera. Por ejemplo `7/2` devuelve como resultado `3`.

En el caso de los reales es similar: cuando dividimos dos números reales obtenemos como resultado otro número real. En el ejemplo anterior, si hacemos `7.0/2.0` obtenemos `3.5`.

[3] Aunque existe el operador () en C++, aquí sólo aparece como una forma de cambiar el orden de evaluación en una expresión y no como un operador propiamente dicho.

[4] Más adelante, veremos que los operadores de asignación se evalúan de derecha a izquierda.

Conversiones aritméticas

Es probable que con los ejemplos anteriores, el lector se esté preguntando acerca de la "mezcla" de distintos tipos. Efectivamente, podríamos escribir la expresión `7.0/2` que corresponde a la división de un número **`double`** por un número **`int`**. En este caso, cuando se realiza una operación entre dos valores de distinto tipo, se realiza una conversión para que los dos tengan el mismo. En nuestro ejemplo, se convierte el entero a real, de forma que se lleva a cabo una división de valores **`double`**.

En este sentido, el lector no debería preocuparse por la mezcla de tipos que almacenan valores numéricos (enteros y reales), al menos para los ejemplos de esta primera parte. Podemos confiar en que el compilador realiza las conversiones necesarias.

Podemos decir que el compilador no sabe realizar operaciones con tipos mezclados, ya que o se hace una división entera o se hace real. Sin embargo, cuando aparezcan mezclas el compilador conoce la relación entre los dos tipos de forma que realiza las conversiones aritméticas que permiten realizar el cálculo con aritmética de punto flotante, es decir, con el tipo de dato que evita la pérdida de información.

2.2.4 Operador de asignación

Una operación especialmente importante, definida para todos los tipos básicos, es la de *asignación*. La forma para escribirla sigue el siguiente formato

```
<variable> = <expresión> ;
```

El operador corresponde al carácter = y se usa situando a su izquierda una referencia a un objeto[5] en memoria —por ejemplo, una variable— y a su derecha un nuevo valor para ese objeto. Por ejemplo, podemos escribir una expresión con un simple literal:

```
x = 3 ;
```

para hacer que el valor de la variable `x` sea `3`. O con una expresión:

```
x = (a+b)/2 ;
```

para hacer que `x` pase a contener el resultado de evaluar la expresión `(a+b)/2`.

Además, en la parte derecha puede haber cualquier expresión, incluso conteniendo la variable de la izquierda. Por ejemplo, podemos hacer:

```
x = 3 ;
x = x*2 ;
```

en la primera línea asignamos el valor `3` a `x` y, en la segunda, asignamos el resultado de multiplicar `x` por `2`. Aunque matemáticamente pueda aparentar ser una barbaridad, el proceso de asignación consiste[6] en:

1. *Evaluar la parte derecha*. Por lo tanto, se usa el antiguo valor de `x`.
2. *Realizar la asignación*. Es decir, actualiza el valor de `x` con el resultado obtenido en el paso anterior.

> **Ejercicio 2.4** Suponga dos variables `x`, `y` que contienen dos valores desconocidos. Escriba las sentencias necesarias para hacer que los valores de esas variables se intercambien.

[5] Consideraremos objeto como "algo en memoria", es decir, una región de almacenamiento contiguo en memoria. El lector también puede encontrar en la literatura el concepto de *l-value* (valor-i) como una expresión que referencia a un objeto (constante o no).

[6] Realmente el proceso corresponde a la evaluación de una expresión que incluye el operador de asignación, de menor prioridad. Sin embargo, para estos temas, no entraremos en detalles de cómo funciona el lenguaje.

Tipos de la variable y la expresión

En las secciones anteriores hemos presentado distintos tipos de datos. Una variable tiene asociado un tipo de dato, almacena un valor que corresponde a ese tipo. En una asignación el valor que se obtiene desde la expresión debe almacenarse en la variable, por lo que es de esperar que sean del mismo tipo.

Efectivamente, en la mayoría de los casos —especialmente al principio del curso— los tipos de la expresión y la variable coincidirán. Sin embargo, de forma similar a las conversiones aritméticas que hemos presentado cuando se mezclan datos, el compilador es capaz de realizar la asignación si sabe convertir el tipo de dato de la expresión al de la variable. En concreto, en este nivel del curso nos interesa lo que ocurre si los dos tipos involucrados son el tipo **int** y **double**.

La asignación de un entero a una variable que almacena un número real no parece muy problemática. Los números reales también pueden corresponder a valores sin decimales. Efectivamente, es de esperar que se realice sin problemas. Sin embargo, cuando asignamos un número real a un entero no es así. Por ejemplo, imagine el siguiente programa:

Listado 1 — Programa *triangulo.cpp*.

```
1  #include <iostream>  // Para usar cout, endl
2  using namespace std; // Usamos el estándar
3
4  int main()
5  {
6     double base, altura;
7
8     cout << "Introduzca la base y altura del triángulo: ";
9     cin >> base >> altura;
10
11    int superficie;
12    superficie= base*altura/2;
13
14    cout << "La superficie correspondiente es: " << superficie << endl;
15 }
```

Una posible ejecución podría ser la siguiente:

Consola

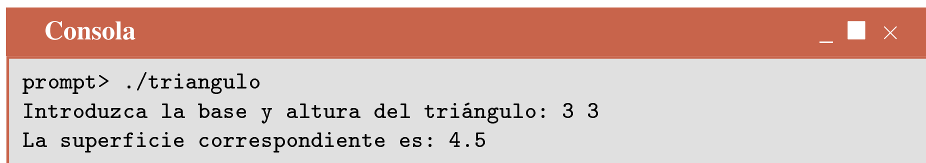
```
prompt> ./triangulo
Introduzca la base y altura del triángulo: 3 4
La superficie correspondiente es: 6
```

que es perfecta, pues ese triángulo tiene, efectivamente, superficie 6. Sin embargo, también podríamos probar con los siguientes valores:

Consola

```
prompt> ./triangulo
Introduzca la base y altura del triángulo: 3 3
La superficie correspondiente es: 4
```

donde podemos ver que el resultado es incorrecto, pues la superficie debería ser 4.5. El problema está en el tipo de la variable `superficie` que hemos indicado en la línea 11. La asignación que se realiza en la línea 12 calcula un valor con decimales, pero lo asigna a una variable de tipo **int**. Esta variable no puede almacenar decimales. El compilador acepta la asignación pero trunca el resultado. Si la declaramos de tipo **double**, la ejecución será:



El problema de la mezcla de datos de distinto tipo aparecerá repetidamente a lo largo del estudio de C++. Es un aspecto fundamental que hay que manejar para poder programar en este lenguaje. A este nivel del curso, lo importante es saber que tenemos distintos tipos de datos, pero que el compilador sabe mezclar automáticamente. Por ahora, bastará que recuerde que podemos pasar de entero a real sin problemas y de real a entero eliminando la parte decimal.

2.3 Entrada/Salida básica

El computador se relaciona con el exterior a través de los dispositivos de entrada/salida (E/S). Un programa también debe implementar este intercambio de información.

Por ejemplo, la E/S se puede realizar *de forma gráfica*: mostrando *ventanas* con botones, menús, casillas de texto para introducir datos, mientras la salida se muestra en otras ventanas. En este caso, hablamos de *interfaz gráfica de usuario*[7].

Definición 2.2 — Interfaz. Una *Interfaz* se refiere a la conexión física y funcional entre dos sistemas independientes. Denominaremos *interfaz de usuario* a la parte del sistema que se encarga de la comunicación entre el programa y el usuario.

El lector probablemente conoce varios medios para realizar la E/S de datos en un programa. Por ejemplo, gráficamente, E/S de datos de archivos, o incluso intercambio con otros programas en *Internet*. La programación de estos sistemas de comunicación es muy compleja y no forma parte del lenguaje de programación ya que depende en gran medida del sistema (por ejemplo, el sistema gráfico usado en *Unix* difiere en gran medida por el usado en *Windows*) y no es fácil llegar a un estándar[8].

El lenguaje C++ ofrece una forma estándar de comunicación que, siendo simple, es muy potente. En esta sección veremos únicamente los elementos más básicos para desarrollar los capítulos siguientes.

Cuando se lanza un programa C++, se realiza una asignación de los *flujos* de E/S estándar `cin`, `cout`, `cerr`[9]. Podemos pensar en el programa como una entidad que tiene un conector de entrada y dos de salida, como muestra la figura 2.1. Cuando ejecutamos el programa, el sistema *"conecta"* un flujo de datos en la entrada estándar, que el programa conoce con el identificador `cin`, un flujo de datos en la salida estándar, que el programa conoce con el identificador `cout`, y otro de salida estándar de error, conocido con `cerr`.

Todo este proceso se realizar de forma automática, por lo que el programador no tiene de qué preocuparse. Lo que sí debe tener en cuenta es:

- Estos identificadores no están disponibles directamente. Para poder usarlos, será necesario incluir el fichero de cabecera `iostream`, es decir, escribir una línea al principio del programa con la directiva `#include` del preprocesador:

[7] En inglés, *Graphical User Interface (GUI)*.

[8] En este sentido, es interesante nombrar otros lenguajes —como Java— que utiliza una máquina virtual para su ejecución. De esta forma, es más fácil implementar bibliotecas estándar que resuelvan estos y otros problemas.

[9] Podríamos hablar de un cuarto, `clog` (aunque se asigna al mismo sitio que `cerr` con la diferencia de que éste es un flujo sin *buffer*) o sus correspondiente extendidos. Es un tema demasiado avanzado para este momento. Volveremos sobre este asunto en el capítulo 10.

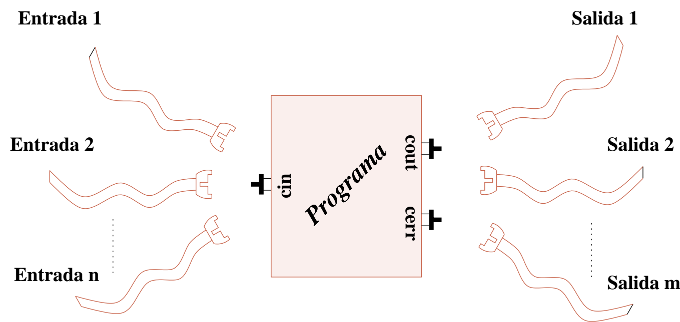


Figura 2.1
Flujos de E/S estándar.

```
#include <iostream>
```

- No sabemos de dónde proceden los datos que entran o a dónde van los que salen. Sólo sabemos que un *flujo de datos* es una secuencia de caracteres. Por ejemplo, si entra el número flotante `125.34`, lo que entra por **`cin`** es la secuencia `'1'`, `'2'`, `'5'`, `'.'`, `'3'`, `'4'`. Podría venir de la consola —la escribimos en el teclado— o de otro lugar, como un fichero o incluso otro programa.

Por tanto, para el programa no es importante lo que se haya “conectado” a la entrada o la salida. Lo único a tener en cuenta es que podemos solicitar datos de entrada desde **`cin`** o sacar datos de salida en **`cout`** y **`cerr`**.

En nuestro caso, normalmente no especificaremos los recursos asociados a la entrada y la salida —al menos en los primeros capítulos— por lo que supondremos que se asignan los de por defecto: el teclado para la entrada y la pantalla tanto para la salida estándar como la de error (figura 2.2).

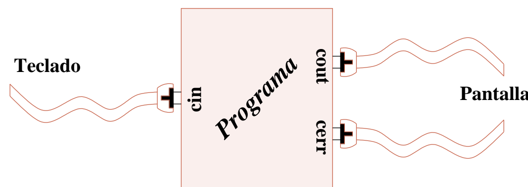


Figura 2.2
Flujos de E/S estándar por defecto.

2.3.1 Operadores de E/S estándar

Usaremos los identificadores del flujo de E/S estándar con los operadores >> y << para realizar las operaciones de entrada y salida, respectivamente. Por tanto, si deseamos leer un dato desde la entrada estándar, usaremos la sintaxis:

```
cin >> destino
```

indicando que leemos un dato desde **`cin`** para guardarlo en *`destino`*, donde éste puede ser, por ejemplo, una variable. Note que no es necesario indicar en el programa cómo leer el tipo asociado al destino. Por ejemplo, si entra la secuencia `'1'`, `'2'`, `'5'`, `'.'`, `'3'`, `'4'` y *`destino`* es de tipo **`double`**, leeremos el valor `125.34`, pero si *`destino`* es de tipo **`char`**, leerá el carácter `'1'` (el

resto de caracteres esperarán a otra lectura). Como el compilador conoce el tipo, la interpretación es automática, y nosotros no nos debemos ocupar de la forma en que se resuelve el problema [10].

Por otro lado, si deseamos escribir un dato en la salida estándar (o la salida de error estándar) usamos una sintaxis parecida:

```
cout << origen
```

indicando que escribimos el dato `origen` en **`cout`**. El tipo de dato determina la forma en que se debe escribir, de manera que el programador no tiene que preocuparse de la forma en que se realiza.

Finalmente, es posible encadenar varias operaciones de entrada (o de salida) en una misma línea. Por ejemplo, si queremos leer un entero y un **`double`**, podemos hacer:

```
int dato1;
double dato2;
    // ... otras líneas ...
cin >> dato1 >> dato2;
```

O si queremos escribir un resultado de tipo **`double`** junto con un mensaje podríamos escribir:

```
double resultado;
    // .... otras líneas.....
cout << "El resultado es: " << resultado ;
```

donde el primer dato a escribir es un *literal* de tipo cadena de caracteres, y el segundo dato un valor real.

Ejercicio 2.5 Indique el resultado de la ejecución del siguiente trozo de código:

```
double f;
int e;

f= 9;
e= 2;
cout << f/e     << ' ' << 9/2       << ' ' << 9/2.0 << ' '
     << f/e*1.0 << ' ' << 9/(e*1.0) << ' ' << 1.0*9/e <<
     << ' ' << 9/e*1.0;
```

2.3.2 Ejemplo

Como ejemplo de E/S y de algunos apartados anteriores en este tema, mostramos un programa que implementa el problema de realizar una suma de dos números reales. El listado es:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6     double a;
7     double b;
8     double suma;
9
10    cout << "Introduzca el valor de a: ";
11    cin >> a;
12
13    cout << "Introduzca el valor de b: ";
14    cin >> b;
15
16    suma= a+b;
17    cout << "La suma es: " << suma << endl;
18 }
```

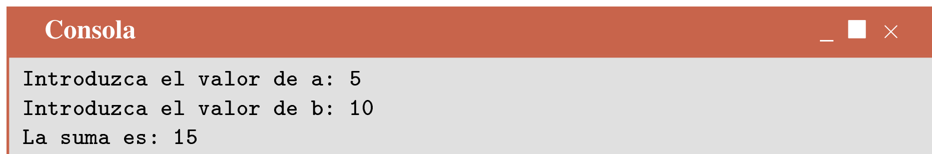
[10]Realmente, si queremos obtener un programa robusto y suponemos que la entrada puede ser errónea (por ejemplo damos una letra cuando espera un número), deberíamos comprobar los posibles errores. A pesar de ello, supondremos que no se comenten errores. Para solucionarlo, el lector puede consultar los detalles del sistema de E/S de C++. Volveremos sobre este asunto en el tema 10.

Las instrucciones o sentencias del programa vienen especificadas en líneas distintas, aunque para el compilador lo que las separa es el carácter de fin `';'`. Recuerde que para el analizador léxico no es importante el "formato" que le demos a las sentencias.

Comentarios específicos para las distintas sentencias son:

- Hemos incluido el fichero **iostream**, ya que usamos los identificadores **cin**, **cout**, **endl** definidos en él. Además, indicamos que usamos el espacio de nombres **std**, ya que estos identificadores son del estándar de C++[11].
- El programa se encuentra entre las dos llaves que delimitan el *cuerpo* de **main**.
- En las líneas 6-8: declaramos 3 variables de tipo **double** para almacenar un valor real. Por tanto, cuando ejecutamos el programa, lo primero que hace es reservar memoria para tres datos de tipo real.
- En la línea 10: realizamos una operación de salida en la que especificamos un literal de tipo cadena de caracteres, que se escribe en la salida estándar (pantalla).
- En la línea 11: realizamos una operación de entrada. Se lee desde el teclado un número real que se guarda en *a*.
- En las líneas 13-14: se repite la operación, ahora para leer el dato *b*.
- En la línea 16: se lleva a cabo una asignación. En la variable *suma* se almacena el resultado de evaluar la expresión *a+b*.
- En la línea 17: escribimos tres datos en una sentencia de salida múltiple. Podemos ver que también hemos usado el identificador **endl** como dato de salida. Éste está definido en **iostream** y tiene como efecto que se escribe un salto de línea en la salida.

Una ejecución típica del programa puede ser:



donde en la primera línea el usuario ha escrito el carácter *'5'* seguido por un *"Intro"*, y en la segunda ha escrito los caracteres *'1','0'* seguidos por otro *"Intro"*.

Un programa similar para el mismo problema podría haber sido:

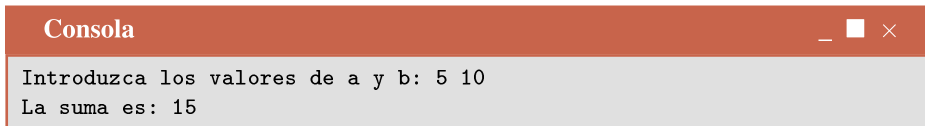
```
#include <iostream>
using namespace std;

int main()
{
   double a, b;

   cout << "Introduzca los valores de a y b: ";
   cin >> a >> b;
   cout << "La suma es: " << a+b << endl;
}
```

donde podemos observar que:

- Se han declarado dos valores **double** en una misma línea. Efectivamente, el lenguaje permite declarar varias variables en una misma línea, separándolas con comas.
- La entrada se especifica en una única línea. En este caso, se entiende que es necesario introducir dos valores distintos de tipo **double**.
- En la última línea especificamos directamente *a+b* por lo que se entiende que deseamos escribir el resultado de esa expresión.

Una ejecución típica del programa puede ser:

[11] Más adelante veremos que podríamos prescindir de esta línea, por ejemplo, escribiendo **std::cin** en lugar de **cin**.



donde puede observar que los valores de entrada múltiples se pueden escribir si los separamos con espacios blancos.

Ejercicio 2.6 Escribir un programa que lea —desde la entrada estándar— la base y altura de un rectángulo, y escriba —en la salida estándar— su área y perímetro.

2.3.3 Entrada de datos. Algunos detalles

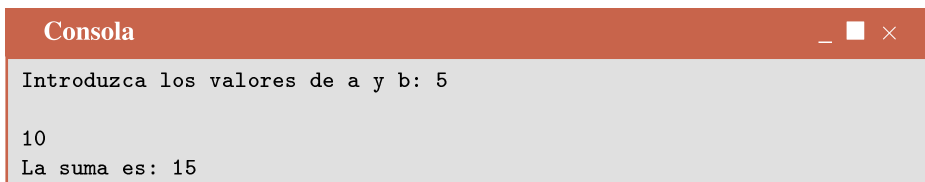
En el ejemplo anterior hemos visto cómo el usuario introduce dos datos numéricos separados por un espacio, de forma que el programa los introducía en dos variables independientes. El lector podría preguntarse sobre otras posibilidades. Por ejemplo, ¿y si introduzco una letra en lugar del número? ¿y si introduzco dos datos separados por un tabulador, o por un salto de línea, o por varios espacios?

Para simplificar el proceso de E/S, en principio, asumiremos que los datos se introducen correctamente, es decir, que si el sistema espera un número en la entrada se encontrará ese número. Si no fuese así, el sistema detecta un error y el programa debería dar algún tipo de respuesta para resolverlo. Lógicamente, el lector podrá realizar estas comprobaciones cuando conozca más a fondo el sistema de E/S de C++.

Por otro lado, es necesario conocer el funcionamiento del operador de entrada >> para entender el formato de entrada de datos. Cuando pedimos la entrada de un dato con este operador, se supone que la entrada tiene un "formato libre"; se empiezan a leer caracteres desde el flujo de entrada considerando que los datos pueden estar separados con "espacios blancos", es decir, por espacios, tabuladores, saltos de línea, etc. Esto significa, por ejemplo, que cuando se solicita un entero el sistema comienza a leer caracteres que, si son separadores, se descartan. Cuando se lee un signo o un número, comienza la lectura del entero, lectura que continúa hasta que el siguiente carácter de entrada deja de ser un número.

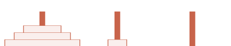
El lector puede pensar en la entrada como un sistema de flujo de caracteres que tiene una llave de apertura. Cuando se pide un dato, se abre la llave y salen todos los caracteres separadores. Cuando se detecta el primer carácter del dato a leer comienza su lectura —decodificación— hasta que el siguiente carácter a entrar deja de corresponder a este dato (normalmente, será un carácter separador), momento en el que se cierra la llave.

Por tanto, en nuestro último ejemplo, se hubieran leído correctamente los dos valores reales aunque hubiéramos dejado varios espacios o incluso varios saltos de línea. Por ejemplo, podríamos haber obtenido el siguiente resultado:



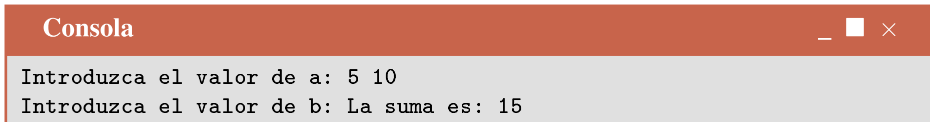
al escribir dos caracteres *"Intro"* entre los dos datos.

Un ejemplo un poco más complejo; pensemos en el programa anterior, cuando leíamos dos valores de forma independiente:



```
// ...
cout << "Introduzca el valor de a: ";
cin >> a;
cout << "Introduzca el valor de b: ";
cin >> b;
// ...
```

Podríamos haber respondido a la primera pregunta incluyendo los dos valores, con lo que hubiéramos obtenido el siguiente comportamiento:



donde vemos que en la primera pregunta hemos introducido los dos números separados por un espacio. Observe que después de escribir el número 5 y un espacio no se pasa a leer el segundo valor. Los caracteres no se mandan al programa mientras no pulsemos *"Intro"*.

Al pulsar *"Intro"*, el programa recibe toda la secuencia de caracteres. Cuando el programa dispone de caracteres de entrada, se realiza la lectura de un valor real, por tanto se "abre" el flujo de entrada, detectando el valor numérico 5, y se vuelve a "cerrar" al encontrar el separador espacio.

Cuando termina la lectura de `a`, se pasa a la siguiente línea del programa; se escribe el mensaje para que el usuario introduzca `b` y pasamos a su lectura. En este instante, el programa tiene caracteres esperando en el flujo, por tanto, se "abre", nos saltamos los separadores —un espacio— y leemos un dato que termina con el último salto de línea introducido. Como hemos llevado a cabo la lectura, el programa escribe la última línea de resultado.

Si piensa en el comportamiento del programa, las instrucciones de lectura desde **`cin`** no implican que el programa se pone en espera para que el usuario use el teclado. Cuando el programa se para, es porque necesita un dato y el flujo no contiene aún nada que ofrecer, como ha ocurrido cuando hemos pedido el primero de ellos.

En nuestros experimentos, normalmente responderemos a las peticiones conforme se formulan, ya sea introduciendo un dato y pulsando *"Intro"* o, en caso de que se pidan varios en una línea, introduciéndolos separados por un espacio y confirmando finalmente con *"Intro"*. Por tanto, no será habitual encontrarnos con un diálogo de E/S con un aspecto tan extraño como el anterior. Sin embargo, es importante que vaya conociendo estos detalles, especialmente porque le serán necesarios[12] para problemas más avanzados.

2.4 Funciones de biblioteca

Las expresiones aritméticas que hemos visto en las secciones anteriores ofrecen un conjunto de posibilidades limitado debido a que muchos problemas requieren cálculos más complejos; por ejemplo, el cálculo de la raíz cuadrada. El lenguaje no ofrece un operador para este cálculo y, por tanto, sería necesario implementar un algoritmo que obtenga el valor deseado.

Sin embargo, esto tipo de funciones son muy útiles y habituales en muchos problemas. No tiene sentido que el programador tenga que realizar un esfuerzo adicional para resolver éste u otros problemas similares.

El lenguaje C++ ofrece una serie de funciones que implementan cálculos ya resueltos, funciones que se guardan en "bibliotecas". Cuando el programador quiere evaluar la raíz cuadrada de un

[12] En estos primeros temas, aunque no es necesario, el lector puede encontrar útil esta forma de introducir datos cuando realice experimentos en el laboratorio y quiera simplificar la forma de darlos. Sobre todo en problemas donde haya que introducir repetidamente las mismas entradas.

número, sólo es necesario llamar a la función correspondiente. Para generar el programa ejecutable, se traduce nuestro fuente —que hace referencia a esa función— a un fichero objeto, al que se añaden las funciones de biblioteca que hayamos usado, en un proceso denominado de *enlazado*.

En este momento, para estos primeros capítulos, haremos uso de alguna de estas funciones. En capítulos posteriores sobre funciones y modularización volveremos sobre este proceso y conoceremos los detalles de su funcionamiento.

Para utilizar estas funciones debemos incluir los archivos cabecera que indican, entre otras cosas, sus nombres, de forma que nuestro programa pueda usarlos. Así, si queremos utilizar la función para obtener la raíz cuadrada, tendremos que incluir el archivo de cabecera **`cmath`**, que nos permite usar el identificador **`sqrt`** para obtener la raíz cuadrada de un número real. Por ejemplo, **`sqrt`**`(9.0)` devuelve el valor `3.0`.

Un ejemplo que usa esta función es:

```
#include <iostream>
#include <cmath> // Para poder usar sqrt
using namespace std;

int main()
{
   double cateto1, cateto2, hipotenusa;

   cout << "Introduzca las longitudes de los catetos: ";
   cin >> cateto1 >> cateto2;
   hipotenusa= sqrt(cateto1*cateto1 + cateto2*cateto2);
   cout << "La longitud de la hipotenusa es: " << hipotenusa << endl;
}
```

Observe que podemos usar la función en cualquier expresión indicando entre paréntesis los datos para el cálculo, en este caso, el número al que queremos obtener la raíz. Otros ejemplos de funciones incluidas en **`cmath`** son:

- *fabs(x)* devuelve un número real que corresponde al valor absoluto de x.
- *exp(x)* devuelve e^x.
- *log(x)* devuelve el logaritmo neperiano de x.
- *pow(x,y)* devuelve x^y.
- *tan(x)* devuelve la tangente de x (ángulo en radianes).
- etc.

Estas funciones son para cálculos con números reales[13]. Para otros tipos de datos, también existen otras funciones que realizan otras operaciones. Por ejemplo, para `x` de tipo **`char`** podemos usar:

- *tolower(x)* devuelve `x` en minúscula (sin modificar si no es una letra).
- *toupper(x)* devuelve `x` en mayúscula.
- *isalpha(x)* devuelve si `x` es una letra del alfabeto.
- *isdigit(x)* devuelve si `x` es un dígito.
- *isgraph(x)*, *islower(x)*, etc.

Podemos usar estas funciones sin más que incluir el archivo de cabecera **`cctype`**. Más adelante veremos otras funciones de biblioteca conforme se vayan necesitando.

Ejercicio 2.7 Escriba un programa que lea el valor de dos puntos en el plano, y escriba —en la salida estándar— la distancia entre ellos. Recuerde que ésta se calcula como:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

[13] Existen muchas más funciones matemáticas, que puede encontrar en cualquier manual de referencia de C o C++. Consulte, por ejemplo, Shildt[27].

2.5 Inicialización de variables y constantes

Supongamos que se desea realizar un programa que lea el valor del radio de un círculo y como salida escriba perímetro y área. El programa podría ser el siguiente:

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

int main()
{
   double radio;

   cout << "Introduzca el radio: ";
   cin >> radio;
   cout << "Longitud: " << 2*3.14*radio << endl;
   cout << "El área: " << 3.14*radio*radio << endl;
}
```

En las dos últimas líneas utilizamos un mismo valor (π). Sería más legible y más fácil de modificar si usamos un nombre. Podemos modificar `main` para obtener:

```
   double PI;
   double radio;
   PI= 3.14;


   cout << "Introduzca el radio: ";
   cin >> radio;
   cout << "Longitud: " << 2*PI*radio << endl;
   cout << "El área: " << PI*radio*radio << endl;
```

En este caso, si queremos modificar el valor de PI para que tenga más decimales, sólo debemos modificar un punto del programa. Sería aún mejor si le damos el valor a la variable a la vez que se declara, de esa forma en una única línea tendríamos el tipo, el nombre y el valor que queremos que almacene.

En C++ podemos declarar una variable y asignar un valor para que se cree con la *inicialización* correspondiente. La sintaxis de la declaración es:

```
<tipo de la variable> <nombre de variable> = <expresión>;
```

que hace que el compilador cree la variable a partir del resultado de la expresión. Nuestro programa —incluyendo más precisión— podría modificarse con:

```
   double PI= 3.1415926536;
   double radio;
```

Finalmente, es claro que nuestra intención es poder usar `PI` con ese valor concreto a lo largo de todo el programa. Al ser una variable, podríamos modificar su valor en cualquier punto (por ejemplo, por error al usar una ecuación que incluye π). Sería mejor si el compilador sabe que es un nombre para algo que no puede cambiar. Para ello, declaramos `PI` como una constante, añadiendo la palabra reservada `const` antes de la declaración, de la siguiente forma:

```
const <tipo de la variable> <nombre de variable> = <expresión>;
```

La operación de asignación no es válida para una constante, ya que produce un error en tiempo de compilación. Por ello, en el momento de la declaración —cuando se crea— es cuando hay que darle un valor que mantendrá durante toda su existencia. Con esta modificación, nuestro programa quedaría:

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

int main()
{
    const double PI= 3.1415926536;
    double radio;

    cout << "Introduzca el radio: ";
    cin >> radio;
    cout << "Longitud: " << 2*PI*radio << endl;
    cout << "El área: " << PI*radio*radio << endl;
}
```

Es importante enfatizar el hecho de que en la práctica este programa funciona igual que el anterior, declaremos `PI` constante o no. Sin embargo, funcionar igual no implica que sean igualmente buenos. En este simple ejemplo, note que el hecho de declarar esa constante facilita:

- El programador que lee el programa sabe que `PI` es una constante. Por tanto, si quiere usarla al final del programa tiene garantizado que tiene exactamente el mismo valor que inicialmente se dio.
- El compilador puede detectar errores de programación cuando encuentra sentencias que cambian el valor de `PI`. Le hemos dado más información al compilador sobre lo que queremos hacer. De esta forma, puede ayudarnos a programar al detectar mejor casos de sentencias que no corresponden con nuestra intención[14].
- El compilador sabe que es una constante y por tanto puede aprovechar que su valor no cambia para mejorar el código. Por ejemplo, si escribimos que se calcule el valor de `2*PI*radio` el compilador podría modificar la sentencia para que el programa calcule `6.2831853072*radio`. Es decir, como sabe que el valor de `PI` es exactamente el indicado, puede precalcular la subexpresión para hacer que la ejecución sea más eficiente.

2.6 Problemas

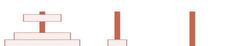
Problema 2.1 Responda con verdadero y falso a las siguientes afirmaciones. En caso de que sea falso, explique brevemente la razón.

1. Es necesario tener mucho cuidado con el espaciado de un programa. Por ejemplo, las palabras `int` y `main` deben estar en la misma línea.
2. No es posible un programa que no contenga una línea con la palabra `using`.
3. La directiva `#include` se tiene en cuenta en el paso denominado de enlazado. En ese momento, se lee su contenido para establecer los enlaces con elementos externos (como las funciones de biblioteca).
4. El literal `'a'` es equivalente al literal `"a"`.
5. Los nombres *var1, dato, resultado, 2ndo, _variable_, true, dirección, año* son identificadores válidos de variables.
6. Las variables se deben declarar antes de usarse.
7. Respecto a la forma de escribir expresiones, la siguiente equivalencia es correcta

$$a+b/c*d+e \equiv (a+(b/(c*d))+(e))$$

8. El identificador `cin` se puede usar si se incluye `iostream` y su uso implica que la entrada se realiza por teclado.
9. Cuando escribimos una sentencia de lectura de un entero desde `cin`, es necesario especificar una variable seguida por la palabra `int` para que el compilador conozca el tipo del dato.

[14] Más adelante insistiremos sobre esta idea, es decir, sobre la importancia de que el compilador conozca mejor nuestras intenciones.



10. Cuando un programa nos pide dos datos de tipo **double**, podemos escribir un número seguido de dos saltos de línea (pulsar dos *"Intro"*) para indicar que el segundo valor es vacío y por tanto cero.
11. Aprender a programar en C++ es muy fácil.

Problema 2.2 Supongamos que el usuario ha introducido las variables `x,y,alfa` (de tipo **double**) desde el teclado. Indique las sentencias necesarias para que se escriba, en la salida estándar, los valores de las siguientes expresiones:

$$\sqrt{1+\frac{x}{e^{10+sen(alpha)}}}$$

$$ln\left(1+\left|\frac{x}{e^{10+sen(alpha)}}+cos(alpha)\right|\right)$$

teniendo en cuenta que `alfa` se ha leído en grados.

Problema 2.3 Supongamos que hemos declarado dos variable `x,y` de tipo entero. Indique el resultado de las siguientes sentencias:

```
x= 5;
y= 2;
cout << "Valor de x" << x << endl;
cout << "Valor de y" << y << endl;
cout << "Valor de x/y\n" << x/y;
cout << "\tValor de y/x" << y/x << endl;
cout << "La mitad de x: " << x/2.0 << endl;
```

Problema 2.4 ¿Cuál es la salida del siguiente programa?

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

int main()
{
   double a, b;
   a= 5;
   b= 3;
   cout <<  5 + a * b + a / b * 4  << endl;
}
```

¿Y si la línea de salida hubiera sido la siguiente?

```
cout <<  6 * ( 5 - b ) + a - b / ( 5 - a ) << endl;
```

Problema 2.5 Indique los errores y modificaciones que considere oportunas para el siguiente programa.

```
include <iostream>

int main() {
   Double valor, base;

   cout << "Indique base: " ;  cin << base
   cout << "Indique valor: ";  cin >> valor;
   cout << "El log en base" << base << "de"
        << Valor << es;
   cout < < log(valor)/log(base) << end;
}
```

¿Podría producirse un error en tiempo de ejecución?

Problema 2.6 Se ha escrito el siguiente programa:

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

int main() {
   double precio, pagado;
```

```
    cout << "Escriba precio y dinero entregado: ";
    cin >> precio >> pagado;
    int centimos= 100*(pagado-precio);
    cout << "Hay que devolver: " << centimos << " céntimos" << endl;
}
```

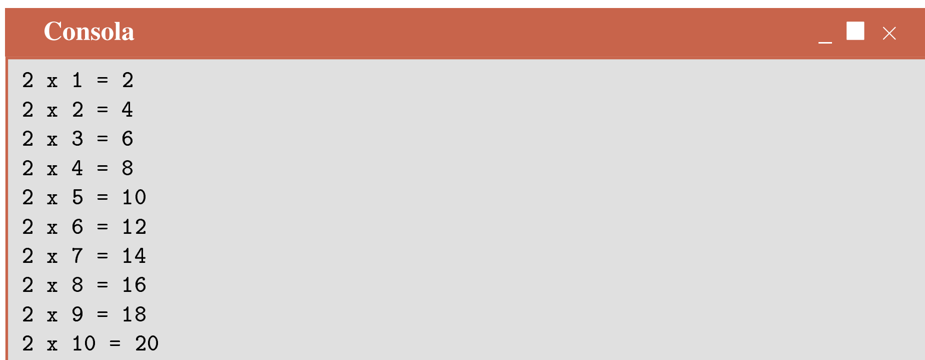
Escriba el programa y pruebe la ejecución con los datos *1.54 2*. En mi máquina, la ejecución obtiene:

```
Consola
Escriba precio y dinero entregado: 1.54 2
Hay que devolver: 45 céntimos
```

¿Cuál cree que es el problema? Proponga una solución.

Problema 2.7 Escriba un programa que pregunte dos números reales y escriba su suma, resta, producto y división. ¿Podría provocar un error en tiempo de ejecución?

Problema 2.8 Escriba un programa que lea un número de tipo entero desde la entrada estándar y escriba en la salida estándar su "tabla de multiplicar", es decir 10 líneas con el número, un carácter 'x', un entero del 1 al 10, un carácter igual y el resultado del producto. Por ejemplo, para el 2, escribiría:



```
Consola
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

Problema 2.9 Escriba un programa para calcular las dos soluciones de una ecuación de segundo grado. Para ello, leerá tres valores reales (a,b,c), que corresponden a la ecuación:

$$ax^2 + bx + c = 0$$

y escribirá, en la salida estándar, las dos soluciones. ¿Se puede producir un error en tiempo de ejecución?

Problema 2.10 Desarrolle un programa para análisis de polígonos regulares. Tiene como entradas la longitud del lado (a), y el número de lados (k). Como salida deberá escribir en la salida estándar los siguientes valores:

- El perímetro del polígono.
- El ángulo entre dos lados consecutivos ($\theta = \frac{k-2}{k} 180$ grados).
- El radio del círculo inscrito ($r = \frac{1}{2} a \cdot cot \frac{180}{k}$). Donde *cot* denota la cotangente o inversa de la tangente.
- El radio del círculo circunscrito ($R = \frac{1}{2} a \cdot csc \frac{180}{k}$). Donde *csc* denota la cosecante o inversa del seno.

- El área ($area = kr^2 tan \frac{180}{k}$). Donde *tan* denota la tangente.

Note que los ángulos están en grados y recuerde que las funciones para el cálculo del seno, coseno y tangente son **sin**, **cos**, **tan**, respectivamente.

Problema 2.11 En la evaluación de los alumnos de la asignatura de programación, se consideran tres notas, la de teoría, la de prácticas, y la de problemas. La calificación final se obtiene teniendo en cuenta que la primera corresponde a un 70%, la segunda a un 20%, y la tercera a un 10%. Escriba un programa que pida las tres notas en una misma línea —se pedirán como tres números separados por espacios— y escriba el resultado de la calificación.

Problema 2.12 Una máquina expendedora de dinero necesita calcular el número de billetes y monedas que corresponden a una determinada cantidad (sin decimales). Escriba un programa que reciba como entrada el número de euros, y escriba en pantalla la cantidad de cada tipo de moneda que corresponde. Suponga que:

1. Siempre hay disponible cualquier tipo de billete y moneda.
2. El número de billetes y monedas debe ser mínimo.

Como consejo, utilice la división entera y el módulo. Por ejemplo, si se escribe 543, al dividirlo por 500 obtenemos 1 (un billete de 500) y al hacer el módulo 500 obtenemos 43 (a calcular con moneda más pequeña).

Problema 2.13 Escriba un programa que lea un entero, correspondiente a una cantidad de segundos, y escriba en la salida estándar el número de días, horas, minutos y segundos que le corresponden. El número de horas debe ser menor que 24, y el de minutos y segundos menor que 60.

Problema 2.14 Escriba un programa que lea 3 valores reales (x_1, x_2, x_3), y escriba en la salida estándar la media y desviación típica de esos valores. Las expresiones son:

$$m = \frac{x_1 + x_2 + x_3}{3} \qquad d = \sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2}{3}}$$

Problema 2.15 En 1987, Borwein propone una forma de calcular el número π, basada en un método iterativo que usa las siguientes ecuaciones:

$$\begin{array}{lll} x_0 = \sqrt{2} & y_1 = 2^{\frac{1}{4}} & \pi_0 = 2 + \sqrt{2} \\ x_{n+1} = \frac{1}{2}\left(\sqrt{x_n} + \frac{1}{\sqrt{x_n}}\right) & y_{n+1} = \frac{y_n\sqrt{x_n} + \frac{1}{\sqrt{x_n}}}{y_n + 1} & \pi_n = \pi_{n-1}\frac{x_n + 1}{y_n + 1} \end{array} \quad (2.1)$$

Escriba un programa que escriba en la salida estándar los valores de π_0, π_1, y π_2.

Problema 2.16 Indique la salida del siguiente programa (precisando el número de espacios, tabuladores y líneas que aparecen).

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

int main()
{
   const char tabulador= '\t';

   cout << "\n¡Bienvenido al programa jeroglífico!\n\n";
   cout << "\tObjetivo: entender los caracteres \"especiales\"";
   cout << tabulador << tabulador << "Como el salto de línea \\n o"
        << " la comilla simple ' (\\')" << endl;
}
```

3 La estructura de selección

Introducción 43
Operadores relacionales. Selección simple .. 44
Selección doble *if/else* 47
Expresiones booleanas. Operadores lógicos . 50
Selección múltiple *switch* 53
Booleanos en C. Algunos detalles 54
Representación gráfica 56
Problemas 56

3.1 Introducción

En los temas anteriores un programa está compuesto por una serie de instrucciones que se ejecutan una tras otra, desde la primera hasta la última. En este sentido, podemos considerar que implementa un algoritmo con una serie de pasos que se ejecutan de forma *secuencial*. Sin embargo, son muy pocos los algoritmos de este tipo, ya que incluso para problemas muy simples, son necesarias otras alternativas.

En C++ hay varias instrucciones que nos permiten especificar que la siguiente instrucción a ejecutar puede no ser la que se ha escrito a continuación. En los programas que hemos visto hasta ahora sólo ha sido necesario un *flujo secuencial*. Sin embargo, en otros programas es posible que después de una instrucción exista una *transferencia de control* a otra instrucción que no es la que se ha escrito a continuación.

Distinguimos tres estructuras de control:

1. La estructura *secuencial*. La instrucción a ejecutar es la siguiente en la secuencia.
2. La estructura *condicional*. Un conjunto de instrucciones se ejecutan si se cumple una condición.
3. La estructura *iterativa*. Un conjunto de instrucciones se ejecutan múltiples veces, incluyendo la posibilidad de que no se ejecuta ninguna vez.

En este capítulo nos centraremos en estudiar la segunda de ellas. El lenguaje C++ ofrece tres tipos de estructuras condicionales o de selección: *simple, doble* y *múltiple*, que veremos a lo largo de este tema.

La “idea” de la estructura condicional consiste en que un conjunto de instrucciones se ejecutan en caso de que se cumpla cierta condición, es decir, se evalúa una condición y en caso de que se cumpla (sea *verdadera*) se ejecutan dichas instrucciones. Por ejemplo, podemos considerar que el programa para resolver la ecuación de primer grado del tema anterior (página 15) no es correcto, ya que si el usuario introduce un valor de *a* incorrecto —el cero— no es posible realizar el cálculo para obtener la solución. Por lo tanto, parece más adecuado que el programa escriba el resultado

sólo si el valor es correcto, es decir, cuando la condición de que el valor es distinto de cero sea *verdadera*. El algoritmo podría ser algo así:

1. Leer los valores a, b.
2. Si a no vale 0:
   a) Calcular $x = -b/a$
   b) Escribir como resultado el valor x.

donde vemos que, después del segundo paso, podría ocurrir que el algoritmo no continuara con la "siguiente" instrucción de cálculo de x. Hemos cambiado el flujo secuencial para evitar una división que no es posible..

3.2 Operadores relacionales. Selección simple

Para usar la estructura de selección es necesario especificar *condiciones*, es decir, expresiones booleanas que se evalúan como *verdadero* o *falso*. El lector recordará que `bool` es un tipo predefinido para representar dos valores: `true` (verdadero) y `false` (falso). Por tanto, las condiciones que aparecen en las estructuras de control serán realmente expresiones que devuelven un valor de tipo `bool`.

Una condición se puede formar usando los operadores relacionales. Por ejemplo, considere *x,y* dos valores de un mismo tipo[1] —entero, real, etc.— podemos escribir las expresiones de la tabla 3.1. Note cómo el operador de igualdad —doble carácter— es distinto al de asignación.

Tabla 3.1
Operadores relacionales.

Operador	Expresión	Resultado
<	x<y	*true* si x es menor que y (*false* en otro caso)
>	x>y	*true* si x es mayor que y (*false* en otro caso)
>=	x>=y	*true* si x es mayor o igual que y (*false* en otro caso)
<=	x<=y	*true* si x es menor o igual que y (*false* en otro caso)
==	x==y	*true* si x,y son iguales (*false* en otro caso)
!=	x!=y	*true* si x,y son distintos (*false* en otro caso)

Podemos usar estos operadores para especificar la condición de la estructura condicional simple. Su sintaxis es la siguiente:

`if` (*condición*) *sentencia*

donde vemos que comenzamos con la palabra reservada `if`, seguida por una condición (que se evaluará como verdadero o falso) y una sentencia. Cuando el programa llega a la sentencia `if`, evalúa la condición y:

- si es `true`, ejecuta la sentencia y sigue con el programa
- si es `false`, ignora la sentencia y sigue con el programa.

Note, además, que la condición debe estar escrita entre paréntesis. Un ejemplo es el siguiente programa:

```
1 #include <iostream>
2 using namespace std;
3
```

[1]Realmente, no es válido cualquier tipo (como veremos), sin embargo, para todos los tipos básicos que veamos en estos primeros capítulos, las expresiones son correctas.

```
4 int main()
5 {
6    int dato;
7
8    cout << "Introduzca un valor entero distinto de cero" << endl;
9    cin >> dato;
10   if (dato!=0)
11      cout << "Es un alumno muy obediente" << endl;
12   if (dato==0)
13      cout << "¡Ups! qué desobediente..." << endl;
14 }
```

En este programa, la línea 8 pide un valor entero y la línea 9 se encarga de guardarlo —desde la entrada estándar— en la variable `dato`. A continuación hay dos sentencias condicionales:

- En las líneas 10-11 una sentencia condicional indica que si se cumple la condición `dato!=0`, ejecuta la sentencia de la línea 11.
- En las líneas 12-13 una sentencia condicional indica que si se cumple la condición `dato==0`, ejecuta la sentencia de la línea 11.

En la ejecución, después de la línea 9, el programa evalúa la primera condición obteniéndose un valor **`true`** o **`false`**, dependiendo de la entrada. En caso de que sea **`true`**, pasa a ejecutar la línea 11 (es la sentencia asociada a **`if`**), y si es **`false`**, la ignora. En cualquiera de los dos casos, una vez que termina con esa sentencia (líneas 10-11), continúa con la estructura secuencial habitual, es decir, con la línea 12. En esta línea, vuelve a repetirse el mismo funcionamiento.

Por tanto, note que hay dos posibles "caminos de ejecución" dependiendo del valor de `dato`:

1. Si vale distinto de cero. La línea 10 evalúa la expresión como **`true`**, se ejecuta la línea 11 (escribe el mensaje), la línea 12 se evalúa como **`false`** y se ignora la línea 13
2. Si vale cero. La línea 10 evalúa la expresión como **`false`**, se ignora la línea 11, la línea 12 se evalúa como **`true`** y se escribe el mensaje de la línea 13.

Ejercicio 3.1 Escribir un programa que lea dos valores reales y escriba en la salida estándar un mensaje indicando cuál es el mayor.

Ejercicio 3.2 Escribir un programa que lea un valor entero y escriba en la salida estándar un mensaje indicando si es par o impar.

Ahora bien, es posible que deseemos ejecutar más de una sentencia cuando la condición es verdadera. Para ello, usamos la sintaxis:

`if` (*condición*) *{ sentencias }*

donde "*sentencias*" corresponde a un número de cero o más sentencias. A esta estructura, en la que delimitamos un conjunto de sentencias entre llaves, la denominamos *bloque de sentencias* o *sentencia compuesta*. Tenga en cuenta que, en este conjunto de sentencias, también es posible volver a usar **`if`**.

Por ejemplo, imagine que deseamos escribir dos mensajes en dos sentencias distintas. El programa anterior se podría escribir de la siguiente forma:

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6    int dato;
7
8    cout << "Introduzca un valor entero distinto de cero" << endl;
9    cin >> dato;
```

```
10     if (dato!=0) {
11        cout << "Ha escrito " << dato
12             << " que es distinto de cero" << endl;
13        cout << "Es un alumno muy obediente" << endl;
14     }
15     if (dato==0)
16        cout << "¡Ups! qué desobediente..." << endl;
17  }
```

donde hemos especificado que si la condición de la línea 10 es **true**, se ejecutará el bloque de sentencias de las líneas 11-13.

Es muy importante observar que se ha escrito el programa con un formato que facilita la lectura. Las líneas 11-13 se han sangrado para que el lector note claramente que corresponden al bloque de la sentencia **if**. Recuerde que el lenguaje C++ es de formato libre y, por tanto, este sangrado no es más que un aspecto estético para la comodidad del lector. Si no existe, el compilador generará la misma salida.

Ejercicio 3.3 Considere el ejemplo anterior eliminando las dos llaves de la sentencia **if**. Si el usuario ejecuta el programa con el dato *0*, ¿qué salida obtendrá?

Ejercicio 3.4 Escriba un programa que lea un número real, correspondiente al radio de un círculo. Como resultado, escribirá el radio introducido, el área del círculo, y la longitud de su perímetro. Además, comprobará si el radio no es positivo, en cuyo caso se obtendrá un mensaje informando sobre ello.

3.2.1 Operadores aritméticos y relacionales

Los operadores relacionales permiten escribir una expresión booleana que representa el resultado de comparar dos valores. Éstos pueden ser el resultado de la evaluación de una expresión, y no solamente una variable o un literal como en los ejemplos anteriores.

Consideremos otro ejemplo en el que un programa pide la entrada de dos notas —teoría y prácticas— de un alumno, y escribe en la salida estándar si está aprobado o no teniendo en cuenta que la nota final es la media de las dos anteriores. Una solución podría ser[2]:

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

int main()
{
   double teoria, practica; // Los acentos no son válidos

   cout << "Introduzca las notas de teoría y prácticas:" << endl;
   cin >> teoria >> practica;
   if ( (teoria+practica)/2 >= 5 )
      cout << "Aprobado." << endl;
   if ( (teoria+practica)/2 < 5 )
      cout << "Suspenso." << endl;
}
```

donde la condición es un poco más compleja, ya que la expresión también contiene operadores aritméticos. En principio, para saber la forma en que se evalúan estas expresiones sólo necesitamos saber que los operadores aritméticos son prioritarios. Por tanto, los últimos operadores que se evalúan son los relacionales. En este caso, primero se obtiene la media entre las dos notas —note que ha sido necesario añadir unos paréntesis, ya que la división tiene mayor precedencia que la suma)— y el resultado se compara con el valor *5*.

[2]Es probable que piense en una solución declarando una tercera variable para almacenar la nota final. Por conveniencia, en este ejemplo repetimos la expresión de cálculo.

Por otra parte, el lector puede detenerse un momento y pensar en los tipos de los datos que se utilizan en las expresiones. Se suman dos valores reales (**double**), se dividen por un tipo entero (**int**), y el resultado (que es de tipo **double**), se compara con un valor entero (**int**).

Al igual que en el tema anterior, insistimos en que usar valores enteros y reales "mezclados" en expresiones no debe preocuparnos, al menos en principio, ya que se realizan conversiones automáticamente para que puedan evaluarse. En nuestro caso, cuando aparecen dos tipos de datos **int** y **double**, relacionados con un operador binario, el tipo entero se convierte automáticamente a **double**.

> **Ejercicio 3.5** Escribir un programa que lea tres números reales que corresponden a una ecuación de segundo grado. En caso de que tenga solución real, la escribirá en la salida estándar. Compruebe para ello si el discriminante es no negativo.

3.3 Selección doble *if/else*

La sintaxis de esta estructura es:

> **if** (*condición*) *sentencia1* **else** *sentencia2*

Teniendo en cuenta que cuando indicamos "*sentencia*" —1 o 2— también puede hacer referencia a una sentencia compuesta o bloque de sentencias entre llaves.

En este caso hemos añadido a la selección simple una nueva palabra reservada —**else**— seguida por una sentencia o bloque de sentencias. Respecto al funcionamiento, primero se evalúa la condición, lo que da lugar a dos posibles casos:

- Si es **true**, se ejecuta la sentencia o bloque de sentencias 1.
- Si es **false**, se ejecuta la sentencia o bloque de sentencias 2.

Por tanto, esta estructura nos permite escoger —por medio de una condición— entre dos posibles caminos independientes en el flujo del programa. Lógicamente, una vez que se ejecuta la parte seleccionada —**if** o **else**— el programa continúa con la sentencia que hay después de todo el bloque **if−else**.

Por ejemplo, volvamos al programa anterior en el que se escribe si una nota corresponde a *"Aprobado"* o *"Suspenso"*. Podemos simplificarlo de la siguiente forma:

```
1  cout << "Introduzca las notas de teoría y prácticas" << endl;
2  cin >> teoria >> practica;
3  if ( (teoria+practica)/2 >= 5 )
4     cout << "Aprobado." << endl;
5  else
6     cout << "Suspenso." << endl;
```

en el que la condición en la línea 3 escoge entre escribir el primer mensaje (si es **true**) o el segundo (si es **false**).

> **Ejercicio 3.6** Escriba un programa que lea tres números reales correspondientes a una ecuación de segundo grado y escriba en la salida estándar las soluciones —si las hay— o un mensaje indicando que no las hay. Compruebe para ello si el discriminante es no negativo, usando una sentencia **if**/**else** para distinguir los dos casos.

3.3.1 Anidamiento

Las sentencias **if** o **if−else** pueden incluir otras estructuras condicionales. En ese caso hablamos de anidamiento. Como ejemplo, el siguiente código corresponde a un programa que lee dos números e indica cuál es el mayor.

```
1 int main()
2 {
3    double n1, n2;
4
5    cout << "Introduzca dos números" << endl;
6    cin >> n1 >> n2;
7    if (n1>n2)
8       cout << "El primero es mayor." << endl;
9    else if (n1<n2)
10           cout << "El segundo es mayor." << endl;
11         else cout << "Son iguales." << endl;
12 }
```

Hemos incluido una sentencia `if-else` —línea 7— para comprobar si el primero es mayor. En caso de que sea falso, hemos distinguido dos posibilidades: que sea menor o que sea igual. Para ello, hemos anidado otra sentencia en la parte `else`. Observe que no hemos usado ninguna llave, ya que no son necesarias porque en la parte `else` sólo hemos incluido una sentencia (en este caso, una sentencia `if-else`).

Ejercicio 3.7 Escriba un programa que lea tres números reales, calcule el máximo, y finalmente lo escriba en la salida estándar.

Ejercicio 3.8 Escriba un programa que lea tres números reales correspondientes a una ecuación de segundo grado y escriba en la salida estándar las soluciones. Para ello, debe comprobar que el valor del coeficiente de grado dos es distinto de cero y que el discriminante es no negativo. En estos casos, en lugar de las soluciones, escribirá el mensaje correspondiente en la salida estándar.

Ejercicio 3.9 Escriba un programa que lea un número real correspondiente a una nota, y escriba en la salida estándar el mensaje “Suspenso” (menor que 5), “Aprobado” (de 5 a 7), “Notable” (de 7 a 9) o “Sobresaliente” (9 o más) dependiendo del valor de la nota.

Emparejando if y else

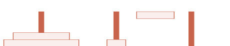
En el siguiente ejemplo hemos fijado el valor de las condiciones, ¿cuál es el resultado?

```
1 int main()
2 {
3    bool uno, dos;
4    uno= false;
5    dos= false;
6    if (uno)
7       if (dos)
8          cout << "Primer bloque" << endl;
9       else cout << "Segundo bloque" << endl;
10 }
```

El resultado es que no escribe nada, ya que el primer `if` es falso, y por tanto se ignoran el resto de líneas (7-9) que sólo se ejecutarían si la condición hubiera sido `true`. En este ejemplo, el primer `if` corresponde a una selección simple, ya que el bloque `else` corresponde al último `if` en el anidamiento.

Si queremos hacer que la parte `else` corresponda al primer `if`, lo podemos hacer añadiendo las llaves que determinan un bloque de sentencias de la siguiente forma:

```
if (uno) {
   if (dos)
      cout << "Primer bloque" << endl;
}
else cout << "Segundo bloque" << endl;
```



Es decir, cuando el compilador encuentra la parte **else**, la asigna al último **if** al que pueda corresponder. Observe que en este último ejemplo, con las llaves añadidas, sólo puede asignarse al primer **if**. En algunas referencias puede encontrar algún consejo relacionado con un estilo de escritura que añade las llaves incluso si hay una única sentencia. En la práctica, si escribe el programa con cuidado, no necesitará estas llaves adicionales.

3.3.2 Formato de escritura

El formato de escritura de la sentencia **if** es "libre". Recordemos que el lenguaje C++ es de formato libre, y no importa que usemos como separadores espacios, saltos de línea o tabuladores.

Sin embargo, es habitual escoger un formato que facilite la lectura. De esta forma será mucho más fácil el mantenimiento del programa, por ejemplo cuando tengamos que revisarlo para realizar una modificación. Además, es posible que sean varios los programadores que trabajen con la implementación. Por tanto, aunque el formato es libre, consideraremos "obligatorio" usar algunas reglas o estilo de escritura válido.

Considere el siguiente trozo de código:

```
1 if (dato!=0)
2    cout << "Ha escrito " << dato << " distinto de cero" << endl;
3    cout << "Es un alumno muy obediente" << endl;
```

donde la sentencia **if** se ha escrito sin usar llaves. Note que aunque el sangrado puede engañarnos, para el compilador la única sentencia que se ejecuta en caso de que la condición sea **true** es la línea 2. Es decir, la tercera línea se escribirá independientemente del resultado de la expresión booleana.

Otro ejemplo que puede llevar a confusión si no se escribe correctamente es:

```
if (uno)
   if (dos)
      cout << "Primer bloque" << endl;
else cout << "Segundo bloque" << endl;
```

El formato es realmente malo, porque no sólo hemos sangrado incorrectamente, sino que el lector puede confundir la asignación de la parte **else** con el primer **if**.

Es importante que no olvide esta relación entre un **else** y los **if** anteriores, porque cambiar el sangrado no modifica el comportamiento. Seguro que en el futuro más de una vez dedicará mucho tiempo a analizar un código que contiene un error como el anterior y le parece perfecto.

3.3.3 El operador ternario "?:"

En algunos casos, especialmente cuando el código es complejo y existen muchos caminos en el flujo del programa, resulta muy cómodo reemplazar algunas sentencias **if**/**else** por expresiones en base al operador ternario "?:". La sintaxis de este operador es:

condición **?** *expresión1* **:** *expresión2*

de forma que primero se evalúa la condición; en caso de que sea cierta, el resultado es la expresión 1; si no, el resultado es la expresión 2. Podríamos leerlo como *"Si condición, entonces expresión 1 y si no expresión 2"*. Por ejemplo, si tenemos un trozo de código como el siguiente:

```
if (condicion)
   variable= expresion1;
else variable= expresion2;
```

podemos sustituirlo por:

```
variable= condicion? expresion1 : expresion2;
```

Note que no son necesarios paréntesis, pues la asignación tiene menos prioridad que el operador "`?:`". Un ejemplo de su uso es un programa que escriba el mayor de dos números. El bloque **`main`** sería:

```
int main()
{
   double a,b;
   cin >> a >> b;
   cout << "Mayor: " << (a>b ? a : b) << endl;
}
```

En este caso sí son necesarios los paréntesis ya que, como veremos, el operador << es de mayor prioridad.

3.4 Expresiones booleanas. Operadores lógicos

En las instrucciones condicionales usamos una expresión que toma el valor **`true`** o **`false`** para escoger entre dos posible caminos de ejecución. Hasta ahora hemos usado *condiciones simples*, es decir, una expresión con un único operador relacional. C++ ofrece los operadores lógicos para operar con valores booleanos, de forma que es posible escribir expresiones más complejas combinando condiciones simples.

En la tabla 3.2 se presenta una descripción de los tres operadores lógicos que C++ (y C) ofrece[3]:

Tabla 3.2
Operadores lógicos.

Operando 1	Operando 2	Operador	Resultado
false	false	&&	false
false	true	&&	false
true	false	&&	false
true	true	&&	true
false	false	||	false
false	true	||	true
true	false	||	true
true	true	||	true
false	-	!	true
true	-	!	false

Por ejemplo, podemos comprobar que un valor real *v* corresponde a una calificación, es decir, es un valor entre 0 y 10. Para ello podemos usar el operador **`and`** (`&&`) para escribir una condición múltiple que es **`true`** si dos expresiones son **`true`**. En concreto, podemos escribir:

```
if (v>=0 && v<=10)
   cout << "El valor es correcto" << endl;
else cout << "El valor no es correcto." << endl;
```

donde la expresión indica que si *v* es mayor o igual que cero **Y** a la vez menor o igual que `10`, es que es un valor correcto. Por otro lado, podemos usar el operador **`or`** (`||`) para escribir una condición múltiple que es **`true`** si alguno de los dos o los dos operandos es **`true`**. El ejemplo anterior se puede reescribir:

[3]En C++ también es posible usar las palabras reservadas **`and`**, **`or`** y **`not`** para los tres operadores, aunque nosotros seguiremos usando la notación tradicional de C.

```
if (v<0 || v>10)
   cout << "El valor no es correcto" << endl;
else cout << "El valor es correcto." << endl;
```

donde hemos cambiado la condición, por lo que hemos tenido que intercambiar los mensajes a la salida estándar. La expresión indica que si `v` es menor que cero **O** es mayor que `10`, el valor no es correcto.

Note que podríamos haber cambiado la expresión al contrario, utilizando solamente el operador **not** (`!`), de la siguiente forma:

```
if ( ! (v>=0 && v<= 10) )
   cout << "El valor no es correcto" << endl;
else cout << "El valor es correcto." << endl;
```

Como ejemplo, escribimos un programa que lee dos números enteros positivos y escribe si uno es múltiplo del otro:

```
1  int main()
2  {
3     int n1, n2;
4
5     cout << "Introduce dos enteros positivos: ";
6     cin >> n1 >> n2;
7     if (n1>0 && n2>0)
8        if (n1%n2 == 0  ||  n2%n1 == 0)
9           cout << "Si, uno es múltiplo del otro" << endl;
10       else cout << "No, ninguno es múltiplo del otro" << endl;
11    else cerr << "Entrada incorrecta." << endl;
12 }
```

donde es de destacar —en la línea 8— una expresión que contiene los operadores *módulo* (`%`) y *o lógico* (`||`). Es decir, las expresiones booleanas ahora pueden ser más complejas, ya que disponemos de más operadores —aritméticos, relacionales y lógicos— y por tanto es necesario conocer la forma en que el compilador las evalúa. En la tabla 3.3 se presenta un resumen de los operadores vistos hasta el momento. Los situados en el mismo recuadro tienen la misma precedencia.

Es importante no olvidar que la condición de la instrucción de selección no deja de ser una expresión booleana. De hecho, podemos usar una variable de tipo **`bool`** para almacenar el resultado de evaluar la expresión, para luego usarla como condición de la siguiente forma:

```
1 bool nota;
2
3 nota= v>=0 && v<=10;
4 if (nota)
5    cout << "El valor es correcto" << endl;
6 else cout << "El valor no es correcto." << endl;
```

En la línea 3 asignamos el valor de la expresión booleana a la variable `nota`. Recordemos que la asignación no es más que un operador. En este caso, podemos ver que la precedencia del operador de asignación es la menor. En primer lugar se evalúan los operadores relacionales, después el operador lógico **`and`** y finalmente se asigna el resultado.

A pesar de que la precedencia haga innecesarios los paréntesis, en expresiones complejas puede ser recomendable incluirlos para facilitar la lectura.

Ejercicio 3.10 Escribir las expresiones booleanas que corresponden a las siguientes condiciones:

1. El valor del entero A está en $\{1,2,3,4,5,6\}$
2. El valor del real Aestá en $[-\infty,-5]\cup[5,\infty]$
3. El valor del real A no está en $[-5,3]$
4. El valor del real A está en $\{1,2,3\}\cup[10,20[$
5. El valor del entero A es par o un valor impar entre cero y diez.

Tabla 3.3
Precedencia de operadores aritméticos, relacionales y lógicos .

!	No	!*expr*
+	Más unario	+*expr*
-	Menos unario	-*expr*
*	Multiplicación	*expr***expr*
/	División	*expr*/*expr*
%	Módulo	*expr*%*expr*
+	Suma	*expr*+*expr*
-	Resta	*expr*-*expr*
<	Menor	*expr*<*expr*
<=	Menor o igual	*expr*<=*expr*
>	Mayor	*expr*>*expr*
>=	Mayor o igual	*expr*>=*expr*
==	Igual	*expr*==*expr*
!=	No igual	*expr*!=*expr*
&&	Y lógico	*expr*&&*expr*
||	O lógico	*expr*||*expr*
?:	expresión condicional	*expr*?*expr*:*expr*
=	Asignación	*valor_i*=*expr*

6. El valor del real A está en $[0,10] \cup [20,30]$
7. El valor del real A no está en $[0,10[\cup]20,30]$

3.4.1 Evaluación en corto

En general, dada cualquier expresión, el orden de evaluación de las subexpresiones que contenga no está definido. Sin embargo, en los casos particulares de los operadores `&&` y `||`, se garantiza que el operando izquierdo se evalúa antes que el derecho.

De esta forma, el segundo operando de `&&` se evalúa sólo si el primer operando es `true`, y el segundo operando de `||` si el primero es `false`. Este tipo de evaluación se llama evaluación en ciclo corto o en corto-circuito[4].

Es probable que su primera reacción después de esta afirmación sea preguntarse: ¿Para qué especificar ese orden de evaluación? Efectivamente, en principio parece que evaluar subexpresiones en un orden u otro no debería ser relevante. Por eso mismo el lenguaje no se molesta en indicar cómo se realiza. Sin embargo, en el caso de los operadores `&&` y `||` puede resultar fundamental para garantizar que el código es correcto.

Por ejemplo, imagine que deseamos escribir un programa que nos indique si un número es positivo y divisor de 100. Podríamos escribir el siguiente código:

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

int main()
{
   int dato;
```

[4] Del inglés short-circuit evaluation.

```
        cout << "Introduzca un dato positivo y divisor de 100: ";
        cin >> dato;

        if (100%dato==0 && dato>0)    // Número positivo divisor de 100
           cout << "El número es correcto" << endl;
        else cout << "El número no es correcto" << endl;
    }
```

donde podemos ver que la condición que hemos incluido es compuesta, en primer lugar evaluamos si es divisor de 100 y en segundo lugar si es positivo. El problema de esa solución es que el programa fallaría en tiempo de ejecución cuando introducimos el valor cero. El razón es que no es posible operar con el cero: no podemos calcular el módulo.

Para resolver el problema podríamos escribir dos sentencias `if`: primero para confirmar que el valor es mayor que cero y una vez que es correcto, una segunda para calcular el módulo. Sin embargo, una solución más sencilla es cambiar el orden de las condiciones de la siguiente forma:

```
    // ...
        cin >> dato;
        if (dato>0 && 100%dato==0)   // Número positivo divisor de 100
           cout << "El número es correcto" << endl;
        else cout << "El número no es correcto" << endl;
    }
```

donde primero comprobamos que el dato es mayor que cero y luego si es divisor. Gracias a la evaluación en corto, si el dato fuera cero no sería necesario evaluar la segunda parte, por lo que no se generaría ningún error en ejecución. Si el valor es cero, la primera parte ya garantiza `false` y directamente se ejecuta el código de la parte `else` sin evaluar la segunda subexpresión.

3.5 Selección múltiple *switch*

Las estructuras de selección anteriores nos permitían escoger entre dos caminos, dependiendo de si una expresión booleana era `true` o `false`. Sin embargo, en algunos algoritmos es necesario escoger un camino de entre muchos, dependiendo del valor de una expresión.

La estructura de selección múltiple se compone de una serie de etiquetas `case` (y opcionalmente una etiqueta `default`) que permiten escoger entre distintas alternativas. El formato de escritura de esta estructura es el siguiente:

```
switch (expresión) {
      case valor1:   sentencias1;   break;
      case valor2:   sentencias2;   break;
         ...
      case valorN:   sentenciasN;   break;
      [default:   sentencias ]
}
```

El funcionamiento de esta estructura consiste en evaluar la expresión[5] e ir comparándola con cada uno de los distintos casos, desde el `1` hasta el *N*. Cuando se encuentra uno igual, se sigue la ejecución con las sentencias asociadas hasta que se encuentra la sentencia `break`.

En caso de que ningún caso coincida, la sentencia `switch` pasa a ejecutar las sentencias de la parte `default`, si se ha especificado.

Un ejemplo de su uso es el siguiente:

[5] La sentencia `switch` es válida para tipos llamados “integrales” como `char`, `int` o `bool`.

```
1  char letra;
2
3  cout << "Haga una selección (s/n)" << endl;
4  cin >> letra;
5  switch (letra) {
6     case 's':
7     case 'S':
8        cout << "Escogió \"Sí\"" << endl;
9        break;
10    case 'n':
11    case 'N':
12       cout << "Escogió \"No\"" << endl;
13       break;
14    default:
15       cout << "No escogió correctamente" << endl;
16 }
```

Es interesante que el lector observe con detenimiento este ejemplo para darse cuenta de que:

- Son distintos los caracteres en minúscula y mayúscula.
- Se han incluido los caracteres de comillas dobles en dos mensajes (líneas 8 y 12). Para no confundirlas con el final de cadena, ha sido necesario precederlas con la barra invertida.
- Después de cada etiqueta **case** debe aparecer una expresión constante.
- Se puede ver que el conjunto de sentencias de una etiqueta **case** puede ser vacío, así como que la sentencia **break** es opcional. Realmente, cada etiqueta indica un punto donde seguir ejecutando hasta que se encuentre con una sentencia **break** que indica fin del **switch**. Un error típico es olvidar un **break**, y que el programa siga ejecutando las líneas de la siguiente etiqueta.
- Si apareciera repetido algún valor en dos etiquetas **case**, se escogería el primero ignorando el segundo.

La estructura de selección múltiple se puede simular con las anteriores, especificando las distintas etiquetas **case** con varias instrucciones **if/else** anidadas. Por ejemplo, en el caso anterior podemos escribir:

```
if (letra=='s' || letra=='S')
   cout << "Escogió \"Sí\"" << endl;
else if (letra=='n' || letra=='N')
   cout << "Escogió \"No\"" << endl;
else // default
   cout << "No escogió correctamente" << endl;
```

Ejercicio 3.11 Reescriba el formato general de la sentencia **switch** con la misma notación, pero haciendo uso de sentencias **if/else**.

Ejercicio 3.12 Escriba un programa que, después de pedir dos números, presente al usuario un conjunto de 4 opciones (suma, resta, producto, división). Una vez que el usuario seleccione la deseada, el programa presentará el resultado de la operación correspondiente.

3.6 Booleanos en C. Algunos detalles

En el lenguaje C no existe el tipo de dato **bool**. Para poder manejar expresiones booleanas se utiliza el tipo de dato entero. Para ello, se establece que el entero cero indica **false** y un valor distinto de cero **true**.

El lenguaje C++ utiliza el tipo **bool**, pero sigue conteniendo las capacidades del lenguaje C. Por lo tanto, también puede manejar los valores enteros del mismo modo que C. Por ejemplo, el siguiente programa es válido:

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

int main()
{
   int valor;

   cin >> valor;
   if (valor)
      cout << "Distinto de cero" << endl;
   else cout << "Cero" << endl;
}
```

Además, el lenguaje permite que el compilador haga conversiones implícitas de tipos enteros y de coma flotante a booleanos. Es decir, si el compilador espera un valor booleano y le damos un tipo de dato entero o en coma flotante[6], puede realizar una conversión considerando que con el cero entiende `false` y con cualquier otro valor `true`.

Esto puede provocar errores inesperados. Por ejemplo, imaginemos que nos olvidamos de escribir los dos signos `'='` en la expresión condicional del siguiente ejemplo:

```
int opcion;

cin >> opcion;
if ( opcion = 5 )
   cout << "Ha escogido el 5" << endl;
```

En la penúltima línea sólo hemos escrito un signo igual. Podemos pensar que el compilador detectaría un error sintáctico. Sin embargo, es un programa válido. Dentro del paréntesis hay una expresión con dos operandos y un operador. Evaluar la expresión implica que la variable `opcion` pasa a valer 5. Como resultado, la asignación devuelve la variable `opcion`[7], es decir, un entero, que como vale distinto de cero se evalúa como verdadero. Por tanto, en este ejemplo, la condición siempre es verdadera. Afortunadamente, la mayoría de los compiladores que se encuentran con esta asignación avisan por si se ha cometido un error.

Pensemos en un ejemplo un poco más extraño. Tal vez difícil de cometer por un programador experimentado, pero interesante para alguien sin experiencia. Pensemos que hacemos lo siguiente:

```
int opcion;
cin >> opcion;
if (0<opcion<=3) cout << "¡Escogió una opción!" << endl;
```

Esto provoca que se evalúe primero el operador más a la izquierda (el operador $<$) devolviendo `true` o `false` dependiendo del valor de `opcion`, para después comparar dicho valor booleano con el 3. Como los valores booleanos se pueden convertir a enteros —recuerde la relación tan estrecha que indicábamos— el compilador lo convierte a cero (si es `false`) o uno (si es `true`).

Por tanto, este programa también sería válido. El resultado de la condición sería siempre `true` independientemente del valor que hayamos dado a la variable `opcion`.

Como puede observar, a pesar de disponer del tipo `bool` y de que podamos obviar el uso de los enteros como valores booleanos, es importante que un programador de C++ sepa manejar correctamente el tipo entero en este contexto. Más adelante, con más experiencia y habiendo leído código escrito por otros programadores, se habituará a entender el uso de otro tipos para indicar verdadero (si son distinto de cero) y falso (si son cero).

[6] Más adelante veremos que también para punteros.

[7] Como puede ver, el operador de asignación se evalúa asignando el valor de la expresión a la variable de la izquierda y devolviendo como resultado esa variable.

3.7 Representación gráfica

Al aparecer algoritmos con distintos caminos en el flujo del programa, resulta útil usar representaciones gráficas de los pasos que se realizan enfatizando las distintas alternativas que se presentan en el flujo.

Por ejemplo, en la figura 3.1 se presenta un *diagrama de flujo*. Se pueden distinguir distintos signos gráficos para indicar principio y fin del algoritmo, entrada o salida de datos, o una condición para seleccionar un camino.

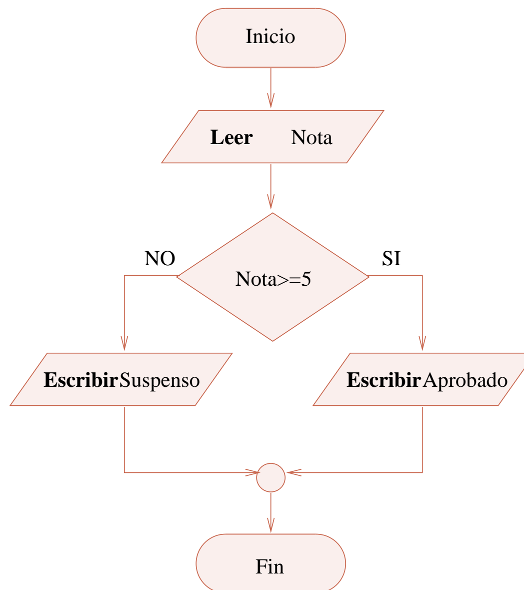


Figura 3.1
Un organigrama o diagrama de flujo.

3.8 Problemas

Problema 3.1 Escriba las condiciones para que un valor V de tipo `double` esté en los siguientes conjuntos:
- $[0,1]$
- $[-\infty,-5] \cup [5,\infty]$
- $\{1.0, 2.0, 3.0\} \cup [5.0,\infty]$

Modifique las condiciones para que indiquen que no está.

Problema 3.2 Identifique los errores de los siguientes trozos de código:

```
1 cin >> opcion;
2 if opcion>=1 && opcion<=5
3    cout << "La opción es correcta (1-5)" << endl;
4 else if opcion=6
5          cout << "Escogió salir... <<endl,
```

```
1 if (a>0)
2    cout << 'Positivo';
3 else if (a>10);
4          cout << "Mayor de 10";
5       else if (a<0)
6              cout << "Negativo"
```

```
switch (opcion) {
    case 1: cout << "opción 1";
    case 2: cout << "opción 2";
    case 3: cout << "opción 3";
    default: cout << "Sin opción";
}
```

Problema 3.3 La posición de un círculo se puede determinar con 3 valores reales, los dos primeros corresponden al centro y el tercero al radio. Escriba un programa que lea el centro (dos coordenadas) y radio de un círculo. Una vez leída la posición del círculo, preguntará al usuario por un punto cualquiera (dos coordenadas) e informará sobre si dicho punto está dentro o no del círculo. Recuerde que un punto está dentro de un círculo si la distancia al centro es menor que el radio.

Problema 3.4 Escribir un programa que lea tres números enteros en tres variables `n1`, `n2` y `n3`, reasigne los valores de las variables para que `n1` sea la más pequeña y `n3` la más grande. Finalmente, el programa escribirá los tres valores ordenados.

Problema 3.5 Escriba un programa que lea día, mes y año, y escriba si la fecha es correcta. Considere que la fecha está en el rango 1950-2050.

Problema 3.6 Realice un programa que fije el valor de un punto en un tablero de 10x10. El programa deberá preguntar tres veces por una posible posición en el tablero al usuario. Después de cada intento, el programa indicará si se ha acertado (en este caso felicitará al usuario y terminará). Si no ha acertado, indicará la distancia al objetivo (*Nota: utilice constantes para fijar el valor del objetivo en el tablero*).

Problema 3.7 Escriba un programa que lea las longitudes de los lados de un triángulo y escriba en la salida estándar:

1. Si es un triángulo. Para ello, la suma de las longitudes de dos lados cualesquiera debe ser mayor que la tercera.
2. En caso de que sea triángulo:
    a) Si es equilátero (las longitudes son iguales).
    b) Si es escaleno (si las longitudes son distintas).
    c) Si es isósceles (sólo dos lados iguales).

Además, el programa deberá indicar un error en caso de que alguna de las longitudes sea no positiva.

Problema 3.8 Considere un programa para jugar al ajedrez. Suponga que se desea realizar un movimiento de una pieza desde la posición (f_1, c_1) a (f_2, c_2). Escriba las expresiones booleanas que corresponden a:

1. El movimiento está dentro del tablero. Es decir, los valores de la fila y columna para las dos posiciones están entre 1 y 8.
2. Es un movimiento válido para la torre.
3. Es un movimiento válido para la reina.

Problema 3.9 La posición de una circunferencia se puede determinar con 3 valores reales, los dos primeros corresponden al centro y el tercero al radio. Escriba un programa que lea la posición de dos circunferencias y escriba información sobre su relación. Tenga en cuenta las siguientes posibilidades:

1. Si son la misma circunferencia.
2. Si una de ellas está dentro de la otra.
3. El número de puntos de corte entre las circunferencias.

Problema 3.10 La localización de un rectángulo se puede determinar con 4 valores que corresponden a los puntos de la esquina superior izquierda y esquina inferior derecha. Escriba un programa

que lea la posición de un rectángulo y de un punto cualquiera. Como resultado, escriba en la salida estándar si el punto se encuentra dentro del rectángulo.

4 La estructura de repetición

Introducción 59
Bucles *while* 59
Bucles *for* 63
Bucles *do/while* 71
Estructuras de control anidadas 72
Declaración y ámbito de una variable 75
Programación estructurada y sentencia *goto* 76
Problemas 77

4.1 Introducción

Muchos problemas se resuelven con algoritmos basados en la *repetición* de un conjunto de acciones. Por ejemplo:

- Un algoritmo puede necesitar un dato de entrada que se pide al usuario con ciertas restricciones. Podríamos pedir el dato y, si no es correcto, volver a pedirlo hasta que lo sea. Es decir, la petición se puede repetir indefinidamente *mientras* que no sea correcto el dato. En este caso, el final de la iteración *se controla por una condición*.
- Un algoritmo puede calcular la media de 100 valores reales. Para ello, tiene que pedir 100 valores e ir acumulando la suma. Es decir, la petición se debe repetir un número determinado de veces (hay que pedir un valor 100 veces). En este caso, vemos que el final de la iteración *se controla por un contador*.

El lenguaje C++ ofrece distintos tipos de sentencias de iteración para poder hacer que se repitan un conjunto de instrucciones. A estas estructuras las denominamos *bucles* (también *ciclos o lazos*). Un bucle nos permite repetir cero o más veces un *conjunto de instrucciones* que denominamos *cuerpo del bucle*.

4.2 Bucles *while*

Un bucle de este tipo tiene el siguiente formato:

while (*condición*) *sentencia*

donde podemos ver que:

- Comienza con la palabra reservada `while`.
- Sigue con una condición entre paréntesis. Esta condición indica si se repite la ejecución de la sentencia o no. Es decir, será la condición que determine si el bucle sigue iterando.
- Finaliza con una sentencia que constituye el cuerpo del bucle.

El funcionamiento se representa gráficamente en la figura 4.1. Podemos ver que consiste en repetir la evaluación de la condición y la ejecución del cuerpo hasta que la condición sea `false`[1].

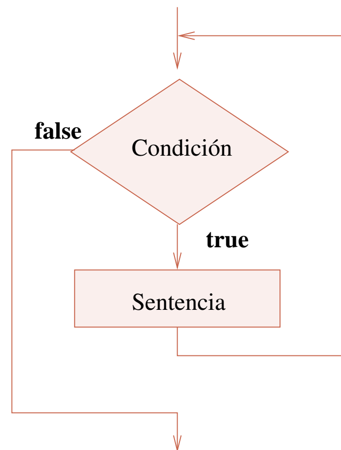


Figura 4.1
Funcionamiento del bucle *while*.

Note que es posible que el cuerpo del bucle no se ejecute ni una sola vez. Es el caso de que la condición sea `false` cuando se llega al bucle.

Por otro lado y como era de esperar, si el cuerpo del bucle lo compone más de una sentencia, debemos encerrarlas entre llaves para crear una sentencia compuesta o bloque de sentencias. En este caso, el formato se puede escribir como:

while (*condición*) { *sentencias* }

donde se pueden incluir cero o más sentencias como cuerpo del bucle.

Es importante darse cuenta de cómo funciona la condición para la iteración en estos bucles. Podemos decir que los bucles son del tipo *mientras* —como su nombre indica— en contraposición a los bucles del tipo *hasta*. Los primeros se formulan con una condición que indica si se sigue iterando, mientras que los segundos lo hacen con una condición que indica cuándo se termina. En C++ todos los bucles son del tipo *mientras*, tanto el que hemos visto como los que veremos más adelante.

Aunque le parezca un comentario "gratuito", pues está claro que el nombre de bucle indica que es del tipo *mientras*, es un error bastante común en los programadores sin experiencia. La escritura de condiciones para controlar los bucles es más intuitiva de lo que a primera vista parece, ya que tenemos la semántica de las palabras "mientras" y "hasta" muy asimilada. No se confunda y aunque el enunciado del problema hable de la condición de parada (indicando un "hasta..."), siempre deberá escribir la condición para seguir con la iteración (pensando en un *"mientras..."*).

Para estudiar el funcionamiento de los bucles `while` presentamos una serie de ejemplos. Estudie detenidamente cada uno de ellos, comprobando tanto la iteración del bucle cuando la condición es `true`, como la finalización cuando es `false`.

[1]También es posible terminar el bucle de otras formas, aunque nosotros las evitaremos para fomentar la programación estructurada (véase página 76).

Ejemplo 4.2.1 Escriba un programa que lea números enteros desde la entrada estándar hasta que se introduzca el cero. Finalmente, deberá escribir el resultado de la suma acumulada de todos ellos.

La solución es utilizar un bucle que itera hasta que se encuentra con el valor cero. Recuerdo que no es un bucle del tipo *hasta*, sino del tipo *mientras*, por lo tanto hay que indicar la condición para seguir iterando, es decir, la condición es que el valor leído sea distinto de cero. En cada iteración se sumará y acumulará un valor. Una solución es:

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

   suma= 0;
   cin >> valor;
   while (valor!=0) {
      suma= suma+valor;
      cin >> valor;
   }
   cout << "La suma total es:" << suma << endl;
}
```

A este valor "especial" que indica la finalización de la entrada también se le denomina *centinela*. En este ejemplo, el final de la entrada está controlado por centinela, un valor que no es válido para la entrada y con el sentido especial de *FIN*.

Fin ejemplo 4.2.1 ■

Ejemplo 4.2.2 Escriba un programa que lea 10 valores reales y escriba su suma en la salida estándar.

Para resolverlo, es necesario repetir 10 veces la lectura de un valor, así como la acumulación de éste en una variable que guarde el resultado final. La solución puede ser:

```
#include <iostream>
using namespace std;
int main()
{
   double suma, valor;
   int i;

   cout << "Introduce 10 valores: " << endl;

   suma= 0;
   i= 1;
   while (i<=10) {
      cin >> valor;
      suma= suma+valor;
      i= i+1;
   }

   cout << "La suma total es:" << suma << endl;
}
```

donde vemos que el cuerpo del bucle consiste precisamente en la lectura y acumulación de ese valor. Note cómo se han incorporado instrucciones, basadas en un contador `i`, para controlar el número de veces que itera el bucle. Concretamente:

- Damos un valor inicial antes del bucle.
- Comprobamos que se itera mientras estemos en el rango correcto.
- Incrementamos el contador en cada iteración.

Fin ejemplo 4.2.2 ■

Ejemplo 4.2.3 Se desea calcular el máximo común divisor (mcd) de dos números enteros. Para ello, utilizamos el algoritmo de Euclides que calcula el mcd entre dos números enteros a,b como sigue:

$$mcd(a,b) = \begin{cases} b & \text{si } b \text{ divide } a \\ mcd(b\ ,\ a \mod b) & \text{en caso contrario} \end{cases} \quad (4.1)$$

Por lo que para buscar el mcd de dos números tenemos que:

1. Obtenemos el resto de dividir `a` entre `b`.
2. Si el resto es cero, terminamos con `b` como solución.
3. Si el resto no es cero, volvemos a repetir el proceso. Ahora `b` tiene el papel de `a`, y el resto el de `b`.

El programa C++ puede ser el siguiente:

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;

    cout << "Introduzca dos números enteros positivos: ";
    cin >> a >> b;

    int resto= a%b;
    while (resto!=0) {
        a= b;
        b= resto;
        resto= a%b;
    }
    cout << "El m.c.d. es: " << b << endl;
}
```

Fin ejemplo 4.2.3 ■

Ejemplo 4.2.4 Desarrollar un programa para resolver la ecuación $x^3 + x - 1 = 0$. Se conoce que hay una solución en el intervalo $[0,1]$, ya que en el cero la función es negativa y en uno es positiva. El programa deberá solicitar el nivel de precisión deseado.

Recordemos, una función $f(x)$ continua en un intervalo $[a,b]$, tiene al menos una raíz si $f(a)f(b) < 0$ (si tiene distinto signo). El algoritmo para encontrarla puede ser:

1. Mientras que el intervalo no sea suficientemente pequeño:
   a) Calcular el centro del intervalo. Por tanto, obtenemos dos sub-intervalos.
   b) Seleccionar como nuevo intervalo el que mantenga la condición de existencia de la raíz (la función tiene distinto signo en sus extremos).
2. Obtener como solución el centro del intervalo final.

```
// Resuelve por bisección
// La solución en [0,1] de x^3+x-1=0
#include <iostream>
using namespace std;
int main()
{
    double precision;
    double izq= 0, der= 1, centro;

    cout << "Escribe la precisión deseada: ";
    cin >> precision;

    while (der-izq > precision) { // Mientras intervalo demasiado grande
        centro= (der+izq)/2;
        if (centro*centro*centro+centro-1<0) // Evaluamos en el centro
            izq= centro;
```

```
        else der= centro;
    }
    cout << "Solución: " << (der+izq)/2 << endl;
}
```

Fin ejemplo 4.2.4 ■

Ejercicio 4.1 Para calcular la parte entera del logaritmo en base 2 de un número entero positivo, se ha decidido implementar un algoritmo iterativo. La idea es que podemos ir dividiendo el número de forma sucesiva por 2 hasta que lleguemos a 1. El número de divisiones realizadas es el valor buscado. Escriba el algoritmo en C++.

Ejercicio 4.2 Escriba un programa para determinar si un número entero es un cuadrado perfecto. Recuerde que estos números son de la forma n^2, es decir, los número 1, 4, 9, 16, etc. Para ello, no podrá hacer uso de la función **sqrt**, sino que tendrá que ir calculando los cuadrados de los números desde el 1 en adelante.

4.3 Bucles *for*

En muchas ocasiones es necesario un algoritmo con un esquema similar al bucle controlado por contador que veíamos anteriormente. Concretamente, el esquema que se ha usado está compuesto de:

- Un paso de inicialización antes del bucle.
- Una condición que controla el final del bucle.
- Después de cada ejecución del cuerpo, una actualización antes de la siguiente iteración.

En el ejemplo 4.2.2 anterior —página 61— teníamos que contar 10 valores. Para ello, inicializamos con valor 1 y mientras que no lleguemos a 10 (menor o igual a 10) vamos incrementando de 1 en 1. Con una sentencia **while** se escribe como sigue:

```
i= 1;
while (i<=10) {
    // ...sentencias...
    i= i+1;
}
```

Otro ejemplo con el mismo esquema lógico puede escribirse si queremos hacer algo para todos los valores enteros pares en el intervalo $[0,100]$ de forma descendente. Podríamos usar el siguiente código:

```
contador= 100;
while (contador>=0) {
    // ...sentencias...
    contador= contador-2;
}
```

La forma más sencilla de entender estos trozos de código es darse cuenta de que son un bloque de sentencias que se ejecutan para ciertos valores determinados por: una *inicialización*, una *condición* y una *actualización*. Resultaría mucho más legible y fácil de entender si esas tres partes se escribieran de forma conjunta. Es más difícil de leer si las separamos, más aún si la inicialización está mucho antes en el código (está lejos del bucle) o el número de sentencias del cuerpo del bucle es alto (la actualización está lejos del comienzo del bucle).

La sentencia **for** es un tipo de bucle que permite expresar eficazmente este esquema. El formato es el siguiente:

for (*inicialización* ; *condición* ; *actualización*)
{ *sentencias* }

y el funcionamiento se representa gráficamente en la figura 4.2.

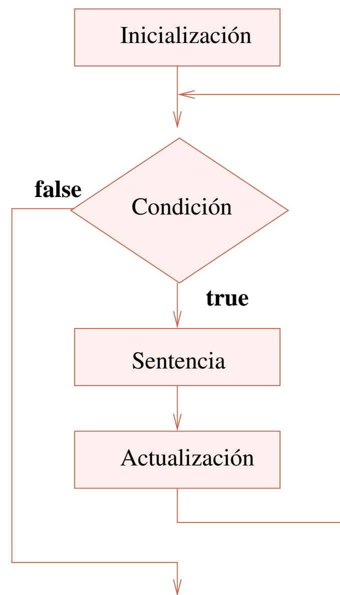


Figura 4.2
Funcionamiento del bucle *for*.

Podemos ver que contiene tres partes:

1. *Inicialización.* Se ejecuta una vez, antes de comenzar con las iteraciones.
2. *Condición.* Funciona de forma idéntica al bucle **`while`**, es decir, el bucle itera *mientras* se cumpla esta condición.
3. *Actualización.* Se ejecuta cada vez que se itera, es decir, cada vez que se ejecuta el cuerpo del bucle, se ejecuta la actualización.

Por lo tanto, el funcionamiento es idéntico al siguiente esquema:

inicialización
while (*condición*) {
    sentencias
    actualización
}

Observe que la parte de *inicialización* siempre se lleva a cabo. Por otro lado, el bucle puede que no itere ninguna vez, y si lo hace, la condición se evalúa siempre después de la actualización.

Esta equivalencia puede hacer que nos preguntemos por el sentido del bucle **`for`**. De nuevo, debemos insistir en que su formato se adecúa perfectamente a los esquemas de iteración habituales en los que necesitamos esas tres componentes. Con el bucle **`while`** separamos *inicialización, condición* y *actualización* de forma que la lectura puede ser mucho más difícil.

Por otro lado, las tres componentes son opcionales. Si no especificamos ninguna inicialización, simplemente no se ejecuta nada al principio. Al igual que la actualización (no se ejecuta nada tras la iteración). Algo más extraño resulta prescindir de la condición. Cuando no se incluye, se supone que siempre es **`true`** y por tanto el bucle no termina. Por ejemplo, es válido:

```
for (;;) {// Bucle infinito
   // ...sentencias...
}
```

que no terminará a menos que se use alguna instrucción especial[2] que rompa el ciclo.

Como ejemplo, modificamos un programa anterior en el que realizamos la suma de 10 valores. Ahora especificamos en una sola línea —con el bucle **for**— que deseamos iterar desde el 1 hasta el 10 y moviéndonos de uno en uno. La solución podría ser:

```
#include <iostream>
using namespace std;
int main()
{
   double suma;
   int i;

   cout << "Introduce 10 valores" << endl;

   suma= 0;
   for (i=1; i<=10; i=i+1) {
      cin >> valor;
      suma= suma+valor;
   }
   cout << "La suma total es:" << suma << endl;
}
```

Note además que la variable *i* es una variable como cualquier otra dentro del bucle y por lo tanto, se puede usar sin ningún problema. A pesar de ello se debe tener cuidado con su modificación, ya que crearíamos confusión y sería propenso a errores. En el ejemplo anterior, es fácil leer que se recorren todos los valores desde el 1 al 10, pero si modificamos *i* dentro del bucle, esta interpretación es errónea, e incluso puede ser costoso adivinar el comportamiento del bucle. Por consiguiente, procuraremos evitar estas situaciones evitando que la variable que controla el bucle se modifique dentro de éste.

Ejercicio 4.3 Escriba un programa que lea 100 valores e imprima el máximo y el mínimo en la salida estándar.

4.3.1 Nuevos operadores: incremento y decremento

Para sumar o restar 1 a una variable hemos usado las expresiones:

```
variable= variable+1; // Incremento
variable= variable-1; // Decremento
```

El lenguaje C++ también ofrece los operadores *unarios* ++ y −− de incremento y decremento, respectivamente. Con ellos, podemos hacer:

```
variable++; // Incremento
variable--; // Decremento
```

para aumentar o disminuir el valor de la variable en 1. Por ejemplo, un bucle que recorre los valores desde el 100 al 1 y los escribe en la salida estándar se puede escribir así:

```
for (i=100; i>=1; i--)
  cout << i << endl;
```

que corresponde a una forma más habitual de escribir.

Prefijo y Sufijo

Se distinguen dos tipos distintos de operadores de incremento/decremento:

1. *Preincremento*. El operador se escribe antes de la variable. El efecto es que primero se incrementa y luego “se usa” la variable.
2. *Postincremento*. El operador se escribe después de la variable. El efecto es que primero “se usa” la variable, y después se incrementa.

[2] Por ejemplo, break, return, goto o throw.

y de forma similar para el decremento. Algunos ejemplos de estos operadores se muestran en el siguiente trozo de código[3].

```
int a= 5, b;
b= a++;          // Valor de b, 5
cout << a;       // Se escribe 6
cout << a++*b++; // Escribe 30
cout << a*b;     // Escribe 42
cout << --a*b;   // Escribe 36
```

Note que estos operadores tienen mayor precedencia que los operadores aritméticos (realmente más que cualquiera de los que hemos visto hasta ahora).

Aunque para un programador experimentado en C++ estas expresiones no son difíciles de leer, algunos las consideran demasiado confusas, ya que son más propensas a errores (sobre todo si la expresión es mucho más compleja). En este sentido, nosotros intentaremos no "abusar" de esta nomenclatura.

Por otro lado, algunos autores indican que esta notación más concisa puede resultar útil al compilador. Por ejemplo, si escribo ++i el compilador puede realizar una traducción directa a alguna operación máquina que, directamente, implemente el incremento en 1. Además, se indica que el operador de preincremento puede resultar ligeramente más rápido.

A pesar de todo, para los tipos simples, muchas de estas mejoras se ven compensadas con la optimización final que realiza el compilador por lo que no tiene sentido escribir un código menos legible si no se va a obtener un mejor resultado. Además, la claridad en el código puede resultar mucho más importante que la eficiencia. De hecho, en general es recomendable dejar la optimización para etapas posteriores, cuando realmente sepamos que es necesaria.

Ejercicio 4.4 Indique el resultado de ejecutar el siguiente trozo de código:

```
1 int a= 2, b= 3, c= 5;
2 cout << ++a + ++b + ++c << endl;
3 a--; --b; --c;
4 cout << a++ + --b * c++ << endl;
5 cout << a << ' '<< b << ' '<< c << endl;
```

Finalmente, resulta interesante considerar que realmente podemos usar cualquiera de los operadores —*pre* o *post*— en nuestros programas. Por ejemplo, el siguiente bucle:

```
for (i=1; i<=100; i++)
  cout << i << endl;
```

obtiene el mismo resultado que el siguiente:

```
for (i=1; i<=100; ++i)
  cout << i << endl;
```

donde hemos usado el preincremento en lugar del postincremento.

En el nivel del curso en el que nos encontramos, las dos opciones son igualmente correctas. Sin embargo, es interesante indicar que un programador C++ probablemente se inclinará por la segunda. El motivo lo estudiará más adelante; ahora sólo indicaremos que en tipos más complejos, los operadores *pre* suelen ser más eficientes que los *post*. Sin embargo, para los tipos simples no es relevante.

Si revisa código de internet o de libros es posible que encuentre frecuentemente los *post*. En C es habitual usarlos de esta forma y como se utilizan sólo para los tipos simples, no se plantea el cambio.

[3]Más adelante veremos nuevos ejemplos de estos operadores, especialmente en el estudio de punteros y su aritmética, donde es muy habitual su uso

4.3.2 Nuevos operadores de asignación

Al igual que podemos abreviar el incremento y decremento, C++ permite abreviar las expresiones que tienen la siguiente forma[4]:

variable = variable operador (expresión);

utilizando nuevos operadores de asignación con la forma:

variable operador= expresión;

Algunos ejemplos de uso de estos operadores son[5]:

```
a+= b; // Equivale: a= a+b
a*= b; // Equivale: a= a*b
a%= b; // Equivale: a= a%b
```

4.3.3 El operador coma

En los bucles **for** resulta especialmente útil el operador coma. Su misión es la de encadenar expresiones. Es un operador binario —aplicado sobre dos expresiones— que obtiene como resultado la expresión derecha. En este sentido, parece inútil, pues sólo usamos el valor de la parte derecha. Sin embargo, el lenguaje garantiza que primero se evalúa la parte izquierda y después la parte derecha. En los bucles **for** lo podemos usar en la parte de inicialización e incremento, cuando deseemos que se realicen varias cosas.

Por ejemplo, si queremos inicializar la variable `suma` también con cero en el ejemplo anterior, podemos hacer:

```
cout << "Introduce 10 valores" << endl;
for (i=1, suma=0; i<=10; i++) {
   cin >> valor;
   suma= suma+valor;
}
cout << "La suma total es:" << suma << endl;
```

Observe que no hemos incluido ningún paréntesis en la parte de inicialización, ya que la asignación se evalúa antes. De hecho, el operador coma es el de menor precedencia de todos los incluidos en C++.

4.3.4 Ejemplos

Ejemplo 4.3.1 Escriba un programa que lea un número *n* entero positivo y escriba el factorial (*n*!) en la salida estándar.

El algoritmo consiste en multiplicar todos los valores desde el 2 hasta el entero `n`. Para ello, declaramos una variable que vamos multiplicando con cada uno de esos valores. Lógicamente, para multiplicar todos los valores usamos un bucle **for** que los recorre de uno en uno. Una solución puede ser:

```
#include <iostream>
using namespace std;
int main()
{
   int factorial, n;
```

[4] En este caso, estos operadores también pueden dar lugar a generación de código más eficiente.
[5] También se incluyen otros operadores como los operadores lógicos bit a bit que no se han visto aún.

```
    cout << "Introduzca un número entero positivo:";
    cin >> n;
    for (factorial=1, i=2; i<=n; i++)
       factorial*= i;
    cout << "Factorial: "<< factorial << endl;
}
```

Fin ejemplo 4.3.1 ■

Ejemplo 4.3.2 Escriba un programa que lea un número entero e indique en la salida estándar si es primo o no.

La solución a este problema se puede plantear de forma directa si consideramos la definición de número primo. Podemos comprobar todos los números menores que él, de forma que si encontramos un número que lo divida, sabemos que no es primo. Una solución puede ser:

```
#include <iostream>
using namespace std;
int main()
{
   int numero, i;
   bool esprimo;

   cout << "Introduzca el número" << endl;
   cin >> numero;

   esprimo= true;
   for (i=2; i<numero; ++i)
      if (numero%i == 0) // Es divisible
         esprimo= false;

   if (esprimo)
      cout << "El número es primo" << endl;
   else cout << "El número no es primo" << endl;
}
```

donde vemos que el bucle recorre todos y cada uno de los valores menores que `numero` para, en caso de encontrar un divisor, indicar con la variable `esprimo` que no es primo. Note que si el número es muy grande, el número de comprobaciones a realizar puede ser muy alto. Para hacer más rápido el algoritmo, podemos comprobar todos los valores hasta llegar a la raíz cuadrada del número[6]. Para incorporar esta mejora, modificamos el programa de la siguiente forma:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   int numero, i;
   bool esprimo;

   cout << "Introduzca el número" << endl;
   cin >> numero;

   esprimo= true;
   for (i=2; i<=sqrt(numero); ++i)
      if (numero%i == 0) // Es divisible
         esprimo= false;

   if (esprimo)
      cout << "El número es primo" << endl;
   else cout << "El número no es primo" << endl;
}
```

En esta solución también podemos encontrar un problema de eficiencia: en cada iteración el programa debe evaluar la condición de fin del bucle. Eso implica que repetirá muchas veces el

[6]Un número no primo tiene un divisor menor o igual que su raíz.

cálculo de la raíz cuadrada. Podemos mejorar el programa si eliminamos esa repetición innecesaria. Para ello, la calculamos una sola vez y almacenamos en una variable entera. La solución sería:

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  int main()
5  {
6     int numero, raiz, i;
7     bool esprimo;
8
9     cout << "Introduzca el número" << endl;
10    cin >> numero;
11
12    raiz= sqrt(numero);
13    esprimo= true;
14    for (i=2; i<=raiz; ++i)
15       if (numero%i == 0) // Es divisible
16          esprimo= false;
17
18    if (esprimo)
19       cout << "El número es primo" << endl;
20    else cout << "El número no es primo" << endl;
21 }
```

Ejercicio 4.5 Indique el comportamiento del programa cuando introducimos el número 1 como dato a analizar.

Por otro lado, si el programa detecta un valor que divide al número, asigna `false` a la variable *esprimo*, pero sigue buscando divisores, es decir, el bucle no termina a pesar de haber encontrado ya un divisor. Podemos modificarlo para que se detenga. Para ello, cambiamos la condición haciendo que el bucle siga buscando un divisor mientras no lleguemos a la raíz **Y** siga siendo primo. Concretamente, el bucle sería:

```
for (i=2; i<=raiz && esprimo; ++i)
   if (numero%i == 0) // Es divisible
      esprimo= false;
```

Ejercicio 4.6 Indique cuál es el efecto si modificamos el trozo final por:

```
if (esprimo)
   cout << "El número es primo" << endl;
else cout << "El número no es primo (divisor:"
          << i << ")" << endl;
```

Finalmente, resulta interesante destacar que el compilador realiza una conversión de tipos automática en varios lugares. Así, la raíz cuadrada se calcula para un dato real, aunque el dato que se introduce para calcular la raíz cuadrada es entero. El compilador convierte ese entero a real de forma automática[7]. De la misma forma, el valor que se obtiene desde `sqrt` es un real, y se asigna a un entero. En este caso, el compilador realiza la conversión eliminando la parte decimal[8].

Fin ejemplo 4.3.2 ■

[7] La conversión de `int` a `double` es automática. Sin embargo, en algunos compiladores podría generarse un error de compilación porque hay varias versiones de `sqrt` disponibles (véase sobrecarga de funciones). Es este caso, el usuario debe indicar la versión que desea utilizar(por ejemplo, realizando una conversión explícita, con `static_cast<double>(numero)`.

[8] Esta conversión es más delicada, ya que se pierde información. En algunos compiladores se genera un mensaje de aviso para indicarlo, por si el programador no es consciente y realmente esa transformación es errónea.

Ejercicio 4.7 El programa anterior aún se puede mejorar si tenemos en cuenta un par de condiciones más:

1. El único entero primo y par es el dos.
2. Un entero n mayor que 3, sólo puede ser primo si verifica $n^2 \mod 24 = 1$.

Modifique el programa anterior para incorporar estas propiedades.

Ejemplo 4.3.3 Modifique el programa anterior para que lea el número tantas veces como sea necesario hasta que sea un entero positivo.

El problema surge porque el usuario podría introducir un valor negativo. En este caso, el programa intentaría calcular su raíz, lo que provoca un error en ejecución. Para evitarlo, podemos volver a pedir el número en el caso de que sea negativo. Esta petición se puede repetir muchas veces, hasta que se introduzca un valor positivo. Por tanto, podemos resolverlo con un bucle.

El bucle se repite mientras se cumpla una condición, es decir, tenemos que volver a pedir el dato mientras sea negativo. La solución tiene la forma:

```
cout << "Introduzca el número: " << endl;
while (numero<1)
   cin >> numero;
```

Sin embargo, la primera vez que llega al bucle no hemos dado ningún valor a la variable. Podemos dar el primer valor con:

```
cout << "Introduzca el número: " << endl;
cin >> numero;
while (numero<1)
   cin >> numero;
```

Y considerando que sólo entra al bucle si nos hemos equivocado, podemos incluso añadir un mensaje de aviso:

```
cout << "Introduzca el número: " << endl;
cin >> numero;
while (numero<1) {
   cout << "El número debe ser positivo: " <<endl;
   cin >> numero;
}
```

En este ejemplo, hemos tenido que introducir una *"lectura adelantada"* para poder disponer del valor en la primera comprobación.

Fin ejemplo 4.3.3 ■

Ejercicio 4.8 Escriba un programa para resolver una ecuación de primer grado ($ax+b=0$). Tenga en cuenta que se debe repetir la lectura de a hasta que sea distinto de cero.

Ejercicio 4.9 Escriba un programa que lea una secuencia de números reales e imprima la media. Realice dos versiones:

1. Un programa para leer 100 números reales.
2. Un programa que lee 100 números o hasta que se introduce un valor negativo. El valor negativo no se deberá incluir en la suma.

Finalmente, resulta conveniente insistir en que la lectura desde la entrada estándar se supone correcta, es decir, suponemos que el usuario introduce un número[9]. En estos casos, se considera

[9]Como prueba, ejecute el programa y observe el comportamiento en caso de introducir una letra en lugar de un número. Comprobará que no sólo falla esa lectura, sino que ninguna más del programa se ejecutará.

que ha habido un error de lectura, por lo que es necesario usar otras herramientas más concretas del sistema de E/S para resolverlo. A este nivel no nos interesa conocer estos detalles, así que supondremos que el usuario introduce un dato adecuado al tipo solicitado cada vez que el programa se lo solicita.

4.4 Bucles *do/while*

Los bucles `do−while` son el tercer tipo de estructura de repetición que nos ofrece el lenguaje C++. Son parecidos a los bucles `while` en cuanto a funcionamiento, pues se componen de una condición y un cuerpo de bucle, pero se diferencia en que éste se ejecuta al menos una vez, ya que es después de su ejecución cuando se comprueba la condición por primera vez. El formato es:

do {
    sentencias
} **while** (*condición*);

en donde se pone de manifiesto que el cuerpo está antes que la condición. En la figura 4.3 se representa gráficamente su comportamiento.

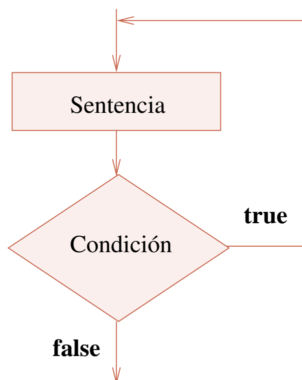


Figura 4.3
Funcionamiento del bucle *do-while*.

Lógicamente, este bucle se utilizará cuando sepamos que se debe ejecutar al menos una vez, y que después de ésta es cuando hay que comprobar la condición. Un ejemplo sencillo, es la lectura de un valor[10]:

```
do {
   cin >> numero;
} while (numero<1);
```

Ejemplo 4.4.1 Escriba un programa que ofrezca un menú con 3 opciones. Deberá solicitar una de ellas y escribir la seleccionada hasta que se escoja la tercera (de finalización). Utilice un bucle `do/while` para controlar la iteración.

[10] Aunque el uso parece claro, según mi experiencia, aparecen pocos casos de uso de esta construcción `do−while`. La mayoría de las veces se tiende a usar bucles `while`. Tal vez, la explicación sea la que indica Stroustrup[31] (página 142) cuando dice que lo considera una fuente de errores y confusión.

```
#include <iostream>
using namespace std;
int main()
{
   int opcion;

   do {
      cin >> opcion;
      switch (opcion) {
         case 1:
            cout << "Opción 1" << endl;
            break;
         case 2:
            cout << "Opción 2" << endl;
            break;
         case 3:
            cout << "Fin" << endl;
            break;
         default:
            cout << "Sólo 1,2,3" << endl;
      }
   } while(opcion!=3);
}
```

Fin ejemplo 4.4.1 ■

4.5 Estructuras de control anidadas

Los bucles son sentencias igualmente válidas para aparecer dentro de otras estructuras de control, ya sean sentencias condicionales u otros bucles. Cuando un bucle está incluido en otro se habla de bucles anidados. Note que un bucle —junto con su cuerpo— se puede considerar una única sentencia. Por ejemplo, si queremos escribir todas las posibles parejas de valores *(i,j)* con los enteros 1,2,3,4,5, podemos escribir lo siguiente:

```
#include <iostream>
using namespace std;
int main()
{
   int i, j;

   for (i=1; i<=5; ++i)
      for (j=1; j<=5; ++j)
         cout << i << ',' << j << endl;
}
```

donde aparece un bucle como la única sentencia de otro bucle (observe que no aparecen llaves). En este caso, para cada valor de `i` se itera `5` veces sobre `j`. Por lo tanto, en la salida obtendremos las parejas *(1,1)*, *(1,2)*, *(1,3)*, ... *(5,5)*.

Note, además, que la variable del primer bucle se puede usar dentro del segundo. Es más, podemos hacer que las iteraciones del bucle interior dependan del exterior. Por ejemplo, podemos escribir todas las parejas de valores `(i,j)` pero con la condición `j>=i`. Se podría hacer:

```
#include <iostream>
using namespace std;
int main()
{
   int i,j;

   for (i=1; i<=5; ++i)
      for (j=i; j<=5; ++j)
         cout << i << ',' << j << endl;
}
```

de forma que para `i=5` sólo se escribe *(5,5)*.

Ejemplo 4.5.1 Desarrolle un programa que escriba todos los números primos menores que uno dado.

El problema de la primalidad de un número lo hemos resuelto anteriormente. Utilizando el mismo código, sólo necesitamos añadir un bucle que recorra todos los posibles valores: desde el 2 hasta el número de entrada. Para cada uno de ellos comprobamos si es primo, en cuyo caso se escribe en la salida estándar. El código puede ser el siguiente:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   int n;
   int numero, raiz, i;
   bool esprimo;

   cout << "Introduzca el número final" << endl;
   cin >> n;

   for (numero=2; numero<=n; ++numero){
      raiz= sqrt(numero); // Máximo divisor a comprobar
      esprimo= true;
      for (i=2; i<=raiz; ++i)
         if (numero%i == 0) // Tiene divisor
            esprimo= false;
      if (esprimo)
         cout << numero << endl;
   }
}
```

Fin ejemplo 4.5.1 ■

Ejercicio 4.10 Escriba un programa que obtenga en pantalla las 10 tablas de multiplicar (del 1 al 10), con 10 entradas cada una (del 1 al 10).

Ejercicio 4.11 Escriba un programa que lea un número de la entrada estándar y escriba su tabla de multiplicar. Al finalizar, preguntará si se desea escribir una nueva tabla, si se responde *'S'* repetirá el proceso y si se responde *'N'* finalizará.

Ejemplo 4.5.2 Se quiere comprobar si la expresión

$$\frac{\pi^3}{32} = \sum_{n=0}^{\infty} \frac{-1^n}{(2n+1)^3}$$

obtiene con pocos sumandos una aproximación aceptable (por ejemplo, 4 decimales) del valor de π.

Para resolver el problema escribimos un programa que imprima en la salida estándar el valor de la suma para 1 sumando, para 2, para 3, ... hasta 100. De esta forma podremos observar el momento en el que obtenemos un valor cercano a π.

Para calcular una aproximación de π —con los sumandos desde cero a un valor `i`— podemos usar el siguiente código:

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

int main()
{
   double valor;
```

```
    double acumulado;
    int i, n;

    cin >> i; // Para 0..i
    acumulado= 0;
    for (n=0; n<=i; n++) {
       valor= 1.0/( (2*n+1) * (2*n+1) * (2*n+1) );
       acumulado+= (n%2==0)?valor:-valor;
    }
    acumulado*= 32;
    cout << "Valor con "<< i <<" sumandos: "
         << pow(acumulado,1.0/3) << endl;
}
```

Observe que algunas constantes tienen un cero como parte decimal para hacerlas de tipo **double** (evitando la división entera). Por otro lado, el valor de -1^n se ha calculado comprobando si n es par (con el operador ternario "*?:*").

Como deseamos un programa para escribir los valores con un sumando, con dos, etc. podemos añadir un bucle que itera sobre todas estas posibilidades. Así, la solución puede ser:

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

int main()
{
    double valor;
    double acumulado;
    int i, n;

    for (i=0; i<100; i++) {
       acumulado= 0;
       for (n=0; n<=i; n++) {
          valor= 1.0/( (2*n+1) * (2*n+1) * (2*n+1) );
          acumulado+= (n%2==0)?valor:-valor;
       }
       acumulado*= 32;
       cout << "Valor con "<< i <<" sumandos: "
            << pow(acumulado,1.0/3) << endl;
    }
}
```

A pesar de ello, ésta no es la mejor solución, ya que no es necesario usar dos bucles anidados. Observe que repetimos múltiples veces el mismo trabajo. Por ejemplo, para 10 sumandos tenemos que calcular y acumular los primeros 10 sumandos de la sumatoria. Para 20 sumandos tenemos que acumular también esos primeros 10 sumandos (además de los 10 siguientes).

Fin ejemplo 4.5.2 ■

Ejercicio 4.12 Modifique el ejemplo anterior para obtener una solución más eficiente.

Ejercicio 4.13 Se quiere comprobar si la expresión

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

obtiene con pocos sumandos una aproximación aceptable (por ejemplo, 4 decimales) del valor de e. Escriba un programa que pida el número de sumandos y escriba el resultado del cálculo.

4.6 Declaración y ámbito de una variable

Hasta ahora hemos separado la declaración de variables de las sentencias que las usan. Para ello, al principio de **main** declaramos todos los datos necesarios para el resto del programa. Sin embargo, la declaración se puede realizar en cualquier punto del programa[11].

En C++ una variable declarada dentro de **main**[12] no existe hasta que se encuentra su declaración. Por lo tanto, el compilador se encarga de *crearlas* cuando se llega a ellas. El tiempo de vida llega hasta el final del bloque en el que aparece su declaración (un bloque es una sección de código delimitado por un par {}). Por tanto, en ese punto, el compilador *destruye* la variable. El trozo de código desde la declaración hasta el final del bloque, la zona donde se puede usar la variable, se conoce como el *ámbito de la variable*.

En general, es recomendable reducir el ámbito de una variable. Con ello, el código puede quedar más claro, ya que las variables no existen en zonas donde no se necesitan y sólo deben tenerse en cuenta cuando son realmente necesarias. Además, evitamos modificar el valor de esas variables en zonas donde no se deben usar. Note, además, que es mucho más sencillo programar cuando se deben tener menos cosas —en este caso, variables— en cuenta. En este sentido, es recomendable hacer que un "recurso" sea lo más local posible.

Por ejemplo, podemos reescribir el programa que implementa el algoritmo de bisección —página 62— de la siguiente forma:

```
double precision;

cout << "Escribe la precisión deseada: ";
cin >> precision;

double izq= 0,der= 1;
while (der-izq > precision) {
   double centro= (der+izq)/2;
   if (centro*centro*centro+centro-1 < 0) // Evaluamos
      izq= centro;
   else der= centro;
}
cout << "Solución: " << (der+izq)/2 << endl;
```

donde la declaración de `izq` y `der` se ha adelantado (no se conocen antes de la declaración) hasta el punto donde se van a usar. Además, la variable `centro` se ha introducido dentro del bucle, que es donde se utiliza. Ahora es más legible y más claro el objetivo de crear esta variable.

Un segundo ejemplo resulta de la modificación del programa anterior sobre la lista de números primos. El código podría ser:

```
int n;
cout << "Introduzca el número final" << endl;
cin >> n;

int numero;
for (numero=2; numero<=n; ++numero){
   int raiz= sqrt(numero); // Máximo divisor a comprobar
   bool esprimo= true;
   int i;
   for (i=2; i<=raiz && esprimo; ++i)
      if (numero%i == 0) // Tiene divisor
         esprimo= false;
   if (esprimo) cout << numero << endl;
}
```

Además, el lenguaje C++ permite realizar una declaración dentro de la parte de inicialización de un bucle **for**[13]. Esto resulta útil porque podemos incluir la declaración de la variable contador

[11] Esta tendencia a separar las declaraciones es más habitual en C, donde las declaraciones se realizan, de forma obligada, al principio de un bloque, es decir, después del carácter '{' que delimita un bloque. Aunque a partir de C99 esta restricción desaparece.

[12] Estas variables son locales a **main** y, como veremos más adelante, se denominan automáticas.

[13] También lo permite en la condición de la estructura condicional **if**, aunque es menos habitual.

del bucle, haciendo que su ámbito se limite a éste[14]. Por ejemplo, podemos modificar el trozo de código anterior como sigue:

```
int n;
cout << "Introduzca el número final" << endl;
cin >> n;

for (int numero=2; numero<=n; ++numero){
   int raiz= sqrt(numero); // Máximo divisor a comprobar
   bool esprimo= true;
   for (int i=2; i<=raiz && esprimo; ++i)
      if (numero%i == 0) // Tiene divisor
         esprimo= false;
   if (esprimo) cout << numero << endl;
}
```

4.6.1 Ocultación de variables

Hasta ahora hemos supuesto que no es legal repetir un nombre de variable. Sin embargo, al existir distintos ámbitos, el lenguaje C++ permite que se puedan volver a usar. Así, cuando se declara un nombre en un bloque puede existir otro idéntico en un bloque exterior. En este caso, el nuevo nombre oculta el antiguo dentro de su ámbito. En general, aunque es válido, intentaremos evitar esta duplicidad para no encontrarnos con este problema[15].

4.7 Programación estructurada y sentencia *goto*

El lenguaje C++ ofrece la sentencia `goto` para realizar un salto incondicional. El formato es:

goto *identificador ;*

que provoca un salto incondicional a la sentencia del programa que hemos etiquetado como *identificador* (se escribe la etiqueta y dos puntos precediendo a la sentencia). Por tanto, su ejecución implica que el programa salta directamente a la sentencia indicada.

Esta sentencia no es necesaria en la programación de alto nivel que se realiza normalmente[16]. Además, ya en los años 60, se observó que el uso de esta sentencia provocaba códigos más propensos a errores, difíciles de leer y de mantener. Por ello, distintos autores escriben sobre la conveniencia de prescindir de `goto`, así como de la posibilidad de usar únicamente la estructura secuencial, condicional e iterativa. Estos cambios dan lugar al comienzo de la *programación estructurada*, donde los algoritmos sólo tienen un punto de entrada y uno salida, evitando la transferencia de control incondicional y usando únicamente esas tres estructuras.

Nosotros respetaremos los principios de la programación estructurada y, por tanto, prescindiremos del uso de la sentencia `goto`. En este tipo de programación, cualquier algoritmo se escribe usando las tres estructuras indicadas (note que se pueden anidar tanto como se desee). Por supuesto, es muy difícil escribir un algoritmo complejo usando únicamente estas sentencias de C++. Podríamos encontrar miles de líneas con miles de posibles flujos de control. En estos problemas, es necesario agrupar líneas en "bloques de sentencias" o "módulos" destinados a resolver un subproblema determinado, de forma que su unión nos lleve a la solución deseada. De este modo, la programación estructurada se entiende como aquella que, usando la modularización —que veremos en los temas siguientes— y esas tres estructuras básicas, resuelve cualquier problema.

[14] Al algunos compiladores antiguos, antes del estándar del 98 consideraban que el ámbito llegaba hasta el bloque que incluía el bucle. Si tiene un compilador actualizado, el ámbito se limita al bucle.

[15] Algunas veces creamos dos nombres iguales "sin querer". Esto se da especialmente con variables *globales* (que estudiaremos más adelante) aunque, como veremos, en el caso de ocultar variables globales es posible referenciarlas.

[16] Aunque resulta de interés, por ejemplo, cuando se genera código automático o se desea programar a bajo nivel con el objetivo de optimizar al máximo un trozo de código

4.7.1 Sentencias continue y break

La sentencia **break** se puede considerar un caso especial, ya que provoca un salto incondicional fuera del bloque donde se encuentra. De hecho, ya la hemos usado: en la sentencia **switch** permite salirnos de ésta después de haber ejecutado las sentencias correspondientes a un caso de terminado.

Este hecho hace que algunos autores consideren esta sentencia **switch** poco recomendable, justificándolo porque su uso rompe los principios de la programación estructurada. Efectivamente, es una sentencia algo incómoda. Por ejemplo, declarar una variable dentro del bloque **switch** puede generar una situación extraña, ya que el flujo podría saltarse la declaración.

A pesar de establecer un salto incondicional, esta sentencia funciona así por conveniencia, especialmente para que el código C siga siendo válido en C++. Además, si se usa con cuidado, aprovechando su capacidad para enfatizar el esquema de un algoritmo de caminos múltiples, sin provocar situaciones erróneas, es una sentencia muy útil.

En la práctica, la sentencia **switch** no suele generar problemas, ya que el código que se escribe se razona considerando cómo funciona, es decir, sabiendo que el flujo pasa a la etiqueta correspondiente hasta la sentencia **break**. Es probable que la use en muchos casos antes de ser consciente de alguna situación de error que, en cualquier caso, aparecerá en la compilación.

Por tanto, correctamente usada, podemos considerar que la sentencia **switch** es válida en la programación estructurada. Tenga en cuenta que se podría escribir fácilmente como varios **if** anidados —aunque resultaría menos legible— y aparece con un esquema similar en otros lenguajes basados en la programación estructurada (como por ejemplo, *Pascal*). Por tanto, la usaremos cuando sea recomendable.

También podríamos usar **break** en el caso de los bucles, ya que provocaría un salto incondicional fuera de la estructura. Este caso sería similar a un **goto** a la siguiente sentencia. Por tanto, no lo admitiremos.

Por otro lado, la sentencia **continue** se puede usar en un bucle para que dé por terminada la iteración, es decir, salte al final del cuerpo del bucle para seguir con la siguiente iteración. De la misma manera, también la evitaremos en nuestras soluciones.

4.8 Problemas

Problema 4.1 Considere el siguiente bucle:

```
while (true) {
  //...sentencias...
}
```

¿Qué efecto tiene? ¿Cuándo terminará?

Problema 4.2 La sucesión de valores:

$$1, 1, 2, 3, 5, 8, 13, 21, \cdots$$

se conoce como la sucesión de Fibonacci. Los valores se pueden calcular con $F_{n+2} = F_n + F_{n+1}$ con $F_1 = 1$ y $F_2 = 1$. Escribir un programa que pregunte el valor de n y escriba en la salida estándar el valor de F_n.

Problema 4.3 Considere la siguiente sucesión de valores:

$$x_i = \begin{cases} 1 & \text{si } i = 0 \\ \frac{1}{2}\left(x_{i-1} + \frac{a}{x_{i-1}}\right) & \text{si } i > 0 \end{cases} \tag{4.2}$$

El valor de x_i tiende a la raíz cuadrada de a, cuando i tiende a infinito. Para mostrar esta propiedad en la práctica, desarrolle un programa que lea el valor de a y de n a fin de escribir en la

salida estándar todos los valores de la sucesión desde el cero hasta x_n, seguidos de la raíz cuadrada de a (calculada con `sqrt`).

Problema 4.4 Se desea resolver el problema de realizar la multiplicación de un número indeterminado de valores reales positivos. Para ello, se ha desarrollado el siguiente programa:

```
#include <iostream>
using namespace std;
int main()
{
   double resultado, numero;

   do {
      cin >> numero;
      resultado*= numero;
   } while (numero!=0);
   cout << "El resultado es: " << resultado << endl;
}
```

donde se ha establecido un final con "centinela" (el valor cero no se encuentra en los valores válidos de entrada). ¿Qué errores detecta en este programa? Modifíquelo para que funcione correctamente, proponiendo una solución con el bucle post-test `do/while` y otra con el bucle pre-test `while`.

Problema 4.5 La desviación media de un conjunto de n valores x_i se define como:

$$\frac{1}{n}\sum_{i=1}^{n} |\, x_i - \overline{x}\,|$$

donde $\overline{x}$ es la media de los n valores. Implemente un programa que lea el valor n y los valores x_i y escriba la desviación media en la salida estándar. Nota: Considere que los valores x_i se dan dos veces, una para calcular la media y otra para calcular la desviación media.

Problema 4.6 El doble factorial de n se define como:

$$n!! = n(n-2)(n-4)....$$

hasta el valor 1 ó 2 (dependiendo de si n es par o no). Escriba un programa que lea el valor n (entero positivo) y escriba su doble factorial en la salida estándar. Para ello, asegúrese de que la lectura es de un valor entero positivo.

Problema 4.7 Escriba un programa que lea números reales positivos hasta que se introduzca el cero. Como resultado, deberá escribir el número, media, mínimo y máximo de los datos introducidos.

Problema 4.8 Considere el siguiente trozo de código, donde se realiza una estimación del punto medio entre los valores `n`, m:

```
#include <iostream>
using namespace std;
int main()
{
   cin >> n >> m;
   i= n; j= m;
   while (i!=j) {
     i++; j--;
   }
   cout << "Se encontraron en " << i << endl;
}
```

¿Qué errores detecta en este programa? Modifíquelo para que funcione correctamente.

Problema 4.9 Use el método de bisección para obtener un valor aproximado de la raíz cuadrada de un valor x. Este método, busca la raíz en un intervalo *(a,b)* calculando el cuadrado del punto central del intervalo. Si el cuadrado de ese punto es menor que x, la solución está en el subintervalo derecho, y si es mayor, en el subintervalo izquierdo. Esta búsqueda puede seguir hasta que el intervalo es suficientemente pequeño (el error será menor que el tamaño del intervalo). Escriba un programa para que lea un valor x y un error ε, y calcule la raíz cuadrada con un error máximo ε.

Problema 4.10 Desarrolle un programa para escribir en la salida estándar la descomposición en números primos de un número entero. Por ejemplo, para el 12 deberá escribir los números *2,2,3*. Note que cuando buscamos un divisor de un número desde el 2 en adelante, el primer número encontrado es un número primo que forma parte de su descomposición.

Problema 4.11 Desarrolle un programa para que escriba en la salida estándar la configuración de todas las fichas del dominó. Para ello, escribirá todos los pares *(i,j)* que corresponden a una ficha, sin repetirlos y suponiendo que el valor cero corresponde a blanco.

Problema 4.12 Hace miles de años, Nicómano descubrió que podía calcular los cubos de los números naturales sumando números impares. Por ejemplo, el cubo de 1 es la suma del primer impar (1), el cubo de 2 es la suma de los dos siguientes (*3+5*), el cubo de 3 es la suma de los tres siguientes (*7+9+11*), etc. Desarrolle un programa que escriba en pantalla los 100 primeros cubos usando esta propiedad.

Problema 4.13 Escriba un programa que lee un valor n y use las siguientes ecuaciones:

$$\begin{array}{ccc} x_0 = \sqrt{2} & y_0 = 0 & \pi_0 = 2 + \sqrt{2} \\ x_{n+1} = \frac{1}{2}\left(\sqrt{x_n} + \frac{1}{\sqrt{x_n}}\right) & y_{n+1} = \frac{(1+y_n)\sqrt{x_n}}{x_n+y_n} & \pi_n = \pi_{n-1} \cdot y_n \cdot \frac{x_n+1}{y_n+1} \end{array} \tag{4.3}$$

para calcular π_n, una aproximación del número π.

Problema 4.14 Considere un programa que lee un mensaje (una secuencia de caracteres terminada en un carácter #). Escriba un programa que determine el número de frases que la componen. Para ello, considere que el mensaje es una secuencia de frases separadas por puntos.

5 Funciones

Introducción 81
Funciones 82
Diseño de funciones 98
Problemas 107

5.1 Introducción

Los programas que hemos resuelto hasta el momento se limitan a unas pocas líneas. Sin embargo, la complejidad de los problemas es normalmente mucho más grande, llegando a alcanzar fácilmente programas con miles de líneas de código. Resolver un problema de ese tamaño, como una secuencia de instrucciones dentro del bloque `main`, resulta prácticamente imposible.

Para poder enfrentarse a este tipo de problemas, es indispensable poder dividirlo en subproblemas que sean más simples de resolver y que permitan integrarse para obtener la solución deseada. Es fácil entender esta forma de abordar un problema complejo. Por ejemplo, imagine la dificultad que existe en el hardware de un ordenador, donde hay cientos de millones de transistores.

Intentar resolver el problema de diseñar el sistema, indicando el papel y la relación entre todos los transistores, resulta impensable. En lugar de ello, se divide el problema de forma que resolvemos cada uno de los subproblemas de forma independiente. Si diseñamos una CPU, no nos interesa la dificultad o los detalles de la solución que se da a la memoria principal. Lo único que debemos tener en cuenta es la *interfaz*, la forma en que se conecta al resto del sistema, es decir, las patillas que tiene nuestro chip así como la función de cada una de ellas.

De la misma forma, podemos plantear el mismo enfoque para resolver el problema de diseñar la CPU, que puede contener millones de transistores. De nuevo, este subproblema se plantea dividiéndolo en otros aún más pequeños. Así, tenemos que resolver el diseño de una unidad aritmético-lógica. Para ello, no nos interesa cómo se resuelven otros problemas como podría ser la unidad de control, sino que una vez que sabemos cómo se va a conectar con los demás dispositivos internos, podemos olvidarnos de ellos y centrarnos en su diseño. Por supuesto, la integración final de todas las soluciones nos permite obtener el sistema deseado.

En el desarrollo de software el planteamiento es similar, ya que los problemas más complejos se deben realizar dividiéndolos en subproblemas que sean más fáciles de resolver. La solución a cada uno de estos subproblemas la denominaremos módulos y la unión de estos módulos nos dará la solución buscada.

La herramienta más básica que nos ofrece el lenguaje C++ para definir módulos son las *funciones*. Ya conocemos algunos ejemplos, como por ejemplo la función **sqrt**. El lenguaje ya tiene resuelto este subproblema, de forma que si queremos desarrollar un programa que resuelve una ecuación de segundo grado, no necesitamos “mezclar” los detalles de nuestro problema con los del cálculo de la raíz cuadrada. Para resolver nuestro problema, sólo es necesario saber que la forma de hacer uso de ese módulo es escribir el identificador **sqrt** y el número real entre paréntesis para obtener la raíz. En este ejemplo tan sencillo ya podemos distinguir algunas ventajas:

- Facilita el *desarrollo*. Por un lado, los subproblemas son más fáciles de resolver; por otro, es más fácil resolver un problema contando con algunas partes ya resueltas.
- Facilita la *reusabilidad*. Una vez resuelto el problema del módulo **sqrt**, podemos usarlo múltiples veces.
- Facilita el *mantenimiento*. Podemos someter el módulo **sqrt** a distintas pruebas para garantizar que funciona correctamente. Si nuestro programa falla, el error probablemente no está en ese módulo. Además, si tenemos varios módulos y nuestro programa falla, es más sencillo limitar el error a uno de ellos, de forma que sólo tendremos que revisar ese subproblema.

Como ejemplo, volvamos al problema de la página 76, donde escribimos los números primos hasta un número dado *n*:

```
for (int numero=2; numero<=n; numero++){
   int raiz= sqrt(numero);
   bool esprimo= true;
   for (int i=2; i<=raiz && esprimo; ++i)
      if (numero%i == 0) // Es divisible
         esprimo= false;
   if (esprimo) cout << numero << endl;
}
```

Este programa podemos dividirlo en dos módulos distintos, incluyendo el subproblema que se refiere a determinar si un número es primo o no. La solución podría ser algo así:

```
for (int numero=2; numero<=n; numero++)
   if (EsPrimo(numero))
      cout << numero << endl;
```

donde se hace uso de un módulo *EsPrimo*, que indica si un número es primo. Observe que en este listado no nos importa cómo se resuelve ese subproblema. Sólo necesitamos saber que le pasamos un entero entre paréntesis y que nos devuelve un booleano indicando la primalidad.

Por otro lado, tendremos que decirle al compilador la forma en que se implementa la función *EsPrimo*. Es decir, tendremos que *definir la función*.

5.2 Funciones

La forma de la definición de una función es la siguiente:

> tipo_función *nombre_función* (parámetros formales)
> {
> ... código función (cuerpo) ...
> }

que nos permite crear un nuevo módulo *nombre_función* donde se encapsula el código —cuerpo de la función— para realizar una determinada tarea. Para entender mejor cada una de las partes, es más sencillo mostrarlo con un ejemplo concreto:

```
bool EsPrimo (int n)
{
   int raiz= sqrt(n);
   bool esprimo= true;
   for (int i=2; i<=raiz && esprimo; ++i)
      if (n % i==0) // Es divisible
         esprimo= false;
   return esprimo;
}
```

donde podemos ver que:

- La primera línea, que se denomina la *cabecera*, especifica la sintaxis para poder llamar a la función. Concretamente:
  - *tipo_función* es el tipo del dato devuelto al finalizar la función. En el ejemplo anterior el tipo devuelto es **bool**, es decir, el resultado de la función será un valor booleano (indicando la primalidad). Otro ejemplo es la función **sqrt** que hemos usado en varias ocasionas y que devuelve el tipo **double**.
  - *nombre_función* identifica a la función. Este nombre es un identificador válido —recuerde los nombres válidos para variables— que nos permite referenciarla. En nuestro ejemplo, al escribir *EsPrimo(numero)*, el compilador puede asociar esta *llamada* a la función con nombre *EsPrimo*.
  - *parámetros formales* es una lista de cero o más declaraciones separadas por comas en las que se especifica, uno a uno, los identificadores y tipos de las variables que se pasan desde el código que llama a la función. En el ejemplo, el único dato que requiere el módulo es un número entero, para determinar si es primo. Por tanto, en este caso la lista de parámetros es un único valor de tipo **int**.
    Observe que el parámetro tiene el nombre *n* para poder usarlo en el cálculo, es decir, dentro de la función.
- *código función* es el conjunto de sentencias que implementan la acción para la que se crea la función. También se denomina el *cuerpo* de la función. Observe que es un bloque de sentencias delimitado, como sabemos, por el par de caracteres {}.
  El contenido del cuerpo no tiene nada de especial, ya que son sentencias y declaraciones similares a las que hemos usado en el bloque **main**. De hecho, este bloque **main** no es más que otra función.
  La función termina con una sentencia **return**, seguida de una expresión que indica el valor a devolver. Observe que en el ejemplo devolvemos la variable *esprimo*, que es de tipo **bool**, como la función.

Para hacer una llamada a la función, la sintaxis es:

nombre_función (argumentos)

donde *nombre_función* es el nombre que se ha indicado en la *definición* para que el compilador la identifique. Los *argumentos* [1] son una lista de tantos parámetros como indica la definición. El compilador gestiona la llamada:

1. Estableciendo una correspondencia, uno a uno y en el mismo orden, de los argumentos con los parámetros formales.
2. Pasando el control del flujo a la función, para ejecutar el cuerpo de ésta.
3. Devolviendo el valor indicado con **return** y el flujo de control al lugar desde el que se hizo la llamada.

[1] En la bibliografía también encontrará el nombre de *parámetros reales* o incluso *parámetros actuales*.

Ejemplo 5.2.1 Escriba una función para leer un entero positivo desde la consola.

El objetivo de esta función es devolver un nuevo número entero, leído desde la entrada estándar, con la condición de que sea positivo. Para ello, deberá solicitar el entero y repetir la lectura hasta que cumpla esta condición. Note que:

- El resultado es un número entero. Por tanto, la función será de tipo `int`.
- Como nombre de la función podemos seleccionar *LeerIntPositivo*, que facilita la lectura y comprensión del código.
- La función no requiere de ningún argumento. No necesitamos saber nada, es decir, no necesita ningún dato de entrada, por lo tanto la lista de parámetros formales será vacía.

Teniendo en cuenta estas consideraciones, la función podría ser:

```
int LeerIntPositivo()
{
   int valor;
   cout << "Introduzca entero positivo:";
   cin >> valor;
   while (valor<1) {
      cout << "El valor no es positivo. Introduzca otro:";
      cin >> valor;
   }
   return valor;
}
```

donde hemos tenido que declarar la variable *valor* para realizar la lectura. Finalmente, la última lectura se devuelve como resultado.

Fin ejemplo 5.2.1 ■

Ejercicio 5.1 Implemente una función *EsVocal* que devuelva si un carácter es una vocal o no.

5.2.1 Un programa con funciones. Declaración y definición

Una vez introducidas las funciones, el primer problema que se plantea es la forma de incluirlas en un programa, es decir, la nueva estructura del programa. En primer lugar, debemos tener en cuenta que el orden de definición de las funciones es libre, es decir, podemos definirlas en el orden que deseemos. Esto implica que podemos escribirlas al principio del archivo, antes del `main`, después del `main`, incluso algunas antes y otras después. Sin embargo, tengamos en cuenta que:

- Es muy probable que las funciones usen otros recursos que requieren de la inclusión de archivos cabecera. Por tanto, las funciones aparecerán después de las directivas `#include`.
- La función `main` es la más importante, ya que es la que determina la ejecución del programa. Por ello, normalmente se escribe en primer lugar o en el último.
- Debemos facilitar la lectura, así que el orden que se utilice debería facilitarlo, por ejemplo, escribiendo las funciones desde las más simples a las más generales intentando la cercanía entre funciones relacionadas, o al revés.

Con estas consideraciones, el esquema que podemos seguir en nuestros programas puede ser el siguiente:

```
[Inclusión de archivos cabecera]
[using namespace std;]

[Definición de funciones]

int main()
{
  [Instrucciones]
}
```

Un ejemplo de un programa completo, con funciones asociadas, es el programa que escribe los números primos hasta uno dado, usando las funciones que aparecen en la sección anterior. El programa completo es el siguiente:

Listado 2 — Archivo *primos.cpp*.

```
1  #include <iostream>
2  #include <cmath> // sqrt
3  using namespace std;
4
5  int LeerIntPositivo()
6  {
7     int valor;
8     cout << "Introduzca entero positivo:";
9     cin >> valor;
10    while (valor<1) {
11       cout << "El valor no es positivo. Introduzca otro:";
12       cin >> valor;
13    }
14    return valor;
15 }
16
17 // ____________________________________________________________
18
19 bool EsPrimo (int n)
20 {
21    int raiz= sqrt(n);
22    bool esprimo= true;
23    for (int i=2; i<=raiz && esprimo; ++i)
24       if (n% i==0) // Es divisible
25          esprimo= false;
26    return esprimo;
27 }
28
29 // ____________________________________________________________
30
31 int main()
32 {
33    int n;
34
35    n= LeerIntPositivo();
36
37    for (int numero=2; numero<=n; numero++)
38       if (EsPrimo(numero))
39          cout << numero << endl;
40 }
```

La única restricción que debemos tener en cuenta cuando usamos una función es que C++ obliga a "conocer" una función antes de poder llamarla. Esto tiene mucho sentido, ya que de esta forma podrá comprobar que la llamada es correcta. Así, en el ejemplo anterior, la llamada a `EsPrimo` que se realiza en la función **`main`** —línea 38— aparece después de la definición de la función y, por tanto, el compilador sabe que es una función que toma un entero y devuelve un booleano, dando como válida la llamada que hemos escrito.

Ahora bien, el lector puede preguntarse lo que ocurriría si la función *EsPrimo* se hubiera escrito después de **main**. Efectivamente, el compilador llegaría a la llamada sin conocer aún la definición. En este caso se genera un error de compilación. Para resolver este problema, es decir, para poder hacer una llamada a una función aunque no se tenga su definición[2], el lenguaje permite *declarar* una función. La declaración de una función consiste en especificar la *cabecera* o *prototipo de la función* seguida de un punto y coma:

tipo_función *nombre_función* (parámetros formales) **;**

Con ella, se declara al compilador la *interfaz* con la función. Así indicamos la información que necesita para poder "comprobar" las llamadas a esa función. Si reescribimos nuestro ejemplo, con las definiciones después del **main**, el listado quedaría como sigue:

```
1  #include <iostream>
2  #include <cmath> // sqrt
3  using namespace std;
4
5  // Declaraciones (prototipos)
6  int LeerIntPositivo();
7  bool EsPrimo (int n);
8
9  // ____________________________________________________________
10
11 int main()
12 {
13    int n;
14
15    n= LeerIntPositivo();
16    for (int numero=2; numero<=n; numero++)
17       if (EsPrimo(numero))
18          cout << numero << endl;
19 }
20
21 // ____________________________________________________________
22
23 // Definiciones
24 // ...
```

donde las definiciones —que no cambian— aparecerían al final de ese código. Aunque parezca que esta discusión sobre las cabeceras de las funciones es relevante sólo por el orden de definición de las funciones, más adelante veremos cómo estas declaraciones son fundamentales para poder estructurar nuestro código. Por ejemplo, cuando se incluye un archivo cabecera como **cmath** para poder usar **sqrt**, realmente sólo se incluye la cabecera o prototipo de esta función.

Traza del programa

Para entender el mecanismo de llamada podemos realizar una *traza del programa*, es decir, un estudio paso a paso de las operaciones que el programa va realizando.

Supongamos que estamos en la línea 17 del listado anterior, después de haber leído el valor 100 desde la entrada estándar e inicializado la variable *numero* a 2. Cuando llegamos a la sentencia **if** el programa debe resolver la llamada *EsPrimo(numero)*. En este instante se realiza la transferencia de control a la función, como muestra la figura 5.1.

Al entrar en la función, podemos decir que se deja "aparcado" el trabajo que se estaba realizando en la de llamada. Se hacen corresponder uno a uno los parámetros; en este caso, el valor que se introduce en *numero* lo "recoge" el parámetro *n*. Note que este parámetro es una variable independiente de la función que llama. De hecho, la función que llama tiene su propia *n* —con valor 100— que es independiente del código de la función. Es decir, si modificamos *n* en la función *EsPrimo*, no afectaría en nada a la función que llama.

[2] Esto es precisamente lo que hace al llamar a **sqrt**.

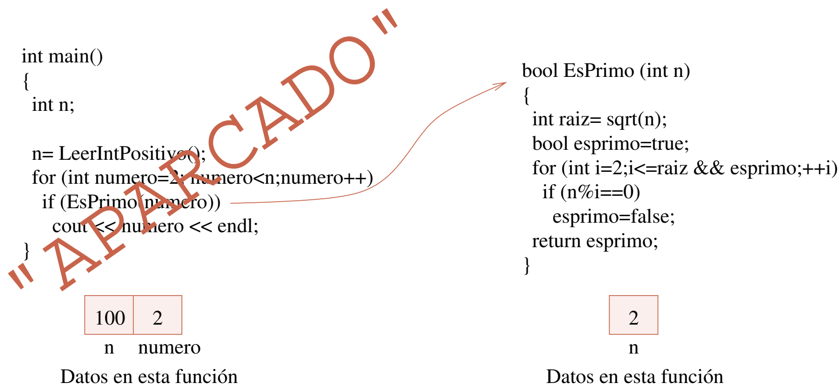


Figura 5.1
Llamada en la primera iteración.

A continuación se ejecuta la función, tal como el lector ya conoce de los temas anteriores, hasta que llega a la sentencia **`return`**. En este punto se conocen tres variables:

- *`n`*: No se ha modificado desde que se introdujo, por tanto sigue valiendo 2.
- *`raiz`*: Se asignó el valor de la raíz cuadrada de *`n`*, es decir, 1 (es un entero).
- *`esprimo`*: Contiene el valor **`true`** ya que se ha calculado que 2 es primo.

En la figura 5.2 se presenta la situación gráficamente. Observe que la variable *`i`* no existe en este punto, ya que su ámbito terminó con el bucle.

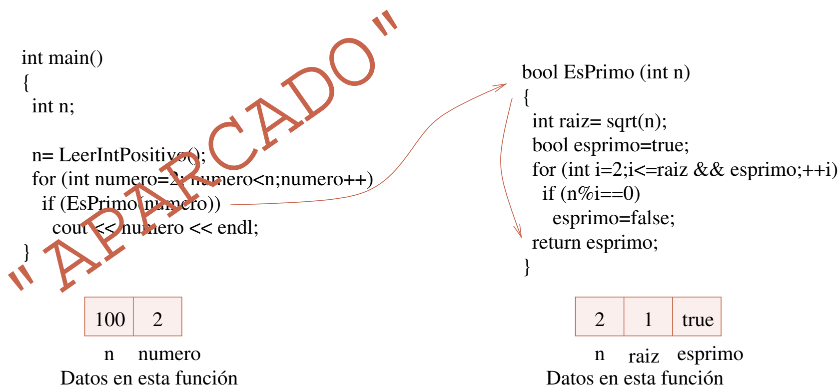


Figura 5.2
Antes de terminar la función.

Finalmente, se devuelve el control a la función **`main`** —con el resultado de **`true`**— momento en el que dejan de existir las tres variables *locales* a la función *`EsPrimo`*.

Si seguimos la traza, el siguiente paso es la escritura del número *2* (ya que la función devolvió **`true`**), se incrementa *`numero`* a 3 y volvemos a entrar en el cuerpo del bucle. Ahora se vuelve a repetir el proceso, aunque el argumento de la función es ahora 3 (en la función, el valor de *`n`* inicial será 3). Por tanto, se hacen tantas llamadas como iteraciones, es decir, 100. Para cada una de ellas, se repite la llamada, creación de *`n`* (con distinto valor cada vez), de los parámetros locales, la destrucción de las variables locales, y la vuelta.

Ejercicio 5.2 Considere el programa de la página 62, donde se calcula el máximo común divisor de dos enteros. Reescriba el programa para incluir una función *MCD* que tenga dos parámetros y devuelva el máximo común divisor de ellos.

Ejercicio 5.3 Escriba un programa que lea un valor entero desde la entrada estándar y escriba el factorial en la salida estándar. Realice la modularización que considere más adecuada.

Ámbito de las variables

Al poder diseñar distintas funciones —no sólo **main**— vuelve a aparecer la discusión sobre el ámbito de las variables. Por ejemplo, en el programa que muestra la figura 5.2, cuando estamos en la función *EsPrimo*, no sólo existen las variables de esta función, sino también las de **main** (recordemos que esta función aún no ha terminado).

En primer lugar, debemos tener en cuenta que *las variables que aparecen en una función no son visibles a ninguna otra función*, aunque se llamen entre ellas. En este sentido, las variables son *locales* a cada función. Además, dentro de cada función, las normas de ámbito son las mismas que hemos visto anteriormente[3]. Así, las variables se crean en el momento de la declaración, y se destruyen cuando se sale del bloque correspondiente.

Por otro lado, aparecen otras declaraciones: los parámetros. En este caso, la situación es la misma que con las variables locales. Lo único relevante es que se crean en el momento de la llamada (como copia de los argumentos de la llamada) y se destruyen cuando se devuelve la llamada. En este sentido, su comportamiento es idéntico a las variables locales.

Por ejemplo, observe que en el ejemplo de la figura 5.2 aparece un parámetro *n*, mientras que en la función **main** también hay una variable local con igual nombre. Las funciones son independientes, así que:

- Si referenciamos *n* en la función *EsPrimo*, nos referimos al parámetro de esta función, que se creó con una copia del valor de *numero* en la llamada.
- Si referenciamos, por ejemplo, *numero* en la función *EsPrimo*, obtendremos un error de compilación, ya que en esta función no existe ningún identificador con ese nombre.

Después de estos comentarios, podrá darse cuenta de que el comportamiento es el más razonable, ya que con la modularización deseamos resolver subproblemas independientes y, por tanto, cuando nos centremos en un módulo podemos ignorar los detalles de otros.

Ejemplo 5.2.2 Considere el programa de cálculo de una raíz de la función $x^3 + x - 1 = 0$ de la página 62, donde se utiliza el algoritmo de bisección para localizar la raíz. Rediseñar el programa mejorando la modularización.

Inicialmente, en este problema podemos distinguir tres módulos que resuelven distintos subproblemas:

- El problema de *evaluar* la función. La función se ocupa de obtener el valor de $f(x)$, dado un número real x.
- El algoritmo de bisección[4]. El problema de obtener una raíz por bisección es independiente de lo que se haga con la solución y de la función (ecuación) que se quiere resolver. Por tanto, podemos plantear una solución genérica que nos sirva para distintos problemas.
- El programa principal: que lee la precisión, resuelve la ecuación y escribe el resultado.

Una solución a este problema puede ser el siguiente programa:

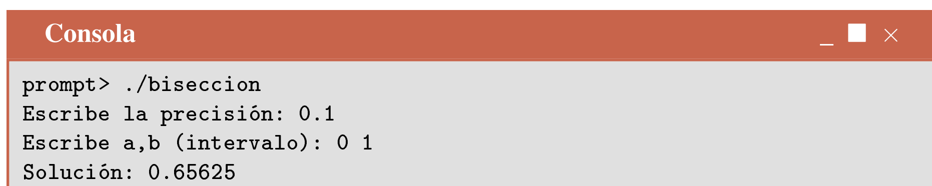
[3] Era de esperar, ya que **main** no es más que una función.

[4] Se podrían usar herramientas que podrían mejorar esta solución. Por ejemplo, haciendo que la función sea un parámetro, aunque está fuera de los objetivos de este tema.

Listado 3 — Archivo *biseccion.cpp*.

```
1  #include <iostream>
2  using namespace std;
3
4  double FuncionF(double x)
5  {
6     return x*x*x + x - 1; // x^3+x-1
7  }
8
9  // ________________________________________________________________
10
11 double Biseccion (double izq, double der, double prec)
12 {
13    while (der-izq>prec) {
14       double centro= (der+izq)/2;
15       if (FuncionF(izq)*FuncionF(centro)<=0)
16          der= centro;
17       else izq= centro;
18    }
19    return (der+izq)/2;
20 }
21
22 // ________________________________________________________________
23
24 int main()
25 {
26    double precision;
27    double a, b;
28
29    cout << "Escribe la precisión: ";
30    cin >> precision;
31    cout << "Escribe a,b (intervalo): ";
32    cin >> a >> b;
33
34    if (FuncionF(a)*FuncionF(b)<0)
35       cout << "Solución: " << Biseccion(a,b,precision) << endl;
36    else cout << "No hay garantía de solución." << endl;
37 }
```

Un ejemplo de ejecución es el siguiente:



Note que este programa, donde dividimos el problema en varios módulos, tiene claras ventajas con respecto al anterior:

- Es más fácil de leer. Se puede entender mejor el objetivo y las partes que componen esta solución.
- Podemos reutilizar la función `Biseccion` en otros problemas.
- La función `FuncionF` encapsula la evaluación. Como puede observar, se utiliza tanto en la función **`main`** como en la función `Biseccion`. Resolver otra ecuación no es más que cambiar este módulo, independientemente del resto del programa.

En este caso, resulta también de interés hacer una traza del programa, por ejemplo, de la ejecución anterior. Supongamos que nos situamos en la línea 35, después de las lecturas, en la llamada a la función `Biseccion`. Además, la precisión leída es 0.1 y el intervalo el $[0,1]$. En ese punto, la situación es la que presenta la figura 5.3.

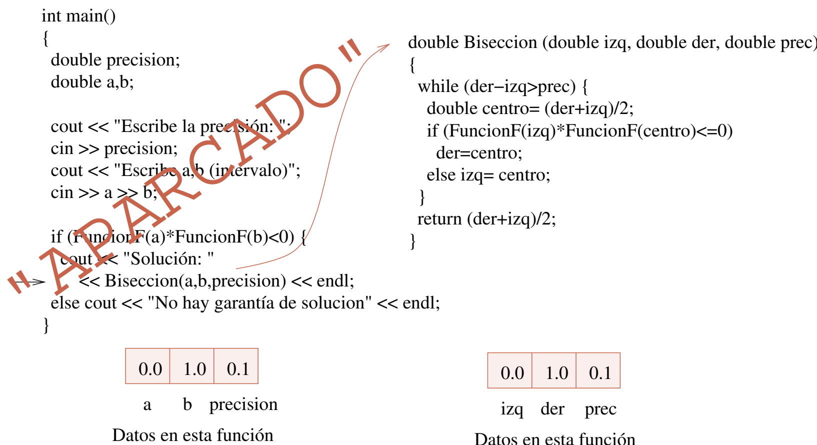


Figura 5.3
Al llamar a *Biseccion*.

En el momento de la llamada se realiza una transferencia de control, desde la función **`main`** a la función `Biseccion`. Es decir, dejamos "aparcada" la función principal y comienza la ejecución de la segunda. Para gestionar la llamada, se establece una correspondencia uno a uno de los argumentos a los parámetros formales. Como vemos,

- el parámetro `izq` se crea con una copia de `a`,
- el parámetro `der` se crea con una copia de `b`,
- el parámetro `prec` se crea con una copia de `precision`.

Observe que la variable local `centro` aún no existe, ya que no hemos entrado al bloque donde se declara. Cuando sigue la ejecución, se evalúa la condición del bucle —que es **`true`**— y se entra definiendo `centro` con el valor 0.5. Al llegar a la sentencia **`if`** existe un nuevo salto a otra función. Esta situación se presenta en la figura 5.4.

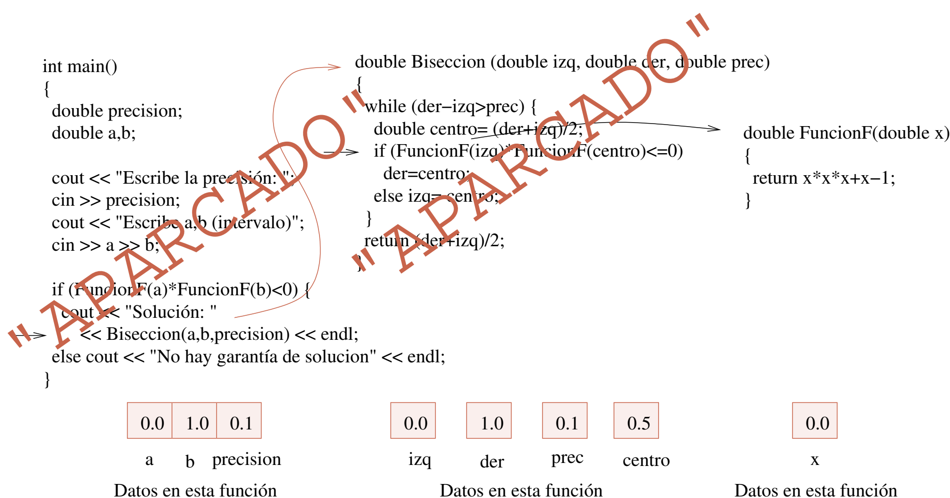


Figura 5.4
Al llamar a *funcion*.

Ahora tenemos dos funciones "aparcadas", la primera **main** y la segunda `Biseccion`, estando situados en la función `FuncionF`. Observe que, en esta figura, ya sí aparece la variable `centro`.

La llamada a la función se realiza de nuevo haciendo corresponder cada parámetro. Así, como sólo tenemos el valor `x`, éste se creará como una copia de `izq` (con valor 0.0). En la situación actual existen tres *contextos* diferentes:

- Contexto de **main**. Se conocen las variables `a`, `b` y `precision`. Está a la espera de volver de la llamada a `Biseccion`.
- Contexto de `Biseccion`. Se conocen las variables `izq`, `der` y `prec`. Está a la espera de volver de la llamada a `FuncionF`.
- Contexto de `FuncionF`. Se conoce la variable `x`.

Note que podemos tener programas que sigan "enlazando" llamadas de una función a otra de forma, en principio, indefinida. Cada vez que se llama a una función, la anterior se queda "aparcada", para volver a ella cuando se haya terminado. Así, cuando una función termina, volvemos al punto de llamada de la última función "aparcada". Esta situación se presenta en la figura 5.5.

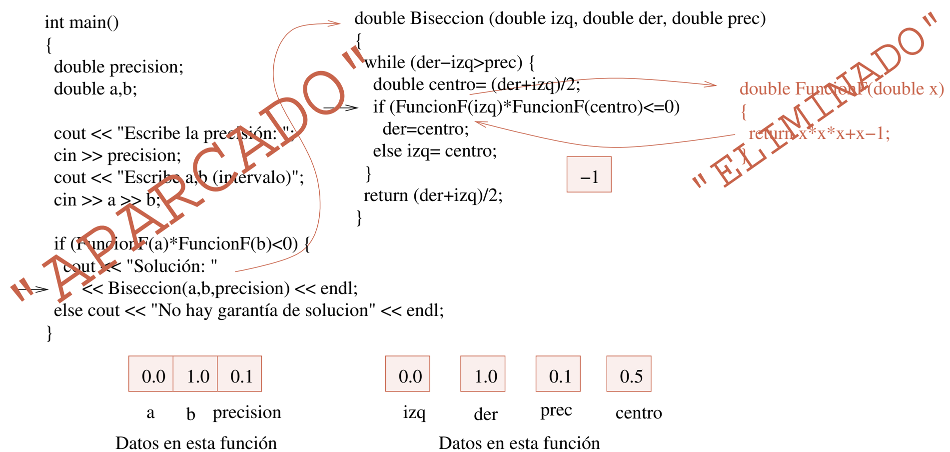


Figura 5.5
A la vuelta de *funcion*.

Cuando volvemos de la última llamada, devolvemos el valor indicado en la sentencia **return**, es decir, -1. Por tanto, la función termina y el contexto asociado a ella se "elimina". En este punto se ha resuelto la primera llamada a `FuncionF` y antes de la segunda —con el argumento `centro`— volvemos a tener sólo dos contextos.

Fin ejemplo 5.2.2 ■

Ejercicio 5.4 Modifique el programa anterior para que resuelva la ecuación $5x^3 + 3x - 1 = 0$.

5.2.2 Funciones *void*

Hasta ahora, todas las funciones se han diseñado para tomar un conjunto de entradas —incluido el caso de ninguna— y devolver un resultado como salida. Sin embargo, existen muchas situaciones en las que el efecto de una función no es devolver un valor. Por ejemplo, imagine que una función se encarga de escribir un resultado en la salida estándar, en la impresora, o simplemente se encarga de rebobinar una cinta. En estos casos, tal vez no nos interese devolver ningún valor.

Para implementar estas funciones, el lenguaje C++ ofrece el tipo **void**, que podemos interpretar como un *"no tipo"*. La sintaxis de uso es similar a la de un tipo simple, pero en realidad indica que no se especifica tipo. De hecho, no podemos declarar una variable de tipo **void**.

Cuando el tipo de una función es **void** se está especificando que no devuelve nada[5]. En este caso, no es necesario escribir la sentencia **return**. La función acaba cuando se llegue al final de ella[6].

Como muestra, podemos crear un módulo adicional, para el ejemplo anterior, con el objetivo de escribir en la salida estándar el resultado de los cálculos. El código puede ser:

```
void Resultados (double a, double b, double p, double res)
{
   cout << "Solución:" << endl;
   cout << "\tIntervalo: " << '[' << a << ',' << b << ']' << endl;
   cout << "\tPrecision: " << p << endl;
   cout << "\tRaíz: " << res;
   cout << " ("<< res-p << ',' << res+p << ")." <<endl;
}
```

y la función **main** cambiaría de la siguiente forma:

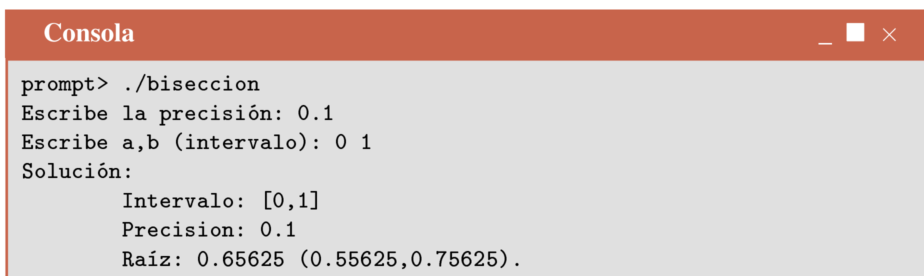
```
1  int main()
2  {
3     double precision;
4     double a, b;
5
6     cout << "Escribe la precisión: ";
7     cin >> precision;
8     cout << "Escribe a,b (intervalo): ";
9     cin >> a >> b;
10
11    if (FuncionF(a)*FuncionF(b)<0)
12       Resultados(a, b, precision, Biseccion(a,b,precision) );
13    else cout << "No hay garantía de solución." << endl;
14 }
```

donde podemos ver que la llamada a `Biseccion` se ha especificado como parámetro de `Resultados`
.

Cuando lleguemos a esa línea —la número 12– el flujo de control saltará a la función `Biseccion` para devolver la raíz. Ese valor será el que se use para crear el parámetro `res` en la llamada a `Resultados`.

Observe además, que no existe ningún valor devuelto por la función `Resultados`, por lo que la llamada a la función se escribe como una sentencia independiente. Lógicamente, la llamada no puede ser parte de una expresión, ya que no devuelve nada.

Un ejemplo de la ejecución de este programa es el siguiente:



```
Consola
prompt> ./biseccion
Escribe la precisión: 0.1
Escribe a,b (intervalo): 0 1
Solución:
        Intervalo: [0,1]
        Precision: 0.1
        Raíz: 0.65625 (0.55625,0.75625).
```

[5]En algunos lenguajes, a este tipo de módulos también se denominan *procedimientos*, incluso utilizando la palabra *procedure* en su sintaxis.

[6]Véase más adelante comentarios adicionales sobre **return** en la página 102.

Ejercicio 5.5 Analice la condición de parada del bucle en la función *Biseccion*. Realmente, damos una iteración de más, puesto que escoger el valor central garantiza que la precisión es la mitad del tamaño del intervalo. Modifique el código para ahorrarnos esta iteración.

5.2.3 Paso por valor y por referencia

Podemos modificar el programa anterior con un nuevo módulo que pida los valores de entrada. Por un lado, podemos añadir uno para pedir un valor **double** mayor que cero que corresponde a la precisión y, por otro lado, un módulo para pedir el intervalo donde buscar. Sin embargo, una función sólo devuelve un valor: ¿Cómo podemos hacer que devuelva *a* y *b* (el intervalo)? Para resolverlo podemos utilizar el *paso por referencia*.

En los ejemplos que hemos visto hasta ahora, todos los parámetros de la función eran de entrada. Es decir, pasábamos el valor —que se copiaba en el parámetro formal— para que se usara en la función. Este tipo de paso se denomina *paso por valor*.

Es posible indicar que un parámetro es de salida (o de entrada y salida) mediante el paso por referencia. La forma de especificarlo es añadiendo el carácter & después del tipo. En este caso, el parámetro se dice que es una referencia a ese tipo base. Por ejemplo, podemos escribir:

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

void Funcion(int& var)
{
   var= 1;
}

int main()
{
   int a= 0;
   Funcion(a);
   cout << a;
}
```

donde especificamos que *var* es una referencia[7] a **int**. El comportamiento del paso por referencia hace que el parámetro no se cree como una copia del valor que se pasa, sino que sea un sinónimo del objeto que se pasa. Esto implica que se refiere al mismo objeto.

En nuestro ejemplo, si el parámetro *var* se pasara por valor, se hubiera creado como una copia del argumento *a*. Es decir, si se modifica el parámetro en la función, al volver a **main** el valor de *a* seguiría siendo cero. Sin embargo, con el paso por referencia que hemos especificado, el parámetro *var* no se crea como copia de *a* sino que se refiere al mismo objeto, es decir, si modifico *var* estoy modificando *a*. En este caso, el programa escribe un uno. Por tanto, con el paso por referencia, modificar el parámetro implica modificar el valor original —argumento— que se le pasó.

Para resolver nuestro problema de leer un intervalo podemos definir una función con la siguiente cabecera:

```
void LeerIntervalo (double& izq, double& der)
```

donde especificamos que tanto *a* como *b* son referencias a **double**. Por tanto, cuando modifiquemos estos parámetros dentro de la función, estaremos modificando los argumentos originales. La función podría ser algo así:

```
void LeerIntervalo (double& izq, double& der)
{
   cout << "Introduzca los dos valores del intervalo: ";
   cin >> izq >> der;
   while (izq > der) {
      cout << "No es correcto. Nuevo intervalo:";
      cin >> izq >> der;
```

[7]Más adelante se estudiará en profundidad el tipo referencia.

```
    }
}
```

Ejercicio 5.6 Modifique la función anterior para que también garantice que el valor de la función en los extremos del intervalo tiene distinto signo.

Como vemos, hemos declarado la función de tipo `void`, ya que "no devuelve" ningún valor. De hecho, no es necesario indicar la sentencia `return` (la función acaba cuando termina el cuerpo, después del bucle). Sin embargo, esta función se ha diseñado para que tenga dos valores `double` de salida.

Consideremos la función `main` modificada:

```
int main()
{
   double precision;
   double a, b;

   cout << "Escribe la precisión: ";
   cin >> precision;

   LeerIntervalo(a, b);

   if (FuncionF(a)*FuncionF(b) < 0)
      Resultados(a, b, precision, Biseccion(a,b,precision) );
   else cout << "No hay garantía de solución." << endl;
}
```

donde se llama a `LeerIntervalo`. En esta llamada, como hemos visto, se hace una correspondencia uno a uno de los argumentos —`a` y `b`— con los parámetro formales `izq` y `der`. Así, `izq` se refiere a `a` (si modificamos `izq` en la función, estamos modificando `a`) y `der` se refiere a `b` (si modificamos `der` en la función, estamos modificando `b`). Después de la llamada, `a` y `b` contendrán los valores del intervalo, es decir, la salida de la función será la pareja de valores `a,b`.

Ejemplo 5.2.3 Implementar una función para intercambiar el contenido de dos variables de tipo `char`.

En muchos casos se necesita intercambiar el valor de dos variables. El algoritmo es muy simple, ya que sólo es necesario disponer de una variable auxiliar para, tras tres asignaciones, intercambiar los valores. Resulta una buena idea disponer de un módulo que se encargue de estos detalles. La función puede ser:

```
void Intercambiar (char& v1, char& v2)
{
   char aux= v1;
   v1= v2;
   v2= aux;
}
```

donde vemos que `v1` y `v2` son datos tanto de entrada (contienen los valores a asignar) como de salida (se les asignan nuevos valores).

Fin ejemplo 5.2.3 ■

Ejemplo 5.2.4 Modificar la función `LeerIntervalo` para que lea los valores y los intercambie en caso de estar desordenados.

La solución puede ser:

```
void LeerIntervalo (double& izq, double& der)
{
   cout << "Introduzca los dos valores del intervalo: ";
   cin >> izq >> der;
   if (izq>=der) {
```

```
        double aux= izq;
        izq= der;
        der= aux;
      }
    }
```

aunque también podríamos diseñar un módulo que intercambie los valores **double**. Por ejemplo, podemos hacer:

```
void CambiarIntervalo (double& v1, double& v2)
{
   double aux= v1;
   v1= v2;
   v2= aux;
}

void LeerIntervalo (double& izq, double& der)
{
   cout << "Introduzca los dos valores del intervalo: ";
   cin >> izq >> der;
   if (izq > der)
      CambiarIntervalo(izq, der);
}
```

En este caso es interesante comentar las llamadas que se producen en el programa, ya que cuando hacemos la primera, los parámetros `izq` y `der` se hacen corresponder con los originales `a` y `b`. A su vez, cuando hacemos la llamada a `CambiarIntervalo`, `v1` y `v2` se hacen corresponder con `izq` y `der` y por lo tanto con `a` y `b`. Así, al modificar sus valores, realmente se modifican los dos originales de la función **main**.

A pesar de ello, es importante insistir en que el diseño de funciones no implica tener en cuenta simultáneamente los detalles de distintos módulos. En este ejemplo, cuando diseñamos `CambiarIntervalo` no tenemos que pensar en los detalles de los módulos que llaman a la función.

Igualmente, si diseñamos `LeerIntervalo` no pensamos en otros detalles. Por ejemplo, en la llamada a `CambiarIntervalo` sólo tenemos en cuenta que queremos intercambiar `izq` y `der` y que esa función lo hace, independientemente de que dichas variables tengan alguna relación con otro módulo o no.

Fin ejemplo 5.2.4 ■

Ejercicio 5.7 Desarrolle una función que tenga como entrada dos enteros correspondientes a una fracción (numerador y denominador) y como salida modifique sus valores para hacerla irreducible. Nota: Recuerde que, para simplificar la fracción, se dividen ambos por el máximo común divisor (una función que ya tenemos resuelta).

Ejemplo 5.2.5 Desarrolle un programa para leer una ecuación de segundo grado y escribir la solución. Tenga en cuenta todos los casos de error.

Una función que resuelve una ecuación de segundo grado devuelve, al menos, las dos soluciones. Por tanto, podemos diseñarla usando paso por referencia. A la función tenemos que pasarle, como entrada, los tres valores que definen la ecuación (pasados por valor). Además, debe devolver dos valores. El lenguaje nos permite escribir algo así:

```
double Ecuacion2Grado (double a, double b, double c, double& x1)
```

Observe que el último parámetro es por referencia, donde devolvemos la primera solución, y la función es de tipo **double** para devolver la segunda. Sin embargo, esta solución resulta poco aconsejable, ya que es muy confusa al devolver resultados simultáneamente de ambas formas. Resulta mucho más claro lo siguiente:

```
void Ecuacion2Grado (double a, double b, double c,
                     double& x1, double& x2)
```

por lo que es probable que diseñemos funciones **void** con varias salidas. A pesar de ello, también es posible devolver algún valor destacado como resultado de la función. Por ejemplo, podemos devolver un valor indicando si la función ha tenido éxito o no. Resulta cómodo poder usar ese valor de vuelta en una condición (sentencia **if**).

En la solución propuesta, hemos incluido un valor **bool** que indica si existe solución. El programa puede ser el siguiente:

Listado 4 — Archivo *ecuacion2grado.cpp*.

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  bool Ecuacion2Grado (double a, double b, double c,
6                       double& x1, double& x2)
7  {
8     bool exito;
9
10    if (a==0)
11       if (b==0)
12          exito= false; // No hay x
13       else {             // primer grado
14          x1= x2= -c/b;
15          exito= true;
16       }
17    else {
18      double discriminante= b*b - 4*a*c;
19      if (discriminante>0) {  // Segundo grado
20         x1= (-b+sqrt(discriminante)) / (2*a);
21         x2= (-b-sqrt(discriminante)) / (2*a);
22         exito= true;
23      }
24      else if (discriminante<0)
25              exito= false; // no hay solución
26           else{ // Una solución
27              x1= x2= -b / (2*a);
28              exito= true;
29           }
30    }
31    return exito;
32 }
33
34 void Resultados2Grado (double a, double b, double c,
35                        double x1, double x2)
36 {
37    cout << "Ecuación con coeficientes: "
38         << a << ',' << b << ',' << c << endl;
39    if (x1==x2)
40       cout << "Una solución: " << x1 << endl;
41    else cout << "Dos soluciones: " << x1 << ", " << x2 << endl;
42 }
43
44 int main()
45 {
46    double a, b, c, s1, s2;
47
48    cout << "Introduzca tres números: ";
49    cin >> a >> b >> c;
50    if ( Ecuacion2Grado(a, b, c, s1, s2) )
51       Resultados2Grado(a, b, c, s1, s2);
52    else cout << "Esta ecuación no tiene soluciones." << endl;
53 }
```

donde usamos el valor devuelto por la función *Ecuacion2Grado* en una sentencia condicional, escribiendo un mensaje de error en caso de que no haya soluciones. Algunos ejemplos de su uso son:

Consola

```
prompt> ./ecuacion2grado
Introduzca tres números: 1 1 1
Esta ecuación no tiene soluciones.
prompt> ./ecuacion2grado
Introduzca tres números: 1 2 1
Ecuación con coeficientes: 1,2,1
Una solución: -1
prompt> ./ecuacion2grado
Introduzca tres números: 2 -7 3
Ecuación con coeficientes: 2,-7,3
Dos soluciones: 3, 0.5
```

En la función **main** se declaran 5 variables con nombres *a*, *b*, *c*, *s1* y *s2*. Como sabemos, al hacer la llamada, los argumentos se hacen corresponder con los parámetros, uno a uno. Así, el argumento *s1* pasa por referencia a *x1*, es decir, cuando modificamos *x1* dentro de la función, realmente modificamos el original *s1* (de una forma similar para *x2* y *s2*). Téngase en cuenta, en cualquier caso, que los identificadores de **main** son desconocidos en *Ecuacion2Grado*, y los de esta segunda, desconocidos en la primera.

Finalmente, note que una solución que incluya otro parámetro pasado por referencia, para devolver si ha habido éxito, sería correcta. En este caso, la función se haría de tipo **void**, aunque resultaría menos cómoda al no poder usar el resultado directamente en una expresión, en particular, en la condición de una sentencia **if**.

Fin ejemplo 5.2.5 ■

Ejercicio 5.8 ¿Cual sería el efecto de hacer la llamada como `Ecuacion2Grado(a,b,c,a,b)`?

Ejercicio 5.9 En los algoritmos de cálculo numérico resulta “delicada” la comparación de números reales con un valor concreto (véase problema 2.6). En el ejemplo anterior, realizamos algunas operaciones dependiendo de si un valor real vale cero. Sin embargo, en la práctica los cálculos de número reales son aproximados ya que no podemos representar todos los valores[a]. Modifique la función *Ecuacion2Grado* para incorporar esta posibilidad. Considere que un valor es igual a otro si su diferencia es menor que un cierto épsilon.

[a]Este problema resulta fundamental en muchos algoritmos. Especialmente, cuando un valor se obtiene tras muchas operaciones en las que se van acumulando los errores de aproximación.

5.2.4 Función *main*. Finalización del programa

El módulo **main**, que contiene las sentencias que el programa debe realizar, no es más que otra función. La única diferencia es que contiene un nombre especial —**main**— por lo que se considera la función principal. Así, la ejecución del programa consiste en transferir el control a esta función. El programa finaliza, por tanto, cuando lo hace esta función[8].

[8]Realmente se pueden ejecutar sentencias antes y después de **main**, aunque es un tema más avanzado que no abordaremos ahora mismo.

La función **main** tiene una cabecera que indica que es de tipo **int**, es decir, que devuelve un entero al finalizar. Este valor indica si el programa ha acabado con éxito o no. Por ejemplo, piense en la posibilidad de que un programa pueda hacer que se ejecute otro para realizar alguna acción previa. Lógicamente, es recomendable tener una forma de comunicar al programa inicial si ha habido algún problema (por ejemplo, no existe algún fichero de entrada, falta espacio en disco o en memoria principal, etc.). Un valor de cero indica finalización con éxito, y distinto de cero que se ha producido algún error (distintos valores podrían indicar distintos tipos de error).

El hecho de que sea una función de tipo **int** debería hacernos pensar que es necesaria la sentencia **return**. Sin embargo, el lenguaje permite prescindir de ella para el caso de **main**. Si no se especifica ninguna, se entiende que al llegar al final el programa devuelve un cero (acaba con éxito). Es la única función en la que podemos hacer esto, es decir, prescindir de **return** sin ser una función de tipo **void**.

Por otro lado, es necesario indicar que el programa también puede terminar de otras formas. Por ejemplo, puede terminar directamente con una llamada a la función **exit** —declarada en el fichero cabecera **cstdlib**— en cualquier punto del programa[9]. Como parámetro, se especifica un valor entero que se devuelve como resultado del programa, es decir, el correspondiente a la devolución de **main**. Estas formas de acabar serán poco habituales, ya que normalmente haremos que nuestros programas terminen mediante la finalización de la función **main**. A pesar de ello, encontrará muchos códigos en los que se usa esta función para simplificar la finalización del programa, una vez que no queda nada por hacer y no queremos preocuparnos de devolver el flujo de control hasta la función **main**.

5.3 Diseño de funciones

Un aspecto fundamental relacionado con las funciones se refiere a la calidad de la solución escogida en el diseño modular. En las secciones anteriores hemos solucionado algunos problemas con un tamaño relativamente pequeño, con menos de un centenar de líneas. A pesar de ello, ya surgen distintas alternativas en el diseño, más o menos módulos, con unos u otros parámetros.

Lógicamente, en problemas de gran tamaño, las soluciones pueden llegar a ser muy distintas. Cuanto mejor sea el analista/programador, mejor será la solución. En nuestro enfoque, no aceptaremos cualquier solución como buena, aunque se comporte correctamente —“funcione”— para todas las entradas, sino que buscaremos soluciones que faciliten la lectura, la modificación, la reusabilidad del programa. Es decir, será un enfoque basado en la calidad de la solución, ya que al haber distintas soluciones igualmente válidas, deberemos seleccionar la que consideremos mejor (véase la sección C.3 de la página 330 para una discusión más extensa.)

Pensemos, por ejemplo, que para resolver un problema disponemos de una solución de 500 líneas consecutivas en la función **main**. Es de esperar que la calidad de esta solución sea realmente baja, ya que será difícil de leer, modificar, mantener o reusar. Es difícil estudiar lo que hace la línea 300 si no tenemos una idea de las cosas que se han hecho en las 300 líneas anteriores y que se harán en las siguientes. Recuerde que, en general, el ser humano no es capaz de retener fácilmente muchos detalles.

Una primera solución podría ser crear distintos módulos para que no tengan más de, por ejemplo, 20 líneas. Con este objetivo, podemos realizar una simple operación: cogemos las 500 líneas y las modularizamos cada 20. Es decir, obtenemos los módulos *modulo1, modulo2,...,modulo25* que contienen las distintas partes. Piense un momento en esta solución, sería incluso más difícil de entender que la anterior. Por tanto, no modularizamos simplemente por hacerlo, sino para simplificar el desarrollo del software.

[9]Realmente, el programa también puede acabar al lanzar una excepción o llamando a la función **abort**, aunque su comportamiento es distinto.

Por otro lado, podemos pensar que cuanto mayor sea el número de módulos, mejor será la solución, ya que está más modularizado. En la práctica, esta relación no tiene que ser así, ya que una vez que llegamos a una buena solución, el crear nuevos módulos puede empeorarla.

Para obtener un diseño modular efectivo, es necesario considerar una división en subproblemas efectiva. Si queremos simplificar el desarrollo, lo mejor es hacer que los distintos módulos sean lo más independientes que podamos (por ejemplo, así podemos desarrollarlos de forma independiente), así como que sus interacciones sean mínimas (por ejemplo, así es más fácil integrarlos). Recordemos el ejemplo inicial de esta tema, si diseñamos una CPU es mejor tener una interfaz lo más simple posible para conectarlo al resto de componentes. El hecho de que un módulo oculte sus detalles internos se denomina *ocultamiento de información*. Es un concepto fundamental[10] en el desarrollo de sistemas complejos. Lo encontrará repetidamente conforme avance en la programación de ordenadores.

Por otro lado, es recomendable que los módulos realicen tareas concretas y no varias tareas que podrían resolverse de forma independiente. Por ejemplo, imagine que queremos un programa que lea un entero y escriba su factorial; podríamos diseñar un módulo que lleva a cabo la lectura y los cálculos como el siguiente:

```
int Factorial()
{
   int n;
   cin >> n;
   int resultado= 1;
   for (int i=2; i<n; ++i)
      resultado*= i;
   return resultado;
}
```

de manera que la función `main` consistiría sólo en escribir el resultado de este módulo. Sin embargo, es muy poco efectivo. Piense que si otro programa necesita calcular el factorial no puede usar esta función. Si hubiéramos dividido de forma que un módulo se ocupa únicamente de hacer el cálculo:

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

sería más sencilla su reutilización. En este sentido, es poco efectivo escribir funciones que realicen operaciones de E/S y cálculos. Es recomendable que lleven a cabo una tarea bien definida. Para una discusión más profunda sobre descomposición modular, véase página 339 o Pressman[22].

Ejercicio 5.10 Considere las ecuaciones:

$$\begin{array}{lll} x_0 = \sqrt{2} & y_0 = 0 & \pi_0 = 2+\sqrt{2} \\ x_{n+1} = \frac{1}{2}\left(\sqrt{x_n} + \frac{1}{\sqrt{x_n}}\right) & y_{n+1} = \frac{(1+y_n)\sqrt{x_n}}{x_n+y_n} & \pi_n = \pi_{n-1}\cdot y_n \cdot \frac{x_n+1}{y_n+1} \end{array} \quad (5.1)$$

presentadas anteriormente (véase el problema 4.13), como un método iterativo para el cálculo aproximado del número π. Escriba un programa para leer un valor n, y escribir el correspondiente π_n. Para ello, tenga en cuenta módulos para leer el valor de entrada, inicializar las tres variables involucradas en el método iterativo, realizar una iteración y calcular π_n.

[10] Más adelante insistiremos sobre él. Véase por ejemplo C.7.2 en la página 339.

Ejercicio 5.11 Desarrolle un programa que lea tres números reales, los ordene y los escriba en pantalla.

5.3.1 Variables globales

El lenguaje C++ permite la declaración de variables fuera de las funciones. El ámbito de estas variables abarca desde su declaración hasta el final del archivo. Además, estas variables existen desde que comienza el programa hasta que termina. Un ejemplo con una variable global es:

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

char global;

void Cambiar()
{
   global= 'B';
}

void Escribir()
{
   cout << global << endl;
}

int main()
{
   global= 'A';
   Escribir();
   Cambiar();
   Escribir();
}
```

que escribe los caracteres 'A' y 'B' en la salida estándar. Para conocer el efecto de la última llamada a `Escribir` es necesario tener en cuenta, no sólo esta función, sino también las que tienen acceso a la variable global (en este caso, todas).

Una declaración puede aparecer en cualquier punto del archivo C++, incluso antes de los archivos cabecera o entre las funciones. Como queremos que todas las funciones tengan acceso a ellas, es habitual encontrarlas declaradas después de los archivos cabecera. En este caso, el esquema de un programa con variables globales podría ser:

```
[Inclusión de archivos cabecera]
using namespace std;
[Definición de variables globales]
[Definición de funciones]
int main()
{
  [Instrucciones]
}
```

Sin embargo, recordemos que una buena modularización implica disminuir las relaciones entre los módulos. Si nos fijamos en las consecuencias de declarar una variable global, es fácil entender que establece una fuerte relación entre los módulos que la comparten. De hecho, podemos considerarla excesiva.

Tenga en cuenta que garantizar que una función es correcta puede ser más complicado, ya que si usa el valor que contiene la variable global, deberá tener en cuenta lo que hacen el resto de las funciones. Si otra función falla y modifica incorrectamente el valor de la variable, nuestra función fallará a pesar de no contener ningún error. Recuerde la ventaja de implementar una función ignorando el resto del programa. Ahora, esta ventaja ha desaparecido.

Para evitar esta situación, sólo se definirán cuando realmente se pueda justificar su necesidad. Por ello, nosotros no llegaremos a usar variables globales en nuestras soluciones.

Constantes globales

Cuando declaramos una variable global podemos usar la palabra reservada **`const`** para indicar que es constante. En este caso hablamos de *constantes globales*.

Ahora no se presentan los problemas derivados de una modificación incorrecta de la variable, ya que no es posible modificarla. Así, podemos usarlas sin miedo a provocar efectos colaterales. Note que una vez declarada la constante con cierto valor ningún módulo la puede modificar, por lo tanto, no existe ninguna relación entre un módulo y otro. La constante siempre vale lo mismo, independientemente de las acciones que se ejecuten.

Por otro lado, su uso puede ser muy recomendable cuando tengamos que usar la misma constante para distintos subproblemas. Por ejemplo, imagine que tenemos varias funciones para realizar cálculos con ángulos (por ejemplo, los vistos sobre polígonos regulares, círculos, etc.). Tiene poco sentido tener que definir una constante `PI` para cada uno de los módulos. La mejor solución será definir una constante global, al comienzo del archivo, que defina este valor para todas las funciones.

De esta forma, será más sencillo el desarrollo y el mantenimiento. Por ejemplo, con una simple modificación puedo cambiar el valor de la constante para todos los módulos.

Ámbito. Operador de resolución de ámbito

En la sección 4.6 hemos presentado el concepto de *ámbito* como el trozo de código donde se conoce la variable, así como el *ocultamiento de nombres*, producido cuando se repite un nombre de variable en un ámbito más local.

Con las variables globales aparece un nuevo *ámbito*, que va desde su declaración hasta el final del archivo. Por tanto, este *ámbito global* puede contener nombres que queden ocultos con declaraciones locales. Por ejemplo, podemos provocar esta situación con el siguiente código:

```
char v;

void Funcion()
{
   int v; // oculta v global
   v= 1;   // usa local
   {
      int v; // oculta v local
      v= 2;   // usa local
   }
   // v sigue valiendo 1
}
```

donde aparecen tres ámbitos distintos, con sus correspondientes variables `v`.

Los nombres de ámbito global se pueden referenciar utilizando el operador de resolución de ámbito `::` delante del identificador. Así, en cualquier punto de la función anterior, podríamos haber usado `::v` para referenciar el nombre `v` de la variable global de tipo **`char`**.

En cualquier caso, la recomendación es no usar los mismos nombres para evitar la ocultación de nombres.

5.3.2 Funciones y programación estructurada

En capítulos anteriores hemos presentado la programación estructurada. Ésta se basa en las estructuras secuencial, condicional y de iteración, evitando el uso de saltos incondicionales. En este capítulo, presentamos la modularización como una forma ideal para enfrentarse a problemas complejos, pero que requiere, para cada llamada, un salto al código de la función, y una vuelta al punto de llamada. En este sentido, ¿no se rompe esa forma de programación?

El término programación estructurada incluye la programación modular. Note que una solución modular podría "expandirse" si sustituimos las llamadas por el código que corresponde a la función.

Así, los módulos son una forma de disponer en capas o niveles, los algoritmos que implementan la solución (véase figura 5.6).

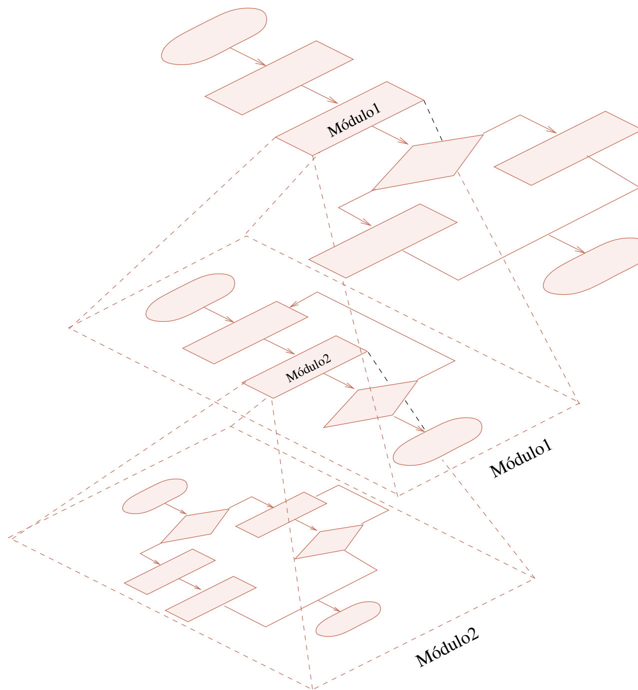


Figura 5.6
Niveles en la programación estructurada.

Por lo tanto, la programación estructurada contiene la modularización como una herramienta válida, siempre que los módulos (funciones) tengan un único punto de entrada y salida (es decir, correspondan a una solución estructurada).

Posición de la sentencia *return*

El hecho de que las funciones tengan un único punto de entrada y salida nos indica que la función comienza con la primera sentencia de su cuerpo y termina con una única sentencia **return** al final (o la última sentencia en las funciones **void**).

Sin embargo, el lenguaje C++ establece que una función acaba en el momento de ejecutar la sentencia **return**, es decir, si la usamos "en medio" de una función, no se ejecutará el resto. Esta situación está en contra de la programación estructurada (se podría incluso considerar un salto al final de la función). Ya que intentamos crear buenos programas, intentaremos evitar estas situaciones.

Veamos un ejemplo, de la función del algoritmo de bisección que hemos visto anteriormente. La función que tenemos es:

```
double Biseccion (double izq, double der, double prec)
{
   while (der-izq>prec) {
      double centro= (der+izq)/2;
      if (FuncionF(izq)*FuncionF(centro) <= 0)
         der= centro;
      else izq= centro;
   }
   return (der+izq) / 2;
}
```

donde hay un punto de entrada y salida. Ahora bien, podemos pensar en la posibilidad de comprobar si la función es exactamente cero en el centro. Por supuesto, si es cero, no es necesario seguir iterando. Se podría modificar de la siguiente forma[11]:

```
double Biseccion (double izq, double der, double prec)
{
   while (der-izq>prec) {
      double centro= (der+izq)/2;
      if (FuncionF(centro) == 0)
          // ¿Qué hacemos?
      else if (FuncionF(izq)*FuncionF(centro) < 0)
              der= centro;
           else izq= centro;
   }
   return (der+izq)/2;
}
```

Si queremos una solución que no rompa la programación estructurada debemos hacer que el bucle termine. Para ello, con la ayuda de una variable booleana podemos solucionarlo así:

```
double Biseccion (double izq, double der, double prec)
{
   bool fin= false;
   while (der-izq>prec && !fin) {
      double centro= (der+izq)/2;
      if (FuncionF(centro) == 0)
         fin= true;
      else if (FuncionF(izq)*FuncionF(centro) < 0)
              der= centro;
           else izq= centro;
   }
   return (der+izq)/2;
}
```

aunque también podríamos haberlo solucionado interrumpiendo la función directamente, con la devolución del valor `centro`, de la siguiente forma:

```
double Biseccion (double izq, double der, double prec)
{
   while (der-izq>prec) {
      double centro= (der+izq)/2;
      if (Funcion(centro) == 0)
         return centro;
      else if (Funcion(izq)*Funcion(centro) < 0)
              der= centro;
           else izq= centro;
   }
   return (der+izq)/2;
}
```

La mejor solución, en principio y según la programación estructurada, es la primera. Sin embargo, para funciones simples donde sea "controlable" el punto de interrupción y en el caso de que la solución sea más sencilla, es aceptable y no se debe considerar una ruptura con la programación estructurada.

Por otro lado, en las funciones **`void`** no hemos usado la sentencia **`return`**, ya que no es necesario especificar un valor de vuelta. Aún así, podemos usar la sentencia **`return`** —seguida directamente de un punto y coma (sin expresión)— para que una función **`void`** termine.

Por último, es importante tener en cuenta que el uso de sentencias **`return`** en posiciones intermedias puede resultar en una solución incorrecta. Podemos encontrar un camino en el flujo que lleve al final de la función sin haber encontrado ninguna sentencia **`return`** y por tanto sin especificar lo que se devuelve. Por ejemplo, considere el siguiente ejemplo:

```
char Funcion (int a)
{
   if (a>0)
      return 'P';
}
```

[11] Podríamos usar una variable auxiliar para almacenar `Funcion(centro)`.

En el caso de que el parámetro no sea positivo, la función alcanza el final sin indicar —con la sentencia `return`— el valor a devolver. Afortunadamente, los compiladores suelen avisar de este tipo de errores.

5.3.3 Especificación de funciones

Para que el diseño modular sea efectivo es necesario que haya ocultamiento de información, es decir, los detalles de un módulo deben quedar ocultos al resto. El resto del programa sólo debe conocer lo mínimo para poder usar la funcionalidad del módulo. Por consiguiente, es necesario especificar la interfaz de las funciones, es decir, indicar la forma de llamar a una función y los efectos que tiene la llamada. Así, para llamar a una función sólo debemos tener en cuenta la especificación, sin necesidad de saber nada sobre los detalles internos que contiene.

La especificación de una función corresponde a un documento que nos permite conocer los aspectos sintácticos y semánticos, necesarios para poder usar la función, sin necesidad de conocer los detalles de su implementación.

La parte sintáctica se refiere a la forma correcta de escribir una llamada a la función. Para ello, será necesario establecer el nombre, así como el número, orden, y tipo de entradas/salidas. La forma más simple, y por tanto más habitual de realizarlo, es utilizando directamente la sintaxis del lenguaje de programación. En nuestro caso, usando la cabecera de la función, la sintaxis queda especificada sin ningún tipo de ambigüedad.

Por otro lado, la parte semántica se refiere al "significado", es decir, a las condiciones que se dan antes y después de haber realizado una llamada a la función. Existen distintas formas de realizarla. En cualquier caso, debe ser una forma de establecer el comportamiento de la función independientemente de los detalles internos de implementación.

Una forma simple de llevarla a cabo es definiendo dos cláusulas:

1. **Precondiciones**. Presenta las condiciones que se deben de dar antes de la ejecución de la función.
2. **Postcondiciones**. Presenta las condiciones que serán ciertas después de la ejecución de la función.

Un ejemplo de este tipo de especificación se presenta a continuación:

```
double Biseccion (double izq, double der, double prec)
/*
  Precondiciones:
    — La función tiene distinto signo en los extremos del intervalo
    — prec>0
  Postcondiciones:
    — Devuelve la raíz con un error máximo de prec
*/
```

Note que las precondiciones son necesarias para hacer que la función tenga el efecto deseado, es decir, son necesarias para garantizar que se cumplen las postcondiciones. Por ejemplo, en la implementación que hemos visto de esta función, si no se cumple la condición de que la precisión sea mayor que cero, puede hacer que la función no acabe nunca (bucle infinito).

Aunque este método es muy conocido, no es el único para expresar la especificación, ya que el programador es el responsable de escoger la forma de especificar que considere más conveniente.

Podemos optar por estructurar mejor la información, ya que en muchos casos el usuario de la función desea consultar un aspecto concreto, sin necesidad de tener que leer todos los detalles. Podemos dividirla en cinco cláusulas:

1. *Parámetros*. Explica el significado de cada parámetro de la función.
2. *Valor de retorno*. Si existe, se incluye esta cláusula, que describe el valor que se devuelve en la función.
3. *Precondiciones*. Son los requisitos que se deben cumplir para que una llamada a la función se comporte como se describe en la especificación. Si no se cumplen estas condiciones, no se

garantiza ningún tipo de resultado, es decir, no se asegura nada acerca del comportamiento del programa.

4. *Efecto*. Describe el resultado que se produce cuando se realiza una llamada correcta (según las precondiciones) a la función.
5. *Excepciones*. Si es necesario, se puede incluir una sección en la que se describe el comportamiento de la función cuando se da una circunstancia que no permite la finalización exitosa de su ejecución.

El siguiente es un ejemplo de este tipo de especificación:

```
double Biseccion (double izq, double der, double prec)
/*
  Argumentos:
    izq, der: Intervalo de búsqueda de la raíz
    prec: precisión deseada en la solución
  Devuelve:
    la raíz, con un error máximo de prec
  Precondiciones:
    La función tiene distinto signo en los extremos del intervalo
    prec>0
*/
```

Herramientas para la especificación

El uso de herramientas automáticas para la especificación es un aspecto importante para mejorar la calidad y productividad en el desarrollo de programas. Los lenguajes de programación son independientes de este tipo de herramientas, por lo que la construcción y mantenimiento de especificaciones se debe realizar por parte del programador, con el uso de herramientas adicionales que incluso podemos encontrar incorporadas en entornos de desarrollo integrados.

El uso de este tipo de software nos facilita esta labor, fundamentalmente porque:

- Automatizan la generación de documentos y proporcionan herramientas para facilitar su uso.
- Permiten mantener la especificación junto con el código, haciendo más fácil que el programador relacione ambas tareas.

Un ejemplo lo encontramos en el programa *doxygen*[12], que permite incluir la especificación como comentarios junto con el código. Una vez incluida, podemos generar documentación de forma automática en distintos formatos. La forma de usar este programa, en el caso de la especificación de una función, consiste en incluir un comentario antes de la cabecera de la función, que comience con "`/**`" en lugar de "`/*`", y que incluya una serie de cláusulas que describen de una manera estructurada el comportamiento de la función. Un ejemplo es el siguiente:

```
/**
*  @brief Calcula la raíz con el método de bisección.
*  @param izq extremo izquierda del intervalo de búsqueda.
*  @param der extremo derecha del intervalo de búsqueda.
*  @param prec precisión deseada o error máximo en la solución
*  @pre  prec>0, izq<der y funcion(izq)*funcion(der)<0
*  @return la raíz con error máximo \a prec
*/
double Biseccion (double izq, double der, double prec)
```

Note que escribimos cada una de las partes de la especificación haciendo uso de palabras clave precedidas por el símbolo "@". El programa *doxygen* procesa este código y genera la especificación en el formato solicitado.[13].

[12] *http://www.doxygen.org.*

[13] Existen otros programas similares. Por ejemplo, *kdoc* que puede usarse con el entorno integrado de desarrollo *kdevelop*.

5.3.4 Pruebas

Una actividad fundamental en el desarrollo de software es la realización de pruebas. Aunque el programador inexperto puede pensar que es una etapa final, donde se prueba el programa resultante, realmente las pruebas se deben realizar desde etapas muy tempranas de desarrollo[14].

En el desarrollo de un módulo intentamos independizar ese problema del resto de los módulos. Sin embargo, no podemos olvidar que unos dependen de otros (usan otros). Es importante comprobar que un módulo es correcto antes de que sea usado por otros. En este sentido, se diseñan pruebas denominadas *de módulo*.

Por otro lado, una vez que los módulos están terminados, es necesario unirlos de manera que podremos obtener nuevos errores. Para resolverlos, podemos diseñar nuevos casos de prueba, denominados *de integración*.

Finalmente, una vez que todo el programa se ha terminado, es cuando se realizan nuevas pruebas de uso del programa, que se denominan *de aplicación*.

5.3.5 Errores

Cuando se diseña una función es necesario tomar decisiones para el caso en que se provoque una situación de error. Por ejemplo, podemos pasar unos parámetros que no son adecuados, o no es posible realizar un paso interno de la función. En esta situación, el módulo no puede llevar a cabo la tarea asignada y deberá resolver el error de alguna forma.

Una persona que está empezando a programar puede estar tentada a escribir un mensaje de error y devolver cualquier resultado. Por ejemplo, en el caso de la función `Ecuacion2Grado` podríamos haber escrito un mensaje de error en caso de que no haya soluciones.

Este módulo sería menos reusable. Imagine que alguien quiere usarlo en un programa sobre un entorno de ventanas. Seguramente, no desea que se escriba un mensaje indicando esta circunstancia, sino realizar otro tipo de acción como obtener una ventana indicando el error o pedir de nuevo los datos de entrada.

Para gestionar los casos de error, son normalmente los módulos de más alto nivel los que saben cómo tratar los errores. Por tanto, los módulos más básicos deberán implementar un mecanismo de devolución de errores, por ejemplo como resultado de la función o como parámetro por referencia[15].

Finalmente, tenemos que tener en cuenta que una función no está obligada a gestionar los casos de error si especifica una cierta precondición que garantiza que no se darán. Por ejemplo, en la siguiente función se establece que el parámetro `a` debe ser distinto de cero[16]:

```
/**
*  @brief Calcula la raíz de una ecuación de primer grado.
*  @param a coeficiente de primer grado. Es distinto de cero.
*  @param b coeficiente independiente
*  @return la raíz de ax+b=0
*/
double Ecuacion1Grado (double a, double b)
{
   return -b/a;
}
```

así que la función no está obligada a gestionar este error, aunque el efecto de la llamada pueda ser irrecuperable (en este caso, provocamos una división por cero). Es responsabilidad del que la usa que se cumpla esa condición. El que la implementa podrá o no incorporar algún mecanismo de gestión de ese error.

[14] Note que si se comenten errores en las primeras fases, se pueden arrastrar y provocar nuevos errores en las siguientes. La vuelta atrás para solucionarlo puede ser costosa.

[15] En algunos casos, incluso se utiliza una variable global para indicar el tipo de error.

[16] Se ha documentado en la descripción del parámetro. También se podría haber documentado con la palabra *@pre* de precondiciones.

5.4 Problemas

Problema 5.1 Implemente una función que reciba tres valores enteros, indicando el día, mes y año, y que devuelva si la fecha es válida o no.

Problema 5.2 Implemente una función que, a partir de la longitud y número de lados de un polígono, calcule el área del polígono (las ecuaciones puede consultarlas en el problema 2.10, página 40).

Problema 5.3 Implemente una función de confirmación que resuelva el problema de escribir un mensaje del tipo *"¿Confirmar (s/n)?"* en la salida estándar, lea de la entrada la respuesta del usuario (aceptando solamente 's','S','n','N') y devuelva si se confirmó o no.

Problema 5.4 Implemente una función para calcular el valor de $log_b x$. Recuerde que:

$$log_b x = \frac{log_{base} x}{log_{base} b}$$

Por ejemplo, puede usar $base = e$.

Problema 5.5 A continuación se presenta un trozo de código poco habitual. El lector deberá entender el funcionamiento del paso por valor y por referencia para adivinar la salida del programa. ¿Cuál es?

```
#include <iostream>
using namespace std;
void Funcion (int a, int& b, int& c)
{
   cout << a << ',' << b << ',' << c << endl;
   b= 2;
   cout << a << ',' << b << ',' << c << endl;
   c= 3;
   cout << a << ',' << b << ',' << c << endl;
}
int main()
{
   int a= 1;
   Funcion(a,a,a);
   cout << a << endl;
}
```

Problema 5.6 Implemente una función que devuelva el punto (x,y) de derivada cero en una parábola. Para ello, asumiremos como precondición que el valor del coeficiente de grado dos es distinto de cero.

Problema 5.7 Implemente una función para calcular el máximo común divisor (MCD) y mínimo común múltiplo (MCM) de dos enteros. Para ello, tenga en cuenta la relación:

$$a \cdot b = MCD(a,b) \cdot MCM(a,b)$$

Problema 5.8 Implemente una función que recibe como entrada un número de segundos y obtiene como salida el número de días, horas, minutos y segundos. Tenga en cuenta que el número de horas debe ser menor que 24 y el de minutos y segundos menor que 60 (véase el problema 2.13).

Problema 5.9 Recuerde el problema 4.6 (página 78). Reescriba el programa que calcula el doble factorial con la modularización que considere más oportuna.

6 Vectores de la STL

Introducción 109
Vectores 110
Vectores dinámicos 132
Matrices 137
Problemas 143

6.1 Introducción

En la mayoría de las aplicaciones, la complejidad de los problemas requiere la gestión de una gran cantidad de datos. No sólo es necesario crear algoritmos más complejos, sino que tenemos que organizar y procesar un mayor número de datos. El desarrollo de algoritmos adecuados sobre una adecuada forma de estructurar la información son las bases de una programación eficaz.

Recordemos que *"Algorithms+Data Structures= Programs"*, una cita que da título al conocido libro de *Niklaus Wirth*, uno de los pioneros en la programación, conocido especialmente como padre de lenguajes como *PASCAL*. Desde los primeros programas, ya se deja patente que la mejor forma de desarrollar un programa es diseñar tanto los algoritmos como las estructuras de datos. No lo considere dos problemas independientes, sino que van estrechamente relacionados; hay algoritmos que se adaptan a ciertas estructuras de datos y al revés. Avanzar en el estudio de la programación implica no sólo abordar algoritmos más complejos, sino ir incorporando estructuras de datos más avanzadas.

En los primeros problemas que hemos resuelto, los algoritmos se han limitado a trabajar con un número de objetos relativamente pequeño, cada uno de ellos de un tipo simple, que hemos podido usar una vez que hemos declarado. En este tema, aumentamos la complejidad de los tipos de datos introduciendo los conocidos como *"tipos de datos compuestos"*.

Los tipos de datos compuestos nos permiten declarar un *único objeto* que estará *compuesto* de varios objetos de otros tipos. En este sentido, podemos decir que un vector es un *contenedor*, un concepto que aquí presentamos intuitivamente y más adelante —cuando se usa la STL en profundidad— cobra especial importancia.

Concretamente, el tipo de dato `vector` es un tipo de dato compuesto que denominamos *homogéneo*, es decir, contiene una serie de objetos, todos de un único tipo de dato *base*.

A continuación se estudiará el tipo `vector` que ofrece C++ como parte de la STL. Este lenguaje también ofrece la posibilidad de definir vectores con un tipo de dato definido dentro del lenguaje, un tipo de vectores que denominamos *vectores-C* (o *arrays-C*) al estar presentes en lenguaje C.

El tipo **vector** de la STL nos permite explorar el mismo concepto, pero evitando las particularidades de los *vectores-C* e incluso añadiendo nuevas características que lo hacen más versátil y potente.

6.2 Vectores

El tipo **vector** está diseñado para poder definir un objeto donde almacenar múltiples objetos de otro tipo de dato —el tipo *base*— de forma que podamos acceder a cada uno de ellos de una forma sencilla y eficaz, permitiéndonos realizar las operaciones correspondientes como si de un objeto individual se tratara.

El tipo **vector** de la STL no está integrado en el lenguaje, sino que se ha definido como una *clase de la* STL, por lo que será necesario incluir el archivo de cabecera **vector** antes de poder usarlo.

El *tamaño de un vector* corresponde al número de objetos de tipo *base* que lo componen. La forma de referirse a cada uno de estos objetos se realiza por medio de un índice. Este índice es un valor entero que comienza en el número cero. Por tanto, un vector de tamaño n contiene n objetos enumerados con los índices $0, 1, \cdots, n-1$.

En la figura 6.1 se muestra gráficamente un objeto de tipo **vector**, compuesto de 8 objetos de tipo **double**. Observe que el rango del índice va desde el 0 hasta el 7.

0	1	2	3	4	5	6	7
5.5	3.0	10.0	4.0	0.0	3.5	6.9	5.0

Objeto: notas **Tipo base:** double **Tamaño:** 8

Figura 6.1
Ejemplo de vector.

6.2.1 Declaración

Para declarar un vector de la STL, será necesario especificar la palabra **vector** y, además, el tipo de dato base entre los caracteres <>. Realmente, el tipo vector no se refiere a un tipo de dato, sino un número ilimitado de tipos, ya que depende del tipo base especificado. Por ejemplo, un vector de objetos de tipo **int** es un tipo distinto a un vector de objetos de tipo **double**. Dado que en C++ podemos definir tantos tipos como deseemos, podemos decir que el tipo **vector** corresponde a una familia ilimitada de nuevos tipos.

En principio, la sintaxis para definir un nuevo objeto de tipo vector es la siguiente:

vector<*TIPOBASE*> *nombre_objeto* (*tamaño*);

Observe que el tipo de dato está a la izquierda del nombre del nuevo objeto, y que entre paréntesis especificamos el número de datos que lo componen. Lógicamente, todos ellos son de tipo *TIPOBASE*.

En el siguiente listado presentamos un ejemplo de declaración del vector `notas`, que contiene 5 objetos de tipo **double**:

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

int main()
{
```

```
    vector<double> notas(5);
    // ...
}
```

En este listado hemos incluido no sólo la línea de declaración, sino que hemos añadido otras líneas para enfatizar que:

1. Es necesario incluir el archivo de cabecera **vector**. Recordemos que el tipo no está integrado en el lenguaje, por tanto, debemos incluir este archivo para poder disponer de él.
2. El tipo **vector** está incluido dentro del estándar. Es necesario acceder a través de **std**. Si en el ejemplo anterior no hubiéramos especificado el espacio de nombres **std**, el compilador no hubiera reconocido el tipo **vector**.

Otra forma de declarar un vector es especificar no sólo el tamaño del vector, sino también el valor que con el que se inicializarán cada una de sus posiciones. La sintaxis es la siguiente:

> **vector** < *TIPOBASE* > *nombre_objeto* (*tamaño*, *valor_inicial*) ;

En el siguiente listado declaramos un vector `notas` que contiene `5` elementos inicializados con el valor `10`.

```
// ...
vector<double> notas(5,10.0); // 5 posiciones con el valor 10
// ...
```

En la figura 6.2 mostramos gráficamente el efecto de esta declaración. El vector contiene `5` elementos y cada uno con el valor inicial `10.0`.

0	1	2	3	4
10.0	10.0	10.0	10.0	10.0

Valor inicial: 10.0 **Tipo base:** double **Tamaño:** 5

Figura 6.2
Ejemplo de vector inicializado.

Lógicamente, el valor de inicialización será del tipo base del vector. Por ejemplo, podríamos declarar los siguientes objetos:

```
vector<int> contadores(10,0);   // 10 contadores que empiezan en cero
vector<bool> acierto(15,false); // 15 booleanos que empiezan con false
vector<char> password(8,'*');   // 8 caracteres que empiezan con '*'
```

Finalmente, consideramos la versión más simple de declaración: como *vector vacío*. La sintaxis es la siguiente:

> **vector** < *TIPOBASE* > *nombre_objeto* ;

Al no especificar ningún parámetro, el compilador preparará un objeto que no contiene ningún elemento. Es decir, es un vector con cero elementos. Por ejemplo:

```
// ...
vector<int> v;    // Sin parámetros: no contiene ningún elemento
// ...
```

crea un objeto con nombre *v*, sin elementos. Aunque parezca que no tiene uso, pues no almacena nada, más adelante estudiaremos cómo podemos hacer que los almacene.

6.2.2 Acceso al contenido

El acceso a un elemento individual de un vector se realiza especificando el índice correspondiente. La formas más simple —y más usada— es indicando el nombre del objeto y a continuación el índice entre corchetes. En concreto:

nombre_objeto [*índice*]

Por ejemplo, si hemos definido el vector `notas` como un vector de objetos **`double`** de tamaño `8`, podemos usar `notas[0]` para referirnos al primer valor, y `notas[7]` para referirnos al último. De nuevo, volvemos a recordar que el rango de índices válidos comienza en cero y, por consiguiente, un vector de tamaño 8 permite un índice que va desde 0 hasta 7 como máximo (el tamaño menos uno).

El programador podrá especificar cualquier índice en el rango válido, pero no fuera de él. Tenga en cuenta que no existen más objetos que los indicados por el tamaño del vector. El intento de acceso a un objeto que no es válido tiene resultados inesperados. En general, podemos decir que el programa fallará, pero podemos obtener desde un comportamiento tan inofensivo como un funcionamiento aparentemente correcto, hasta un error grave que hace que el sistema detenga directamente el programa.

Si piensa que este párrafo resulta excesivamente alarmista, créalo, no lo es. Es absolutamente inaceptable un acceso fuera del rango válido de un vector. Puede incluso generar una situación aún más grave y mucho más problemática. Es posible que el acceso incorrecto no sea detectado por el sistema, el programa continúe, y finalmente termine el programa con resultados incorrectos, o sin sentido. Piense un instante sobre esta situación: una vez finalizado, sabemos que hay un problema, pero no sabemos nada sobre el punto donde falla. Si el programa tiene miles de líneas, o cientos de miles, puede ser muy complicado localizar el error.

Después de esta exposición, se habrá dado cuenta de que es importante evitar estos accesos prohibidos. Puede pensar que pondrá especial cuidado y será muy difícil que le ocurra. La mala noticia es que le ocurrirá, además, muchas veces. Si le ocurre múltiples veces, y pierde mucho tiempo localizando los errores, podría pensar... ¿Es que no puede el programa avisarme del error justo en el momento en que ocurre? La respuesta a esta pregunta es que sí, es posible, y es tan fácil como usar otra forma de acceso a un elemento. En este caso, en lugar de los corchetes, usamos la siguiente sintaxis:

nombre_objeto.**at** (*índice*)

es decir, especificamos el nombre, seguido de un punto, seguido de —el nombre de la operación— y finalmente el índice entre paréntesis. En este caso, el acceso a un índice incorrecto provocaría que el programa lanzara una "señal de aviso" que en principio[1] —como puede comprobar— provoca la finalización del programa.

Ambas operaciones son equivalentes siempre que accedamos a un valor de índice correcto. Sin embargo, es lógico esperar que el uso de `at` implica programas más ineficientes, ya que cada vez que accedemos a un elemento tenemos que comprobar que la posición es válida. Por tanto, esta operación es más lenta, pero más "segura". En cualquier caso, ya usemos una u otra opción, un programa que esté terminado, depurado y en producción debería garantizar que nunca se accede a posiciones no válidas.

Un ejemplo de estas operaciones se presenta en el siguiente programa:

[1] En concreto, el programa lanza una *excepción*, que se estudiarán más adelante, y que podría ser capturada por el programa para resolver el error de forma adecuada.

```
#include <iostream> // cout, cin
#include <vector>   // vector
using namespace std;

int main()
{
   vector<double> notas(5);

   notas[0]= 2.5;
   cin >> notas[1];
   notas[2] = 5.0;
   cout << notas[10];    // Error: resultado indeterminado
   notas[3]= 7.5         // Tal vez lleguemos aquí y siga el programa
   cout << notas.at(5); // Error: El programa se parará (fuera de rango)
   notas[4]= 10;         // No llegará a esta línea
}
```

Es interesante que observe cómo se usan los objetos `double` que aparecen en este programa. Se usan exactamente como cualquier `double` que haya declarado en sus programas, es decir, se les puede asignar un valor, leer desde `cin`, obtener en `cout`, etc. La única diferencia es que ahora el nombre es un poco más extraño al estar compuesto por el nombre de un vector junto con el índice.

En los ejemplos anteriores, el índice siempre se ha especificado como un literal de tipo entero. Realmente, el valor del índice puede ser cualquier expresión que dé lugar a un valor entero. Así, podemos usar una variable para poder especificar el valor del índice, o incluso el valor resultante de una expresión más compleja. Por ejemplo, si queremos calcular la media de 5 valores podemos hacer:

Listado 5 — Archivo *media5.cpp*.

```
1  #include <iostream> // cout, cin
2  #include <vector>   // vector
3  using namespace std;
4
5  int main()
6  {
7     vector<double> notas(5);
8
9     for (int i=0; i<5; ++i)
10       cin >> notas.at(i);
11
12    double media= 0;
13    for (int i=0; i<5; ++i)
14       media= media + notas.at(i);
15    media= media/5;
16
17    cout << "Resultado: " << media << " es la media de ";
18    for (int i=0; i<4; ++i)
19       cout << notas.at(i) << ',';
20    cout << notas.at(4) << '.' << endl;
21 }
```

Observe que:

- Los bucles comienzan en cero. Recuerde que el primer elemento del vector está en la posición cero.
- La condición de parada del bucle es un menor estricto, es decir, el último valor usado es el 4, el tamaño menos uno del vector.
- En el último bucle llegamos hasta el valor 3, ya que dejamos la impresión del último elemento para la última línea.

Ejercicio 6.1 Modifique el ejemplo del listado 5 de manera que no se lean 5 valores, sino 10.

Como habrá podido comprobar, es incómodo —si no incorrecto— diseñar una solución para un número de datos predeterminado (en este caso 5 datos) de forma que una modificación tan sencilla como cambiar el número de datos requiera tantos cambios. Para resolver este problema, tal vez se le ocurra establecer una constante que indique el número de valores a procesar, sin embargo la solución es mucho más simple, ya que un vector almacena no solamente cada uno de los objetos que lo componen, sino también el número de objetos que contiene. La consulta del tamaño se hace con la siguiente sintaxis:

nombre_objeto.**size**()

que es una operación que devuelve un entero con el número de objetos que contiene el vector. Por ejemplo, para imprimir todos los elementos del vector `notas` podríamos hacer:

```
for (int i=0; i<notas.size(); ++i)
   cout << notas.at(i) << endl;
```

Ejercicio 6.2 Modifique el ejemplo anterior de manera que el número de datos a procesar sólo aparezca en la declaración del vector.

Por último, de la misma forma que el índice de un vector se puede especificar como una expresión de tipo entero, también podemos establecer el tamaño de un vector con una variable o expresión que devuelva un entero, en lugar de un literal entero.

Ejemplo 6.2.1 Escriba un programa para procesar una serie de notas obtenidas por un grupo de alumnos. El programa pregunta el número de alumnos y a continuación las notas que ha obtenido cada uno de ellos. Después de introducir los datos, preguntará repetidamente dos valores que determinan un intervalo, imprimiendo para cada intervalo el número de calificaciones que están dentro de él. El programa termina cuando se introduce un intervalo incorrecto (el primer valor es mayor que el segundo).

Aunque la solución requeriría algo más de código, depurando las entradas para que los valores sean correctos (el número de alumnos es un entero positivo y una nota es un valor entre 0 y 10), simplificaremos la solución para centrarnos en el algoritmo principal y en los detalles del tipo vector.

Listado 6 — Archivo *notas.cpp*.

```
1  #include <iostream> // cout, cin, endl
2  #include <vector>   // vector
3  using namespace std;
4
5  int main()
6  {
7     int n;
8
9     cout << "Introduzca el número de calificaciones: ";
10    cin >> n;
11
12    vector<double> notas(n);
13
14    for (int i=0; i<notas.size(); ++i) {
15       cout << "Introduzca calificación " << i+1 << ": ";
16       cin >> notas.at(i);
17    }
18
19    double izq, der;
20    cout << "Introduzca intervalo: ";
21    cin >> izq >> der;
```

```
22
23      while (izq<=der) {
24         int ndatos= 0;
25         for (int i=0; i<notas.size(); ++i)
26            if (izq<=notas.at(i) && notas.at(i)<=der)
27               ndatos++;
28         cout <<"Datos en [" << izq <<','<<der<<"]: " << ndatos << endl;
29
30         cout << "Introduzca otro intervalo: ";
31         cin >> izq >> der;
32      }
33   }
```

Observe que en esta solución:

- Podríamos haber usado la variable `n` a lo largo del programa para referirnos al tamaño del vector. Sin embargo, cuando se realiza un programa que recorre todas las posiciones de un vector siempre será más seguro usar la operación `size`, ya que es la que garantiza que se ajusta al tamaño. Imagine, por ejemplo, que usamos la variable `n` para otra operación y cambiarla provocaría como efecto secundario que los bucles no se ajustan al número de datos.
- En los bucles hemos usado el preincremento `++i`. Hubiera sido también válido y equivalente usar el postincremento `i++`.
- Este programa debería ampliarse. Por ejemplo, deberíamos filtrar el tamaño `n`, ya que el programa no controla si el usuario dará un valor demasiado alto o incluso negativo. En este caso, podría provocarse una terminación anormal del programa que, como sabemos, no debería permitirse.

Ejercicio 6.3 Modifique el ejemplo anterior para garantizar que el tamaño es un entero positivo y las notas están en el rango $[0,10]$.

Fin ejemplo 6.2.1 ■

Ejercicio 6.4 Modifique el ejemplo anterior añadiendo al final un trozo de código que primero calcule la media de las notas introducidas, y segundo calcule el número de datos que se encuentren por encima y por debajo de la media. En este trozo de código, use los corchetes como método de acceso a cada posición.

Ejemplo 6.2.2 Suponga un vector de objetos `v` de tipo **`int`**. Escriba un trozo de código para invertir el vector. El resultado es que al final el elemento que estaba en la primera posición estará en la última, el de la segunda en la penúltima, etc.

La solución la podemos plantear como el recorrido de los elementos del vector para intercambiar cada pareja de valores “simétricos”. Por ejemplo, si el vector tiene tamaño 10, tendremos que intercambiar:

```
int aux;
aux= v[0];
v[0]= v[9];
v[9]= aux;
```

para todas las parejas (v_0, v_9), (v_1, v_8), (v_2, v_7), etc. La solución puede ser la siguiente:

```
for (int i=0; i<v.size()/2; ++i) {
   int aux= v[i];
   v[i]= v[v.size()-1-i];
   v[v.size()-1-i]= aux;
}
```

donde es interesante que observe que:

- Intercambiamos el elemento `i` con el `v.size()-1-i`. Si sustituye el caso concreto de `i=0` puede comprobar que da lugar al primero de los intercambios.
- Llegamos a la mitad del vector. Si el bucle lo hubiéramos implementado para todos los posibles valores de `i` —hasta `size()` menos uno— el resultado hubiera sido que el vector `v` contendría exactamente los mismos valores que al principio, como si no hubiéramos hecho nada.

Fin ejemplo 6.2.2 ■

Asignaciones de vectores

La asignación de vectores es una operación válida con los mismos efectos que la asignación de tipos de datos simples. Lógicamente, el vector es un tipo de dato compuesto, un contenedor, que puede ser de un tamaño considerable, pero que puede asignarse en su totalidad con una única instrucción de asignación. En el código siguiente podemos ver un ejemplo:

```
vector<double> notas1(10, 1.0); // Contiene 10 objetos con valor 1.0
vector<double> notas2(20, 2.0); // Contiene 20 objetos con valor 2.0

notas2= notas1;
```

Aunque en las secciones anteriores hemos trabajado —a través de un índice— con cada uno de los objetos que componen un vector, en la asignación consideramos el nombre de un objeto de tipo **`vector`** como un todo, es decir, como un contenedor que tiene un determinado número de elementos y cada uno de ellos con un valor concreto. La asignación duplica el contenido del vector de la derecha para reemplazar el contenido del de la izquierda.

En el ejemplo anterior, el vector `notas1` se asigna al vector `notas2`. Como resultado, el vector `notas2` tiene exactamente 10 elementos con el valor 1.0. Además, si modificamos uno de los vectores, el otro vector seguirá teniendo exactamente esos valores, ya que son dos objetos independientes.

Otro ejemplo es el intercambio de los valores de dos variables, una operación habitual que hemos implementado con los tipos de datos simples. El código para realizarlo con vectores es muy similar:

```
vector<double> notas1(10, 1.0); // Contiene 10 objetos con valor 1.0
vector<double> notas2(20, 2.0); // Contiene 20 objetos con valor 2.0

vector<double> aux;

aux= notas1;
notas1= notas2;
notas2= aux;
```

Observe que la variable `aux` no se crea con ningún dato. Sería absurdo declararla con algún elemento si lo primero que vamos a hacer es reemplazar su contenido con una copia de `notas1`. Al final `notas1` acaba con 20 datos con el valor 2.0, y tanto `notas2` como `aux` tendrán 10 datos con el valor 1.0. Exactamente como en el caso de los datos simples.

Realmente, la única diferencia relevante que podría interesarnos para el caso de intercambiar los valores de dos vectores es que pueden llegar a ser muy grandes, y esas 3 asignaciones convertirse en algo costoso en tiempo y espacio[2]. Tenga en cuenta que una asignación implica muchos objetos a copiar y que estamos usando otro objeto vector —`aux`— que duplica los elementos que había en `notas1`.

[2]En este punto del curso no introducimos más posibilidades, pero resulta importante indicar que el estándar ofrece un algoritmo `swap` para poder realizar este intercambio de una manera muy eficiente.

6.2.3 Funciones y tipo vector

El uso de vectores en relación con las funciones no requiere prácticamente ningún tipo de discusión, ya que no implica ninguna situación novedosa. Es decir, si ya sabe manejar funciones con otros tipos simples, el comportamiento con los vectores es idéntico. Recordemos:

- Si queremos usar un vector como parámetro de entrada, podemos declarar un parámetro de tipo vector que pasa por valor. Como con otros tipos, el paso por valor implica una copia del vector original de forma que el objeto que manejamos en la función puede modificarse sin que afecte al parámetro real (el argumento de llamada).
- Si queremos usar un vector como entrada y salida o sólo salida, podemos declarar un parámetro de tipo vector que pasa por referencia. Al igual que con otros tipos, esto lo hacemos simplemente añadiendo el carácter `&` después del tipo. En este caso, si la función modifica el parámetro formal, el cambio afectará al parámetro real, es decir, al argumento usado en la llamada.
- Si queremos obtener un vector como resultado de una función, podemos devolver un objeto de tipo vector. Para ello, basta con declarar la cabecera de la función indicando el tipo vector antes del nombre de la función, como con otros tipos simples. Lógicamente, será la palabra reservada **`return`** la que indique el vector que debe enviarse como resultado al punto de llamada.

Ahora bien, en el caso de objetos de tipo vector debemos considerar un nuevo factor: el tamaño de un objeto de este tipo puede resultar muy grande. Por tanto, una copia del objeto puede implicar un gran esfuerzo.

Como consecuencia de ello, diseñar una función en la que intervienen vectores debe tener en cuenta, también, si el esfuerzo de realizar esas copias es aceptable o no. Tenga en cuenta que el paso por valor y la devolución de un vector como resultado de una función implican[3] una copia completa del vector, es decir, la creación de un nuevo objeto copiando todos y cada uno de los datos que lo componen.

Para resolver esta situación, podemos plantear las siguientes soluciones:

- Para evitar devolver un objeto de tipo vector podemos pasar por referencia un vector donde obtener el resultado. Esto puede llevarnos a una función de tipo **`void`**, ya que el código de llamada ofrece directamente el nombre del objeto donde queremos obtener el resultado.
- Para evitar copiar un vector que sólo se va a usar de entrada, podemos pasarlo por referencia. El problema de este cambio es que el compilador considerará que queremos modificarlo, y puede generar errores en caso de que sea un vector no modificable.
  Realmente, pasar un objeto por referencia es equivalente a decir al compilador que vamos a modificar el original, aunque no lo hagamos. Por tanto, para poder hacerlo correctamente, tendremos que añadir la palabra reservada **`const`** para que el compilador sepa que es un objeto que no se podrá modificar.

Aunque esta discusión se incluya en este tema, no significa que sea exclusivo de vectores, sino que puede usarse con cualquier otro tipo, incluso para los tipos simples que hemos visto. Sin embargo, es con un tipo que puede ser muy grande cuando se plantea la necesidad de usarlo. Por tanto, podemos decir que nos planteamos una nueva situación: si un parámetro es de entrada, puede pasarse por valor y realizar la copia o puede pasarse como **`const`**`&` para usar el original sin modificarlo.

[3] Realmente, un buen compilador puede resolver alguna de estas situaciones evitando hacer la copia. Sin embargo, no podemos dar por hecho que el compilador lo resuelve sino que debemos diseñar la función pensando en que se realizarán las copias.

Ejemplo 6.2.3 Escriba un programa que lea una serie de números en un vector de tipo **double**, y escriba como resultado la suma de todos ellos. Para resolverlo, incluya una función que suma todos los valores que incluye un vector.

```
1  #include <iostream> // Para cout, cin, endl
2  #include <vector>   // Para vector
3  using namespace std;
4
5  double Sumatoria (const vector<double>& v)
6  {
7     double suma= 0;
8     for (int i=0; i<v.size(); ++i)
9        suma+= v[i];
10    return suma;
11 }
12
13 int main()
14 {
15    int n;
16
17    cout << "Introduzca el número de datos: ";
18    cin >> n;
19
20    vector<double> datos(n);
21
22    for (int i=0; i<datos.size(); ++i) {
23       cout << "Introduzca dato " << i+1 << ": ";
24       cin >> datos[i];
25    }
26
27    cout << "Suma total: " << Sumatoria(datos) << endl;
28 }
```

En el diseño de la función *Sumatoria* hemos tenido en cuenta que el vector que se introduce es sólo de entrada, es decir, sólo lo vamos a consultar sin modificar. La idea es que cuando volvamos a **main** el vector *datos* siga valiendo lo mismo. Podríamos haberlo pasado por valor, es decir, podríamos haber escrito:

```
double Sumatoria (vector<double> v)
{
   double suma= 0;
   for (int i=0; i<v.size(); ++i)
      suma+= v[i];
   return suma;
}
```

y el resultado de la suma hubiera sido el mismo. Sin embargo, la ejecución hubiera implicado más tiempo, ya que en la llamada desde el **main** —línea 27— el paso por valor implicaría que *v* es un objeto copia de *datos*. Por tanto, hubiéramos "perdido" tiempo copiando el vector original. En lugar de eso, lo pasamos por referencia. Por tanto, cada vez que accedemos a una posición de *v*, realmente estamos usando el vector *datos*.

Por último, fíjese cómo es recomendable incluir la palaba **const** para indicar que es de entrada. Puede probar a cambiar el programa haciendo que dentro de la función, además de acumular el elemento i-ésimo en la suma, lo modifique para hacerlo cero. Verá que obtenemos un error de compilación. Con la palabra **const** le hemos indicado al compilador que es un objeto que no debe cambiar, y es el mismo compilador el que cuidará de que sea así.

Fin ejemplo 6.2.3 ■

6.2.4 Búsqueda con vectores de la STL

El algoritmo de búsqueda se refiere al problema de localizar un elemento en un vector. El resultado del algoritmo será la posición del elemento —en caso de que se encuentre— o un valor especial para indicar que no se encuentra.

El diseñador de la función debe decidir qué valor tiene que devolver en caso de que el elemento no se encuentre. Tenemos muchas posibilidades, ya que podría valer cualquier número que no sea un índice válido del vector. Lo más habitual es:

- Un valor -1, que deja muy claro que no es una posición válida.
- Una posición más allá de la última posición. Tiene especial interés si tenemos en cuenta que una posición es un valor mayor o igual que cero. Si asumimos que una función de búsqueda devuelve una posición, cualquier posición debe ser no negativa; el valor -1 no sería un valor para posición. Por ejemplo, puede devolver el tamaño del vector —la siguiente a la última— para indicar que no se encuentra[4].
- Una constante predeterminada. Esta solución permite adaptarse a distintas posibilidades, ya que si decidimos usar otro valor, simplemente tenemos que cambiar esta constante para que todo funcione. Por ejemplo, si hemos desarrollado múltiples programas que comprueban si devuelve -1 y decidimos usar otro valor, los tendremos que cambiar con el nuevo valor. Si hemos definido una constante, los programas que la usan seguirán siendo válidos, ya que no hacen referencia al valor concreto, sino a dicha constante.

Secuencial

La versión más simple que se nos puede ocurrir es recorrer todos y cada uno de los elementos de un vector a fin de determinar si se encuentra o no. Lógicamente, el recorrido se hará de forma secuencias desde el primer elemento hasta el último. La función puede ser:

```
int Secuencial (const vector<double>& v, double elemento)
{
   int resultado= -1;
   for (int i=0; i<v.size() && resultado==-1; ++i)
      if (v[i]==elemento)
         resultado= i;

   return resultado;
}
```

Ejercicio 6.5 En la función anterior siempre obtendremos la posición de la primera ocurrencia del elemento en el vector. Proponga un cambio en la función con el objetivo de que se use para localizar otras posiciones donde también se encuentra el elemento.

Secuencial garantizada

Aunque muy similar al algoritmo mostrado en la sección anterior, es interesante destacar un caso particular de búsqueda secuencial: la búsqueda de un elemento que sabemos que está en el vector. Esta función sólo se puede llamar en caso de que se pueda garantizar que el elemento está en alguna posición del vector. Si el elemento no está, el efecto de la llamada es indeterminado, dando lugar —probablemente— a un error en tiempo de ejecución.

Una posible implementación del algoritmo es la siguiente:

```
int SecuencialGarantizada (const vector<double>& v, double elemento)
{
   int resultado= 0;
   while (v[resultado]!=elemento)
      resultado++;

   return resultado;
}
```

En este código se puede ver la ventaja de esta variación. En cada iteración sólo es necesario comprobar si hemos encontrado el elemento, ignorando si hemos llegado o no al final del vector. Esta simplificación en el número de operaciones por iteración hace que la función sea más rápida.

[4]En la práctica, esta posición es la que se usa cuando optamos por esta forma de señalar que no se encuentra el elemento.

Recuerde que a este nivel estamos interesados, especialmente, en programas que sean robustos y fáciles de mantener. En este caso, la primera función parece más recomendable. Sin embargo, no podemos olvidar que si un programa necesita ser optimizado, tendremos que recurrir a modificaciones de este tipo a fin de conseguir disminuir el tiempo de ejecución en lo posible.

De hecho, cuando estamos buscando un elemento en un vector que no sabemos si lo contiene, podemos optar por modificar el vector añadiendo el elemento al final y llamar a una función como la que hemos presentado. En este caso, estamos seguros de que el elemento está.

Por ejemplo, si tenemos un vector de 10 elementos, podemos añadir el elemento buscado en la posición 10 —el vector pasa a tener 11 elementos— y llamar a la función. Si obtenemos como resultado el valor 10, sabemos que el elemento no estaba en el vector original. Observe cómo este método sugiere que devolver el índice después de la última posición es una forma de señalar la no existencia del elemento en el vector.

Binaria o dicotómica

La búsqueda binaria es un algoritmo que nos permite localizar un elemento en un tiempo de ejecución mucho más corto que la búsqueda secuencial. El problema es que es necesario que los elementos estén ordenados.

No es un algoritmo especialmente novedoso si consideramos que en la vida real aplicamos una técnica parecida cuando buscamos un elemento en una secuencia ordenada. Por ejemplo, cuando buscamos una palabra en un diccionario, seleccionamos una posición y comprobamos si el elemento buscado está antes o después. Un vez determinada la parte en la que está, volvemos a repetir la operación, seleccionando una posición y comprobando de nuevo si está antes o después de la nueva posición.

El algoritmo consiste, básicamente, en repetir lo siguiente:

1. Dividir el vector en dos partes de igual tamaño.
2. Si el elemento está en la izquierda, reducir el espacio de búsqueda a ese subvector de la izquierda y, en caso contrario, reducir el espacio de búsqueda al subvector de la derecha.

Este proceso iterativo se repetirá hasta que nos quedemos con un solo elemento, y por tanto, la búsqueda concluirá comprobando si este elemento es el buscado.

Como debemos controlar en qué subvector estamos buscando el elemento, declararemos dos índices (`izq`, `der`) que indican qué parte del vector estamos explorando. Inicialmente, estos índices valdrán 0 y `size()-1`, respectivamente. El proceso iterativo acaba cuando ambos índices son iguales.

El código correspondiente a este algoritmo puede ser el siguiente:

```
int BusquedaBinaria(const vector<int> &v, int elem)
{
   int izq= 0, der= v.size()-1, centro;

   while (izq<der) {
      centro= (izq+der)/2;
      if (v.at(centro)<elem)
         izq= centro+1;
      else der= centro;
   }

   return v.at(izq)==elem ? izq : -1;
}
```

Ejercicio 6.6 En la función anterior escogemos entre dos mitades por cada iteración del bucle. La primera mitad empieza en `izq` e incluye la posición `centro`. Considere la posibilidad de dejar esa posición en la segunda mitad. ¿El código sería correcto?

Observe que en la solución anterior siempre iteraremos hasta que encerremos un único elemento entre los índices `izq`, `der`. Es posible que haya pensado en la posibilidad de localizar directamente el elemento cuando nos preguntamos por el elemento central, de forma similar a cuando busca una palabra en el diccionario.

Podemos plantear un algoritmo que comprueba el elemento central y que se detiene en caso de que sea el buscado. Si no lo es, podemos seguir la búsqueda en la primera o segunda mitad, sin incluir ese elemento central. El código en este caso podría ser el siguiente:

```
int BusquedaBinaria(const vector<int> &v, int elem)
{
   bool encontrado= false;
   int izq= 0, der= v.size()−1, centro;

   while (izq<=der && !encontrado) {
      centro= (izq+der)/2;
      if (v.at(centro)>elem)
         der= centro−1;
      else if (v.at(centro)<elem)
              izq= centro+1;
           else encontrado = true;
   }

   return encontrado ? centro : −1;
}
```

Observe que si encontramos el elemento en la posición `centro`, el algoritmo termina. Por tanto, nos ahorramos las siguientes iteraciones. Cada iteración requiere más comprobaciones, aunque descartamos un elemento más cuando nos quedamos con la mitad izquierda (movemos `der`).

Por último, es interesante que estudie con detenimiento la condición de fin del bucle. Si no está el elemento, el bucle sigue hasta que *los índices se cruzan*. Para ello, piense en las situaciones que se pueden generar cuando quedan uno o dos elementos que estudiar.

6.2.5 Algoritmos clásicos de ordenación

Los algoritmos de ordenación constituyen un tema muy amplio que tratan el problema de reorganizar una serie de elementos siguiendo un orden determinado. Aunque existen múltiples algoritmos y variantes propuestos a lo largo de las últimas décadas, aquí sólo vamos a centrarnos en los más básicos, primero para introducir al lector en el problema de la ordenación y segundo para mostrar ejemplos prácticos de solución de problemas usando vectores.

Los algoritmos presentados se denominan de *selección*, *inserción* y *burbuja*. Los tres se consideran básicos, ya que son relativamente simples de implementar. Además, se consideran poco eficientes, ya que hay otros algoritmos que resuelven el problema en mucho menos tiempo, especialmente si el número de elementos a ordenar es muy grande.

Los algoritmos que estudiamos a continuación reciben como entrada un vector de elementos con un orden indeterminado. Los implementamos como una función que recibe el vector por referencia, ya que el efecto será la reorganización de sus elementos de tal forma que, finalmente, el vector queda con todos ellos ordenados de menor a mayor. Como ejemplo ilustrativo, ordenaremos un vector de enteros.

Selección

El algoritmo de ordenación por selección responde a la idea de que podemos ordenar los elementos del vector resolviendo una a una todas las posiciones del vector. Para ello:

1. Empezamos con la primera, *seleccionamos* el elemento más pequeño del vector, y lo situamos como primero.

2. Al quedar la primera posición resuelta, resolvemos la segunda con el mismo procedimiento, es decir, *seleccionamos* el más pequeño del resto del vector y lo colocamos en la segunda posición.
3. Al quedar las dos primeras resueltas, resolvemos la tercera **seleccionando** el elemento más pequeño del resto del vector y colocándolo como tercero.
4. El procedimiento se repite hasta que situamos el penúltimo elemento en su sitio.

Observe que para poder ordenar un elemento es necesario *seleccionar* el más pequeño de los elementos del vector a partir de la posición a resolver.

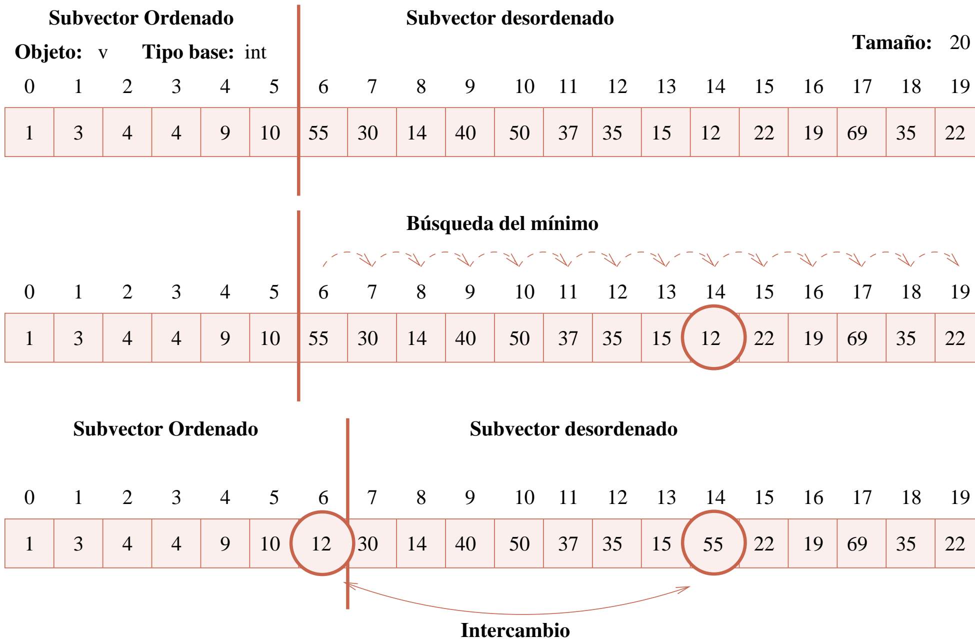


Figura 6.3
Ordenación por selección.

Un ejemplo de iteración para resolver la posición 6 se puede ver en la figura 6.3. Observe que el subvector en las posiciones 0-5 ya está definitivamente ordenado. El primer paso consiste en buscar el mínimo del subvector desordenado, mínimo localizado en la posición 14. Podemos implementar la siguiente función para localizarlo.

```
int BuscarMinimo(const vector<int> &v, int pini)
{
   int pmin= pini;
   for (int i=pini+1; i<v.size(); i++)
      if (v.at(i)<v.at(pmin))
         pmin= i;
   return pmin;
}
```

que localiza el índice del elemento más pequeño teniendo en cuenta las posiciones que van desde `pini` hasta el final del vector. Tenga en cuenta que devolvemos la posición donde está el elemento, no el elemento en sí.

A continuación debemos intercambiar el número localizado con la posición que estamos ordenando. En el ejemplo, intercambiamos el elemento de las posiciones 6 y 14. Después de esto, ya tenemos resuelta la iteración. Si realizamos esta operación para todas las posiciones, obtendremos el vector ordenado.Por tanto, podemos expresar el algoritmo como:

```
void OrdenarSeleccion(vector<int> &v)
{
   for (int pos=0; pos<v.size()−1; pos++) {
      int pmin= BuscarMinimo(v,pos);
      Intercambiar(v.at(pos),v.at(pmin));
   }
}
```

donde podemos ver que en cada paso resolvemos el elemento de la posición `pos`. Para ello, localizamos el más pequeño a partir de `pos` y, una vez localizado, lo intercambiamos con el de esta posición. Como sabe, la función de intercambio es la siguiente:

```
void Intercambiar(int &v1, int &v2)
{
   int aux= v1;
   v1= v2;
   v2= aux;
}
```

Es interesante observar que el algoritmo resuelve todas las posiciones excepto la última: si todos los elementos hasta el penúltimo están en su lugar, el último seguro que también está resuelto.

Por otro lado, el algoritmo `BuscarMinimo` podría localizar la posición `pmin` con el mismo valor que `pos`, es decir, el mínimo ya está en su sitio. En este caso, la función `Intercambiar` dejaría el vector sin cambios al intercambiar un elemento consigo mismo.

En la figura 6.4 se puede ver el progreso del algoritmo por selección aplicado a un vector de elementos aleatorio. Se presentan distintos momentos, al 25 %, 50 %, 75 % y 100 % de elementos ordenados. Se han enfatizado los elementos ordenados representándolos como una barra sólida con el tamaño proporcional al valor, mientras que los elementos sin ordenar se representan únicamente con un punto a la altura correspondiente. Observe que en la parte derecha todos los valores están situados por encima del último ordenado. Recordemos que los de la izquierda ya están ordenados en su posición definitiva, por tanto, no puede haber elementos pequeños por ordenar.

Ejercicio 6.7 Reorganice el código anterior de forma que el algoritmo esté incluido en su totalidad en una única función `OrdenarSeleccion`. Intente reescribirlo teniendo en cuenta la lógica del algoritmo en lugar de consultar los detalles del código anterior.

Inserción

El algoritmo de ordenación por inserción responde a una forma intuitiva de ordenar una serie de objetos. Es probable lo haya aplicado en la vida real, cuando tiene que ordenar una serie de objetos por algún criterio. Por ejemplo, si tiene una serie de folios numerados, puede ordenarlos comenzando con unos pocos y después ir insertando, en su lugar correspondiente, uno a uno el resto de folios.

Si quiere pensar en un ejemplo sin números, imagine que tiene un montón de expedientes de personas y tiene que ordenarlos de forma alfabética. Es posible que los primeros 4 o 5 sean fáciles de ordenar y termine por ir añadiendo el resto uno a uno, insertándolos en su lugar entre los ya ordenados.

Esta misma idea la podemos aplicar para ordenar un vector. Imagine que tenemos un vector `v` con `pos` elementos ordenados (desde la posición cero a `pos−1`) y el resto de elementos sin ordenar. Podemos realizar una operación para añadir un elemento más al conjunto de ordenados: el de la posición `pos`.

Si resolvemos el problema de *insertar* ese nuevo elemento, pasaremos a tener los primeros `pos+1` elementos ordenados. Para conseguirlo, desplazamos una posición hacia la derecha todos los mayores al situado en la posición `pos`, para finalmente insertar el elemento que estaba en `pos` hasta su lugar correspondiente. Es lo mismo que haría para ordenar un montón de expedientes: saca

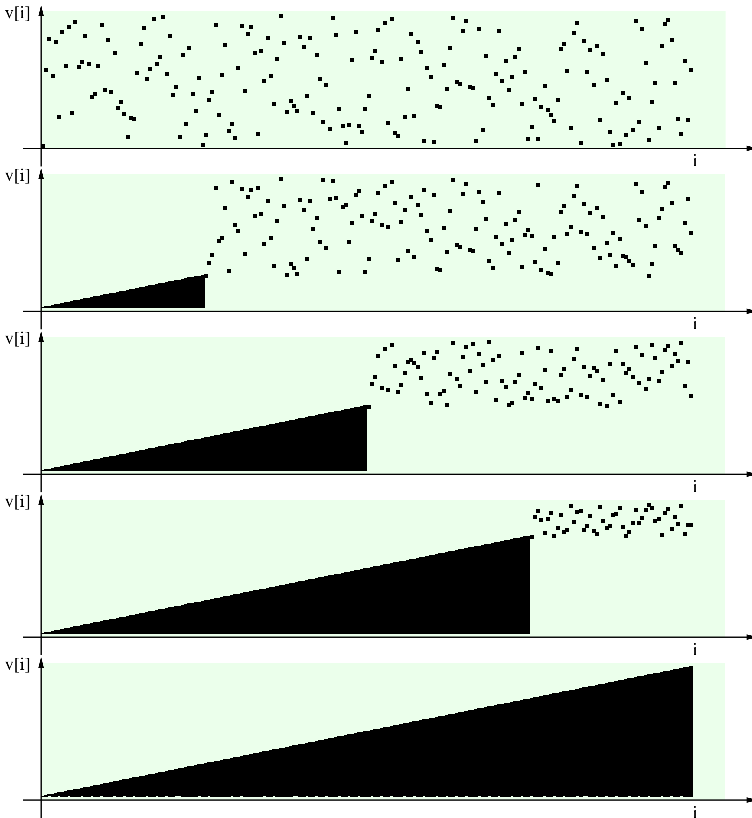


Figura 6.4
Evolución del algoritmo de ordenación por selección.

el expediente a ordenar, desplaza los que quedan por encima de él y, cuando encuentra su sitio, lo inserta.

En la figura 6.5 se presenta un ejemplo de desplazamiento para conseguir insertar un elemento más entre los ordenados. Es este ejemplo, las primeras 14 posiciones ya están ordenadas. La operación consiste en pasar a tener 15 posiciones ordenadas, insertando el elemento de índice 14 a su posición ordenada.

A continuación se presenta el código para realizar esta operación de insertar un elemento más. Esta función sólo añade un elemento al conjunto de ordenados, suponiendo que los `pos` elementos iniciales están ya ordenados. El código podría ser el siguiente:

```
void DesplazarHaciaAtras(vector<int> &v, int pos)
{
   int aux = v.at(pos); // sacamos el de pos

   int i;
   for (i=pos; i>0 && aux<v.at(i-1); i--)
      v.at(i)= v.at(i-1);
   v.at(i)= aux;
}
```

Observe que para poder llamarla, las posiciones de `0` a `pos-1` deben estar ordenadas. El desplazamiento se hace a lo largo del subvector ordenado a la izquierda de `pos`. Precisamente es esta condición la que garantiza que el resultado final de esta función es que el subvector de una posición más (de la `0` a la `pos`) queda ordenado.

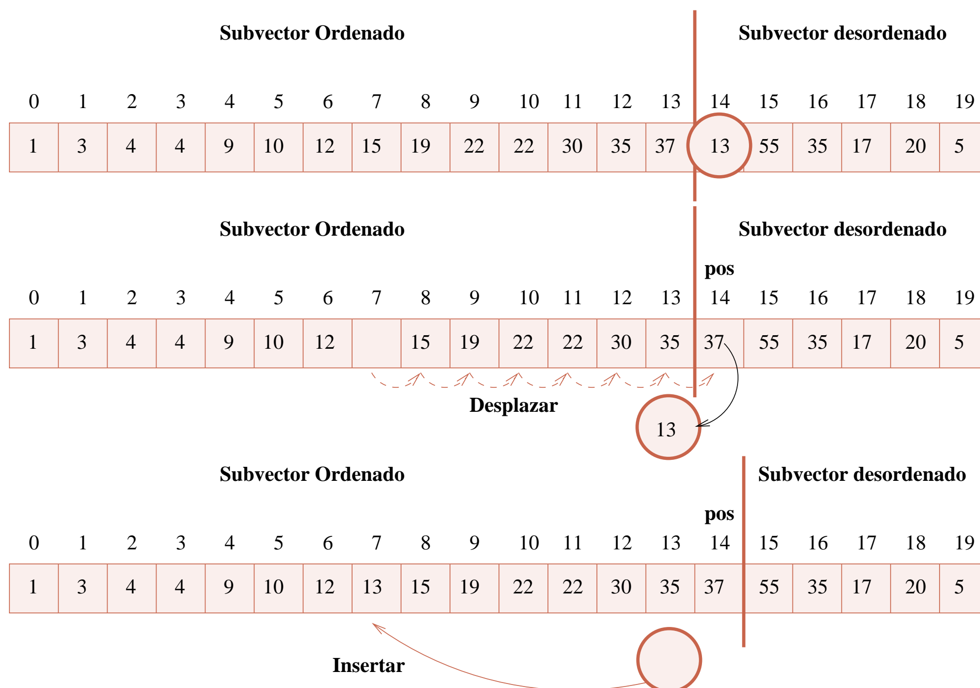


Figura 6.5
Ordenación por inserción.

Por otro lado, la variable `i` que se usa en el bucle se ha declarado fuera de él. Si no fuera así, la última línea de la función sería incorrecta, ya que la variable `i` no existiría fuera del bucle **for**.

> **Ejercicio 6.8** Considere la condición del bucle anterior. Es una condición compuesta que usa el operador lógico *AND*. ¿Qué ocurriría si cambiamos el orden de los operandos?

El algoritmo de ordenación por inserción comienza considerando que la primera posición del vector ya está ordenada, pues sólo tiene un elemento. Después, desde la posición *1* a la última, aplica el algoritmo de desplazamiento de forma que, en cada iteración, hace que el conjunto de elementos ordenados crezca en uno. El código podría ser el siguiente:

```
void OrdenarInsercion(vector<int> &v)
{
   for (int pos=1; pos<v.size(); pos++)
      DesplazarHaciaAtras(v,pos);
}
```

Observe que el bucle comienza en la posición `1`, ya que comenzamos suponiendo que tenemos un subvector —con el elemento `v[0]`— ordenado de tamaño uno.

En la figura 6.6 se puede ver el progreso del algoritmo por inserción aplicado a un vector de elementos aleatorio. Se presentan distintos momentos, al 25 %, 50 %, 75 % y 100 % de elementos ordenados. Se han enfatizado los elementos ordenados representándolos como una barra sólida con el tamaño proporcional al valor, mientras que los elementos sin ordenar se representan únicamente con un punto a la altura correspondiente. Observe que en la parte derecha hay valores situados en cualquier altura. Realmente, ni se han llegado aún a consultar.

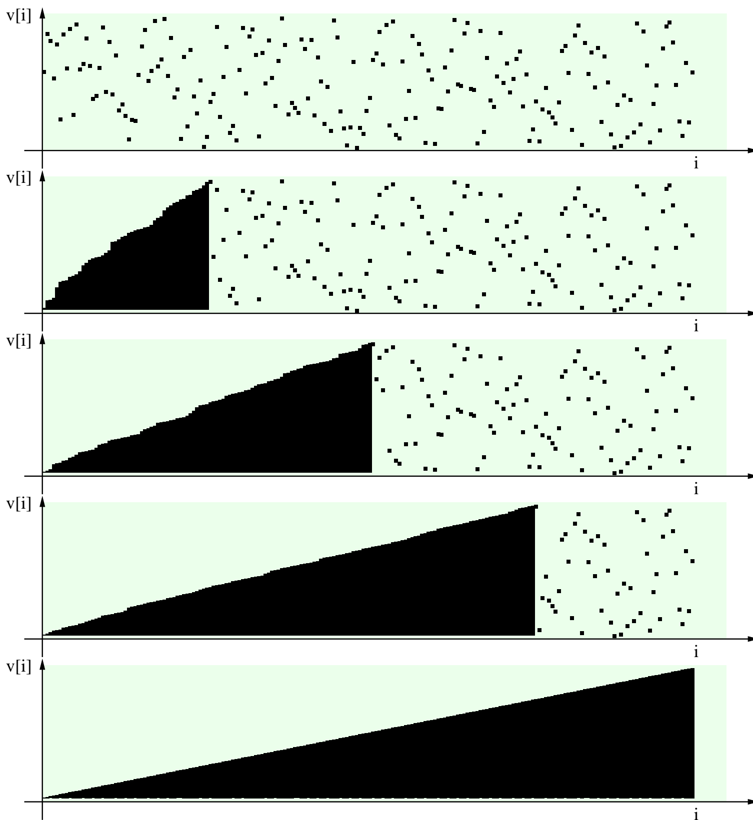


Figura 6.6
Evolución del algoritmo de ordenación por inserción.

Ejercicio 6.9 Reorganice el código anterior de forma que el algoritmo esté incluido en su totalidad en una única función `OrdenarInsercion`. Intente reescribirlo teniendo en cuenta la lógica del algoritmo en lugar de consultar los detalles del código anterior.

Burbuja

El nombre de este algoritmo se deriva de la forma en que se mueven los elementos para que queden ordenados. Cuando se intenta ordenar un elemento, el vector se procesa de un extremo al otro, haciendo que los elementos mayores se desplacen hacia su posición de tal forma que sugiere —con un poco de imaginación— el ascenso de una burbuja.

Considere el problema de poner el elemento más grande en la última posición del vector. Para realizarlo, podemos comparar los elementos dos a dos: el par en posiciones (0,1), el par (1,2), el par (2,3) etc. La comparación se hace de forma que si los dos elementos no están ordenados se intercambian.

Cuando el algoritmo llega a la pareja de posiciones `(size−2, size−1)`, es decir, a la última, tendremos el mayor de los elementos al final. Tenga en cuenta que en cuanto alcancemos ese elemento máximo, cualquier par que se compara hará que ese elemento quede por encima, y por consiguiente se arrastrará par a par hasta que se sitúa al final.

En la figura 6.7 se presenta un ejemplo de ordenación con este algoritmo. En los distintos estados se presenta la situación después de ordenar cada par de valores y, por lo tanto, al final se

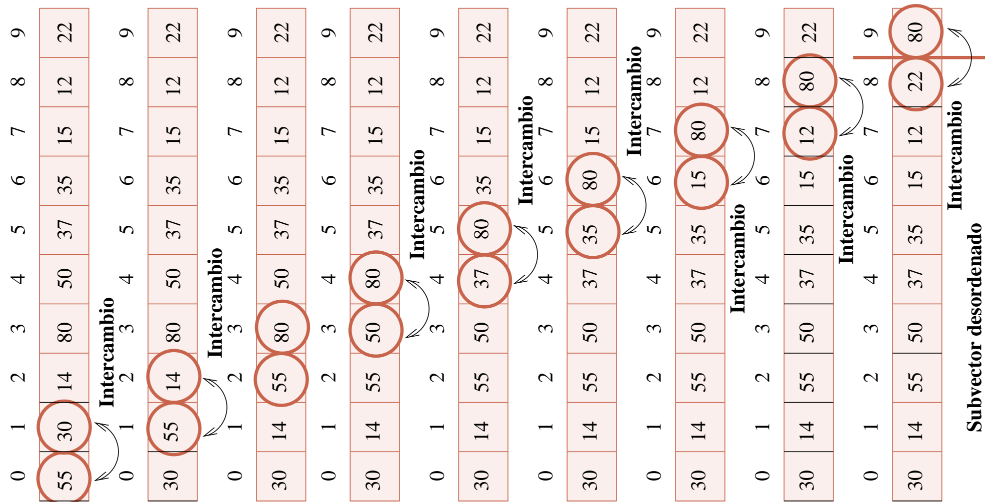


Figura 6.7
Ordenación de la burbuja.

presenta el estado del vector con el mayor de los elementos en su posición. En este caso, hemos dibujado los vectores verticalmente con la intención de que al lector le resulte más sencillo asociar este algoritmo con su nombre.

El código para realizar esta operación, la de elevar el máximo elemento hasta la cierta posición, podría ser el siguiente:

```
void SubirElemento (vector<int> &v, int hasta)
{
   for (int i=0; i<hasta; i++)
      if (v.at(i)>v.at(i+1))
         Intercambiar(v.at(i),v.at(i+1));
}
```

donde podemos ver que el algoritmo compara cada pareja de valores hasta que el mayor elemento llega a la posición `hasta`. Con esta función, para efectuar la operación que refleja la figura 6.7 no tendríamos más que llamar a esta función con el valor `size-1` como segundo parámetro.

Lógicamente, si volvemos a llamar a la función con el valor `size-2`, obtendríamos también el penúltimo elemento ordenado. Repetir esta operación para todas las posiciones desde la última a la primera nos permitirá obtener el vector totalmente ordenado.

```
void OrdenarBurbuja(vector<int> &v)
{
   for (int hasta=v.size()-1; hasta>0; hasta--)
      SubirElemento(v,hasta);
}
```

Observe que el bucle itera hasta el valor `1`, ya que si resolvemos esta posición, seguro que la posición cero queda correctamente resuelta.

En la figura 6.8 se puede ver el progreso del algoritmo de la burbuja aplicado a un vector de elementos aleatorio. Se presentan distintos momentos, al 25 %, 50 %, 75 % y 100 % de elementos ordenados. Es interesante observar que en las fases intermedias ya se intuye cierto orden incluso en las posiciones que aún quedan pendientes. Por ejemplo, cuando tenemos el 25 % ordenado, los elementos que quedan en la zona más baja ya tienden a ordenarse. Tenga en cuenta que, aunque sólo elevamos un elemento, en las posiciones anteriores también se han realizado intercambios.

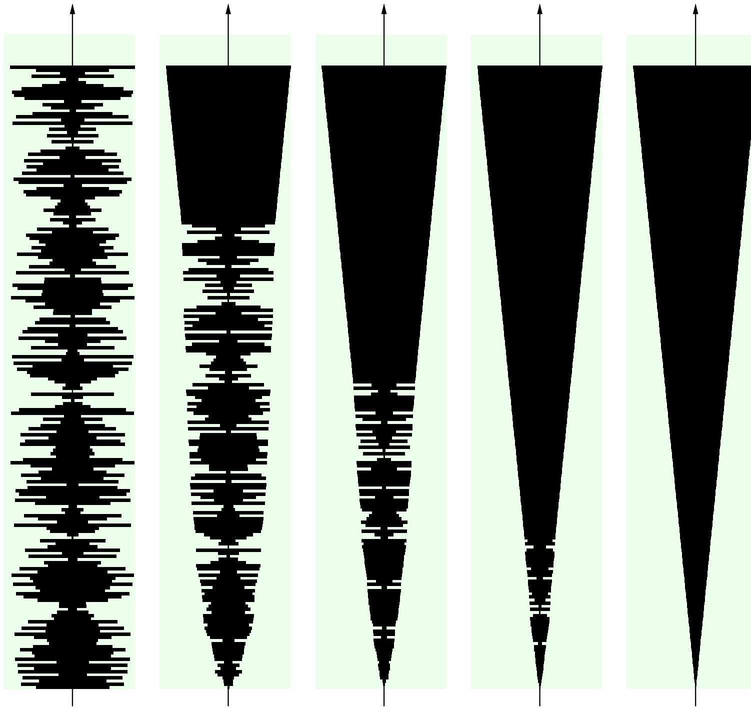


Figura 6.8
Progreso del algoritmo de burbuja.

Ejercicio 6.10 Reorganice el código anterior de forma que el algoritmo esté incluido en su totalidad en una única función `OrdenarBurbuja`. Intente reescribirlo teniendo en cuenta la lógica del algoritmo en lugar de consultar los detalles del código anterior.

Comparación de algoritmos

El problema de determinar el mejor algoritmo de ordenación es muy amplio, como demuestra la gran cantidad de trabajos que se han realizado y que se siguen realizando. Se pueden realizar estudios teóricos —que se salen del objetivo de este tema— y prácticos, es decir, basados en la experimentación con distintos conjuntos de datos.

Los algoritmos que hemos presentado son muy básicos y en general se consideran los más lentos. A pesar de su simplicidad, ya podemos presentar alguna discusión sobre su tiempo de ejecución.

El tiempo de ejecución depende de los datos a ordenar, por lo que si deseamos comparar los tiempos mediante experimentación, tendremos que hacerlo considerando distintos casos. En concreto, podemos estudiar los siguientes:

1. El conjunto de datos está muy desordenado. Incluso podemos suponer el peor caso, es decir, que está totalmente desordenado. Por ejemplo, podemos ordenar de menor a mayor una secuencia de elementos que está ordenada de mayor a menor. En este caso el algoritmo debe realizar la inversión completa del vector.
2. El conjunto de datos está totalmente ordenado. Aunque parezca un caso muy extraño, en la práctica los algoritmos de ordenación podrían recibir una secuencia de datos que ya se encuentra ordenada.

3. El conjunto de datos es aleatorio. En este caso consideramos que no hay un patrón definido en la secuencia de datos. Puede estar más o menos ordenado, por lo que podemos plantear la ejecución de múltiples casos de ordenación para distintas entradas aleatorias.
4. El conjunto de datos contiene muchos repetidos. En este caso podemos tener una secuencia muy grande, pero al tener múltiples repeticiones, realmente tenemos pocos valores.

Si tenemos en cuenta estos casos, y consideramos el código que hemos presentado para los distintos algoritmos, podemos indicar que:

- El algoritmo de selección es muy poco sensible al orden inicial de los datos. Observe que el número de pasos a realizar es muy similar en los casos en que esté totalmente ordenado o desordenado. Las comparaciones de elementos dos a dos serán siempre las mismas, independiente del valor que tengan.
- El algoritmo de inserción es ideal para el caso en que los datos estén prácticamente ordenados. Si observa el código encargado de insertar el siguiente elemento, el subalgoritmo tiene que buscar el sitio adecuado para dejarlo insertado, y en el caso de un vector ya ordenado, ese subalgoritmo es trivial, pues localiza la posición en un solo paso.
- El algoritmo de la burbuja puede llegar a ser muy lento si se dan muchos casos de intercambio de parejas de elementos. En el peor caso, con los elementos ordenados a la inversa, será muy lento.

> **Ejercicio 6.11** Considere el código que se ha presentado para el algoritmo de ordenación por el método de la burbuja. Proponga algún cambio que mejore el caso de una secuencia de datos que ya esté ordenada.

Ordenación indirecta de datos

En esta sección presentamos un ejemplo simple de cómo podemos manejar los datos de forma indirecta, es decir, mediante un referencia a su localización. Más adelante descubrirá la importancia de la indirección, desde programas a bajo nivel, mediante el manejo de direcciones de memoria, hasta complejas estructuras de datos como las que se manejan en un sistema de gestión de bases de datos.

En los algoritmos presentados en las secciones anteriores se han ordenado los elementos de un vector, es decir, se han redistribuido para situarlos de forma ordenada. En algunos casos, se desea obtener el orden de una serie de datos sin modificarlos, es decir, nos interesa determinar el orden dejándolos en el mismo lugar.

Para determinar el orden de los elementos de un vector podemos crear otro vector de índices del mismo tamaño y ordenarlo en su lugar. Para ello, consideraremos que una posición en el vector de índices tiene físicamente el valor `i`, aunque representa lógicamente el elemento *i-ésimo* del vector original. Así, podemos ordenar el vector de índices en lugar del vector original.

En la figura 6.9 se presenta un ejemplo de vector a ordenar, cómo se inicializa el vector de índices, y el resultado final de la ordenación.

> **Ejemplo 6.2.4** Desarrolle una función que determine la ordenación de un vector de objetos **`double`** sin modificarlo. Para ello, deberá devolver un vector de índices que indica el orden de los elementos. Por ejemplo, en la posición cero de este vector se almacenará el índice del elemento más pequeño del vector original, mientras que en el último elemento se almacenará el índice del mayor valor del vector de entrada.

Para resolver el problema usaremos el algoritmo de selección programado en una función que hace uso de `Intercambiar` como módulo auxiliar para intercambiar dos valores enteros.

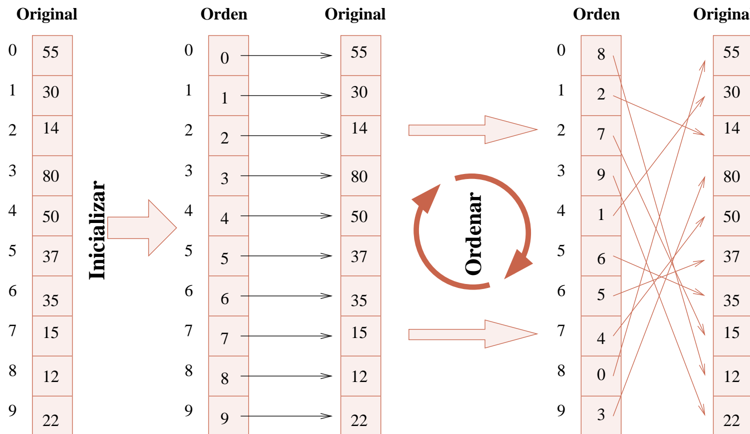


Figura 6.9
Ordenación indirecta de datos.

La solución consiste en declarar un vector `orden` como se muestra en la figura 6.9. El algoritmo es muy parecido al que hemos presentado en las secciones anteriores ya que básicamente consiste en los mismo, es decir, ordenar el vector `orden`.

El detalle fundamental que debemos tener en cuenta es que cuando queremos comparar dos elementos `i, j` del vector `orden`, realmente no debemos comparar los valores en esas posiciones, ya que en esas posiciones lo que realmente se guarda son los índices que indican dónde están los elementos.

El código de esta función podría ser el siguiente:

```
1  vector<int> OrdenarSeleccion(const vector<double> &original)
2  {
3     vector<int> orden(original.size());
4
5     // Inicializar índices
6     for (int i=0; i<orden.size(); ++i)
7        orden[i]= i;
8
9     // Ordenar índices
10    for (int pos=0; pos<orden.size()−1; pos++) {
11       int pmin= pos;
12       for (int i=pini+1; i<orden.size(); i++)
13          if (original[orden[i]]<original[orden[pmin]])
14             pmin = i;
15       Intercambiar(orden[pos],orden[pmin]);
16    }
17
18    return orden;
19 }
```

Si estudia detenidamente la ordenación de los índices, verá que es casi idéntica a la ordenación de un vector, con una única diferencia en la condición de la instrucción **if** de la línea 13. En lugar de comparar los elementos de las posiciones `(i,pmin)`, usamos `original` para poder acceder a los elementos que representan los índices almacenados en esas posiciones.

El resultado es que el intercambio de índices es equivalente a la reordenación de elementos en el vector original, ya que los índices no son más que referencias a la posición de los datos a ordenar.

Fin ejemplo 6.2.4 ■

Ejercicio 6.12 Considere el código que se ha presentado para el algoritmo de ordenación indirecta de datos. Escriba una función que recibe el vector de índices y datos y escribe en la salida estándar la secuencia de elementos ordenados.

Mezcla de secuencias ordenadas

Una operación relativamente frecuente y muy relacionada con la ordenación de datos es la unión de secuencias ordenadas. Imagine que tiene dos archivos independientes de dos subconjuntos de datos ordenados y quisiera tenerlos en un único archivo. Lógicamente, podría unirlos colocando un archivo detrás de otro, pero lo ideal sería procesarlos para tener un archivo ordenado con todos ellos.

Una vez que tenemos el código para ordenar un vector, una solución del problema de unir dos secuencias es ponerlas en un único vector y llamar a un algoritmo de ordenación. Sin embargo, esta solución es muy poco aconsejable ya que desaprovechamos el hecho de tener gran parte del trabajo hecho, por lo que en este caso lo mejor es diseñar un nuevo algoritmo.

El algoritmo de mezcla de secuencias ordenadas consiste en recorrer los dos vectores originales de izquierda a derecha volcando, de forma ordenada y de uno en uno, los elementos en un vector destino.

Básicamente, mantenemos dos índices que controlan la posición por donde vamos en los vectores origen y en cada paso seleccionamos cuál de los dos tenemos que volcar para resolver ese elemento. En concreto, cada iteración consiste en:

1. Comparamos los elementos en dichas posiciones para volcar el más pequeño en el destino.
2. Avanzamos el índice del elemento volcado.

Lógicamente, cuando uno de los vectores se vacía, ya no podemos seguir realizando esa comparación. En este punto, sólo resta volcar el resto de elementos del vector que aún no se ha terminado.

El código podría ser el siguiente:

```
vector<double> Mezclar(const vector<double> &orig1,
                       const vector<double> &orig2 )
{
   vector<double> mezcla (orig1.size()+orig2.size());
   int i1= 0, i2= 0, res= 0;

   // Mientras haya elementos en ambos vectores
   while (i1<orig1.size() && i2<orig2.size()) {
      if (orig1[i1]<orig2[i2]) {
         mezcla[res]= orig1[i1];
         i1++;
      }
      else {
         mezcla[res]= orig2[i2];
         i2++;
      }
      res++;
   }

   while (i1<orig1.size()) { // volcar los que queden de orig1
      mezcla[res]= orig1[i1];
      i1++;
      res++;
   }

   while (i2<orig2.size()) { // volcar los que queden de orig2
      mezcla[res]= orig2[i2];
      i2++;
      res++;
   }

   return mezcla;
}
```

Observe que después del primer bucle aplicamos directamente dos bucles para volcar el resto de elementos de los vectores. Es probable que su primera opción hubiera sido escribir una instrucción condicional para saber a qué vector le quedan elementos. Sin embargo, la condición de ese condicional sería idéntica a la de los bucles. Realmente, el programa pasa por los dos bucles, pero en uno de ellos no entrará ni una sola vez.

6.3 Vectores dinámicos

Uno de los aspectos más importantes —si no el que más— que hace al tipo vector de la STL tan potente es su naturaleza dinámica, es decir, ha sido creado de forma que es muy simple y eficaz cambiar el tamaño de un vector en tiempo de ejecución.

Observe que en los ejemplos anteriores siempre hemos considerado que el vector tiene un tamaño fijo, el cual podemos consultar con la función `size`. Lógicamente, todos los ejemplos presentados siguen siendo válidos, aunque ahora tenemos más posibilidades, puesto que podemos añadir o eliminar elementos si lo deseamos.

Por ejemplo, una forma de leer una secuencia de *n* elementos en un vector es declarándolo de ese tamaño, y luego asignando cada uno de los valores. El código, recordemos, puede ser el siguiente:

```
int main()
{
   int n;
   cout << "Introduzca el número de calificaciones: ";
   cin >> n;

   vector<double> notas(n);

   for (int i=0; i<notas.size(); ++i) {
      cout << "Introduzca calificación " << i+1 << ": ";
      cin >> notas.at(i);
   }

   // ... Procesamos las n notas...
}
```

Ahora bien, imagine que no sabemos el número total de elementos que vamos a leer. Sería el caso en que no sabemos cuántas notas se leen, por ejemplo, porque el final de la lectura lo determina una última nota de valor negativo. Para resolverlo, podemos crear un vector de notas vacío, sin elementos, y cada vez que leemos una nota válida lo hacemos crecer. El código puede ser el siguiente:

```
int main()
{
   vector<double> notas; // Vector vacío

   double dato;
   cout << "Introduzca una calificación: ";
   cin >> dato;

   while (dato>=0) {
      notas.push_back(dato);  // Añade un elemento al final
      cout << "Introduzca siguiente calificación: ";
      cin >> dato;
   }

   // ... Procesamos las notas.size() notas...
}
```

En este ejemplo podemos observar que:

- La declaración de un vector sin parámetros, sin especificar tamaño, implica un tamaño cero. Por tanto, el vector `notas` no tiene ningún elemento, ninguna posición puede ser usada. Por consiguiente, el valor que devuelve `size` es cero.

- La función *push_back* añade un elemento al final del vector. Esta función provoca un incremento en el tamaño del vector. Después de cada ejecución, el valor que devuelve *size* aumenta en uno.

Es importante que entienda la diferencia entre el acceso al vector con la función *at*, que accede a un elemento que ya existe, y la función *push_back*, que añade un elemento más que antes no tenía el vector. Para ello, reescribamos una nueva versión de la lectura de *n* datos:

```
int main()
{
   int n;
   cout << "Introduzca el número de calificaciones: ";
   cin >> n;

   vector<double> notas;

   for (int i=0; i<n; ++i) {
      cout << "Introduzca calificación " << i+1 << ": ";
      cin >> notas.at(i); // Error, no existe
   }
}
```

En este ejemplo se ha cometido un error al acceder a la posición *i*, ya que el vector está vacío, con cero elementos, y por tanto no podemos referirnos a ningún elemento con la función *at*. Antes del bucle tenía cero elementos, y después del bucle seguiría teniendo cero elementos, puesto que la función *at* no cambia el tamaño.

Para solucionarlo, podemos modificar el bucle con el siguiente código:

```
for (int i=0;i<n;++i) {
   double dato;
   cout << "Introduzca calificación " << i+1 << ": ";
   cin >> dato;
   notas.push_back(dato);
}
```

En este caso, en cada iteración se añade un elemento al final del vector, aumentando su tamaño en uno.

Para que entienda mejor la diferencia, considere la siguiente versión errónea, y estudie el comportamiento que tendría:

```
int main()
{
   int n;
   cout << "Introduzca el número de calificaciones: ";
   cin >> n;

   vector<double> notas(n);

   for (int i=0; i<notas.size(); ++i) {
      double dato;
      cout << "Introduzca calificación " << i+1 << ": ";
      cin >> dato;
      notas.push_back(dato);
   }

   // ... Procesamos las n notas...
}
```

Ejercicio 6.13 ¿Por qué el código anterior no es válido para leer un vector de *n* calificaciones?

Por otro lado, al igual que podemos añadir un elemento al final de un vector, incrementando el tamaño, podemos eliminar el último de los elementos. Se realiza con la función *pop_back*, que no tiene parámetros. Lógicamente, esta operación tiene sentido sólo en el caso en que haya elementos en el vector.

Es importante darse cuenta de que esta operación está diseñada para eliminar el último de los elementos, es decir, el que está en la posición `size()-1`. Por ejemplo, si queremos eliminar el elemento más pequeño de un vector de notas como el leído anteriormente, respetando el orden en que quedan el resto de elementos, podemos hacer lo siguiente:

```
int main()
{
   // ... Leemos número de datos n

   vector<double> notas(n);

   // ... Leemos cada una de las notas

   // Localizar posición del mínimo
   int pos_minimo= 0; // posición del elemento más pequeño
   for (int i=1; i<notas.size(); ++i) {
      if (notas[i]<pos_minimo)
         pos_minimo= i;
   }

   // Desplazamos todos los elementos a la izquierda
   for (int i=pos_minimo+1; i<notas.size(); ++i)
      notas[i-1]= notas[i];

   // Descartamos el último elemento
   notas.pop_back();
   // ...
}
```

donde hemos reemplazado parte del código con comentarios para poder centrar el contenido en la parte que resuelve el problema. En la solución propuesta hemos presentado tres pasos:

1. Localizamos la posición del mínimo. En este punto es interesante observar que nos interesa la posición, no el valor del mínimo. El siguiente código localiza el mínimo:

   ```
   int main()
   {
      // ...
      // Localizar valor del mínimo
      double minimo= notas[0]; // Valor del mínimo
      for (int i=1; i<notas.size(); ++i) {
         if (notas[i]<minimo)
            minimo= notas[i];
      }
      // ...
   }
   ```

   pero sería inútil para nuestro problema, ya que estamos interesados en la posición, no en el valor.
2. Desplazamos los elementos. Ahora pasamos uno a uno los elementos hacia atrás. Cada elemento reemplaza justo al que tiene a la izquierda. En este caso hay que tener cuidado con los índices, pues tenemos que empezar a la derecha del mínimo y terminar en el tamaño menos uno. Por ejemplo, podríamos cambiar los índices como sigue:

   ```
   int main()
   {
      // ...
      // Desplazamos todos los elementos a la izquierda
      for (int i=pos_minimo; i<notas.size()-1; ++i)
         notas[i]= notas[i+1];
      // ...
   }
   ```

   con el mismo efecto. En esta versión habría que tener especial cuidado con el final, ya que no podemos salirnos por la derecha. Proponemos la primera versión, donde nos ahorramos una resta en la evaluación de la condición.
3. Eliminamos el último elemento. Después de desplazarlos, el tamaño del vector sigue siendo el mismo. Aunque el mínimo ha sido reemplazado, al final tenemos repetido el último elemento. Para terminar tendremos que ejecutar `pop_back` que elimina la última posición.

En la figura 6.10 se puede ver gráficamente el efecto sobre un vector que contiene 8 objetos de tipo **double**.

Figura 6.10
Eliminar el elemento más pequeño.

Ejemplo 6.3.1 Escriba una función que reciba un vector de números reales y elimine los repetidos, manteniendo sólo la primera instancia de cada elemento en el mismo orden que en el vector original.

El algoritmo puede implementarse recorriendo cada uno de los elementos del vector de forma que si tiene un elemento en una posición anterior, podemos eliminarlo. El código puede ser el siguiente:

```
void ComprimirSinRepetidos (vector<double>& v)
{
   int i= 1;
   while (i<v.size()) {
      bool repetido= false;
      for (int j=0;j<i && !repetido;++j) // Está antes?
         if (v.at(i)== v.at(j))
            repetido= true;
      if (repetido) { // i hay que eliminarlo
         for (int j=i+1;j<v.size();j++)
            v.at(j−1)= v.at(j);
         v.pop_back();
      }
      else i++; // Avanzar sólo si no está repetido
   }
}
```

Observe que el bucle principal se ha implementado con un **while**, en lugar de un bucle **for**. Se ha realizado para evitar confundir al lector, ya que un bucle **for** podría sugerir que la variable debe aumentar en uno para cada iteración, cuando no tiene que ser así. Si estudia el cuerpo del bucle, observará que en caso de que el elemento esté repetido no es necesario aumentar el índice, ya que todos los elementos se han desplazado a la izquierda, y ya estamos situados en el siguiente a estudiar.

Fin ejemplo 6.3.1 ■

6.3.1 Eficiencia y vectores dinámicos

Las operaciones que afectan al tamaño de un vector pueden ser poco eficientes. En principio, debemos centrarnos en crear algoritmos simples, claros, y fáciles de mantener. A pesar de ello, en la práctica puede resultar muy conveniente —si no necesario— considerar qué tiempos de ejecución implican para optar por los mejores algoritmos.

El tipo vector, en general, puede ser muy eficiente, ya que una vez que tiene un tamaño determinado, el acceso a cada uno de sus elementos —para modificar o para consultar— es un acceso muy rápido. Sin embargo, operaciones que añaden o eliminan elementos, es decir, operaciones que cambian su tamaño, pueden resultar lentas.

A pesar de todo, las operaciones `push_back` y `pop_back` se pueden implementar de una forma muy eficaz, de forma que los códigos que hemos presentado anteriormente y que realizan múltiples ejecuciones, incluso miles o millones, son códigos relativamente rápidos.

Para evitar la ejecución múltiple de estas operaciones que añaden o eliminan un único elemento, podemos usar las dos siguientes[5]:

- La función `clear` del tipo vector, sin parámetros, permite asignar un tamaño de cero elementos. Es decir, después de llamar a esta función, `size` devolverá cero. Observe que con ese tamaño no tendrá ninguna posición disponible.
- La función `resize`, a la que se le pasa en valor entero, permite cambiar el tamaño de un vector. Por tanto, después de llamarla con un tamaño n, la función `size` devolverá ese valor. Observe que después de eso, podrá usar sin problemas las posiciones desde la 0 a la $n-1$.

Ejemplo 6.3.2 Considere el ejemplo presentado para eliminar repetidos de un vector. Cada elemento repetido necesita un desplazamiento del resto del vector. Proponga una nueva solución más eficiente.

En el código presentado aparecen dos inconvenientes:

- El más importante se refiere al bucle interno, ya que cada vez que hay un repetido hay que trasladar una secuencia completa de datos hacia la izquierda.
- Cada vez que eliminamos un repetido, llamamos a la función de eliminar el último elemento. Esta operación es muy rápida, mucho más que la de añadir un elemento. En cualquier caso, podemos evitar que se llame muchas veces.

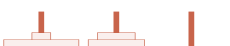
Para mejorar ambos aspectos podemos plantear un nuevo algoritmo. Básicamente, la idea es mantener dos índices en el vector, ambos recorriendo el vector de izquierda a derecha, uno para controlar dónde se copiarán los nuevos elementos (`destino`) y otro para controlar el siguiente elemento a estudiar (`fuente`).

El algoritmo consiste en comprobar si el siguiente elemento a estudiar —que controlamos con la posición `fuente`— es nuevo. Si lo es, se vuelca a la posición controlada con `destino` y avanzamos ambos índices. Si no es nuevo, simplemente se ignora avanzando el índice `fuente`. Inicialmente, ambos índices valen lo mismo, pero conforme aparecen repetidos, el índice que controla el elemento a estudiar irá avanzando y alejándose de la posición de `destino`.

El siguiente código podría ser una implementación de esta idea:

```
void ComprimirSinRepetidosEficiente (vector<double>& v)
{
   int destino= 1; // El primero ya está añadido

   for (int fuente= 1; fuente<v.size(); ++fuente) {
      bool repetido= false;
      for (int j=0;j<destino && !repetido;++j)
         if (v.at(fuente)==v.at(j))
            repetido= true;
```

[5] Adicionalmente, podríamos usar la función `reserve`. Sin embargo, es mejor retrasar su presentación, ya que es un tema avanzado para este momento.



```
        if (!repetido) {
            v.at(destino)= v.at(fuente);
            destino++;
        }
    } // de for

    v.resize(destino); // Descartamos los elementos no copiados
}
```

Observe que la última línea recorta la longitud del vector para que sólo considere los elementos que se han copiado al destino, es decir, los que se han detectado como no repetidos.

Fin ejemplo 6.3.2 ■

Ejercicio 6.14 Considere el código de mezcla de vectores de la sección 6.2.5 (página 131). En él se declara un vector resultado con el tamaño necesario para almacenar la solución. Reescriba la función declarando un vector vacío que va creciendo conforme se añaden elementos al final.

6.4 Matrices

Una matriz nos permite organizar un conjunto de datos de algún tipo *base* en una estructura bidimensional, pudiendo acceder a cada uno de ellos especificando dos índices: la fila y la columna donde se encuentra.

Por ejemplo, imagine que tenemos un conjunto de n estudiantes, con sus correspondientes calificaciones. Como hemos visto anteriormente, podríamos usar un vector de tamaño n para almacenar una calificación por estudiante. Sin embargo, en el caso de que cada estudiante tenga m calificaciones, necesitaríamos almacenar $m \times n$ calificaciones. Para resolverlo, podemos crear un objeto matriz que las almacene todas, asignando una fila por estudiante y una columna por cada calificación.

Por ejemplo, en la figura 6.11 se muestra un ejemplo con 2 filas y 8 columnas. Observe que cada elemento puede ser localizado mediante dos índices. Como en el caso de los vectores, cada uno de los índices empieza en cero.

	0	1	2	3	4	5	6	7
0	5.5	3.0	10.0	4.0	0.0	3.5	6.9	5.0
1	3.5	2.0	16.0	1.0	0.5	4.2	7.9	1.0

Tipo base: double **Tamaño:** 2 filas y 8 columnas

Figura 6.11
Ejemplo de matriz de tamaño 2×8.

La STL no ofrece un tipo de dato especializado para manejar matrices, a pesar de que es una estructura muy importante para simplificar la implementación de múltiples algoritmos. La razón fundamental para ello es que el lenguaje C++ ya ofrece el tipo matriz que también aparece en lenguaje C, y en caso de querer ampliar las posibilidades que tiene, siempre podemos usar el tipo vector de la STL.

Recordemos que el tipo vector es una estructura unidimensional donde hemos organizado los elementos de algún tipo base mediante un índice. Si deseamos implementar una matriz, con dos índices, podemos obtenerla simplemente creando un vector de vectores.

En esta sección vamos a estudiar cómo podemos usar el tipo vector para representar matrices y cómo resolver problemas relacionados. En la práctica, C++ nos permite resolver el problema de distintas formas, aunque nuestro interés es trabajar con matrices haciendo uso de una solución

simple y potente —como la propuesta en este tema— lo que nos permitirá también afianzar los conocimientos sobre vectores.

6.4.1 Declaración

Si deseamos declarar un objeto que contenga los elementos organizados de forma matricial, mediante dos índices, podemos declarar un vector cuyo tipo base sea también un vector. Por ejemplo, si deseamos una matriz *m* de números reales —de tipo **double**— declaramos un vector de vectores de **double**. La declaración de *m* puede hacerse como sigue:

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

int main()
{
   vector<vector<double> > m;
   // ...
}
```

Si analiza detenidamente la línea de declaración observará que simplemente declaramos un vector siguiendo las reglas sintácticas que hemos presentado en las secciones anteriores: escribiendo el tipo base entre los caracteres <>. Lógicamente, este tipo base se especifica como un vector, ahora de tipo **double**, lo que implica que vuelven a aparecer los caracteres <>.

El único detalle importante que tenemos que indicar se refiere al espacio extra que hay entre los dos últimos signos de cierre (>) antes del nombre de la variable. Este espacio es necesario para que el compilador de C++ interprete correctamente el tipo[6]. Si los dos caracteres aparecen sin separación, el compilador interpreta el operador >>, que ya hemos usado por ejemplo para leer datos desde **cin**, pero que no corresponde en esta declaración.

El resultado de la declaración es un objeto vacío; es decir, tiene un tamaño de cero elementos, como corresponde a un vector declarado sin especificar ningún parámetro de tamaño. Las operaciones válidas con este objeto son las mismas que ya hemos presentado, por ejemplo, añadir vectores al final (incrementando el tamaño en uno). Si queremos que *m* tenga un tamaño inicial, podemos declarar que contiene un número de elementos como sigue:

```
int main()
{
   vector<vector<double> > m (10);

   cout << v.size()      << endl; // Escribe 10
   cout << v[5].size()   << endl; // Escribe 0
   cout << v[10].size()  << endl; // Error: vector no existe
   cout << v[5][2]       << endl; // Error v[5] está vacío
   // ...
}
```

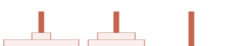
donde podemos ver que la matriz *m* tendrá inicialmente `10` vectores. Es decir, el valor que devuelve la operación *size* será `10`, y cada elemento `m[i]` (con `0<=i<=9`) que contiene es un objeto de tipo **vector** de **double**.

Observe que aunque la hemos declarado de un tamaño `10`, no contiene ningún elemento. La línea que declara *m* crea una matriz con `10` filas, pero cada fila tiene `0` elementos.

Es posible que ya esté considerando la posibilidad de usar la función *push_back* para añadir vectores con elementos al final de *m*. Efectivamente, podemos hacerlo sin problemas. Por ejemplo, imagine que deseamos una matriz de 10 filas por 5 columnas, podríamos hacer:

```
int main()
{
   vector<double> aux(5);
   vector<vector<double> > m;
```

[6] Este detalle ya no es necesario en C++11, pero si queremos que nuestro código sea también compatible con C++98, será necesario separarlos.



```
    for (int i=0; i<10; ++i)
        m.push_back(aux);

    // ...
}
```

que nos permite obtener una matriz m de tamaño 10×5.

En la práctica, un programador experimentado simplificaría en gran medida este código. Para ello, tengamos en cuenta que en la declaración de un vector podemos añadir un segundo parámetro que corresponde al valor inicial de todos los elementos del vector. Así, el código anterior se podría escribir:

```
int main()
{
    vector<double> aux(5);
    vector<vector<double> > m(10, aux);
    // ...
}
```

donde hemos indicado que `m` debe tener un tamaño de `10` elementos, todos ellos inicializados con el valor del objeto `aux`. Observe que podríamos crear la matriz inicializada con ceros como sigue:

```
int main()
{
    vector<double> aux(5, 0.0); // 5 double's inicializados con 0.0
    vector<vector<double> > m(10, aux); // 10 vectores inicializados con aux
    // ...
}
```

Finalmente, incluso podemos eliminar el objeto `aux`, ya que podemos indicar al compilador que queremos crear un objeto de tipo vector simplemente con el nombre del tipo seguido con los parámetros entre paréntesis. En concreto, podemos hacer:

```
int main()
{
    vector<vector<double> > m(10, vector<double>(5,0.0) );
    // ...
}
```

Observe que el segundo parámetro para crear `m` es idéntico a la declaración de `aux` aunque sin el nombre de la variable[7].

6.4.2 Matrices rectangulares

En general, cuando se habla de una matriz se considera un objeto bidimensional, con dos índices, con un número de elementos igual al producto de número de filas por número de columnas. En otra palabras, cada fila tiene exactamente el mismo número de elementos. Por ejemplo, en la figura 6.11 —página 137— se mostraba una matriz rectangular.

Ejemplo 6.4.1 Escriba un programa para leer una serie de m notas para cada uno de los n alumnos a procesar. Para ello, almacene todas la notas en una matriz para finalmente escribir un informe con todas ellas. Este informe imprime, en cada línea, las notas de cada alumno seguidas de su media. Finalmente escribe la media global de todas ellas.

Una solución para este problema, aunque con un diseño no muy adecuado, podría ser la siguiente:

[7] Aunque no es relevante en este tema, es interesante indicar que esta sintaxis no es algo especial para el tipo vector, realmente podemos generar un objeto de cualquier tipo simplemente escribiendo el nombre del tipo seguido, entre paréntesis, con los parámetros con los que se debería inicializar.

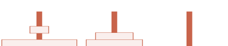
Listado 7 — Archivo *alumnosYnotas.cpp*.

```
#include <iostream> // cout, cin, endl
#include <vector>   // vector
using namespace std;

int main()
{
   int alumnos, ndatos;

   cout << "Introduzca el número de alumnos: ";
   cin >> alumnos;

   cout << "Introduzca el número de notas para cada uno: ";
   cin >> ndatos;

   if (alumnos>0 && ndatos>0) {
      vector<vector<double> > notas(alumnos, vector<double>(ndatos) );

      // Lectura de datos
      for (int i=0; i<alumnos; ++i) {
         for (int j=0; j<ndatos; ++j) {
            cout << "Introduzca nota " << j+1 << " del alumno " << i+1 <<
                 ": ";
            cin >> notas[i][j];
         }
      }

      // Cálculo de medias
      vector<double> medias;
      double media_global= 0.0;

      for (int i=0; i<alumnos; ++i) {
         double media= 0.0;
         for (int j=0; j<ndatos; ++j)
            media+= notas[i][j];
         medias.push_back(media/ndatos);
         media_global+= media/ndatos;
      }

      media_global= media_global/alumnos;

      // Informe final
      for (int i=0; i<alumnos; ++i) {
         cout << "Alumno " << i+1 << ":";
         for (int j=0; j<ndatos; ++j)
            cout << ' ' << notas[i][j];
         cout << '.' << " Media: " << medias[i] << endl;
      }
      cout << "Media global: " << media_global << endl;
   }
}
```

Fin ejemplo 6.4.1 ■

Ejercicio 6.15 La solución anterior refleja un mal diseño, ya que hemos incluido todo el código —que realiza varias tareas independientes— en la función `main`. Proponga una mejor solución. Para ello tenga en cuenta que:

1. Se puede crear una solución para la entrada de datos y otra para la salida.
2. Se puede crear una solución para calcular las medias. Para ello, tenga en cuenta que podemos escribir una función para calcular la media de un vector de `double` y usarla repetidamente.



6.4.3 Matrices no rectangulares

La versatilidad que nos ofrece el tipo vector para gestionar una estructura bidimensional nos permite resolver problemas donde tenemos un número de elementos distinto para cada fila.

Si revisamos la literatura sobre matrices, descubrimos que la gran mayoría de los problemas se plantean con matrices rectangulares. Realmente, cuando nos referimos a casos con un número variable de elementos por fila sería más correcto evitar el término matriz —hablar por ejemplo de *estructuras bidimensionales no rectangulares*— ya que el término matriz sugiere que la estructura es rectangular, es decir, que el número de columnas es el mismo para todas las filas.

Si revisa otras referencias bibliográficas, tenga en cuenta que el término matriz generalmente se usa para una estructura rectangular de $f \times c$ elementos. En esta sección hemos abusado del término matriz para enfatizar el hecho de que el tipo que usamos es el mismo, es decir, un vector de vectores.

Por ejemplo, imagine que deseamos almacenar las temperaturas mínimas alcanzadas para cada día de un determinado año. Una posibilidad sería almacenarlas con un vector de valores reales con una posición por cada día del año. Sin embargo, si deseamos almacenar los valores de cada mes en un vector independiente podemos usar un vector de vectores de valores `double`. Así, el resultado sería una matriz de 12 filas por un número variable de columnas, ya que cada mes tiene un número de días variable.

Ejemplo 6.4.2 Escriba una función que recibe una matriz de datos `double` y devuelve si es rectangular.

Para saber si una matriz es rectangular sólo tenemos que comprobar que el tamaño de las filas es el mismo. Para ello, es suficiente comprobar, para cada fila, si el tamaño es idéntico a la anterior. El código podría ser el siguiente:

```
bool Rectangular (const vector<vector<double> >& matriz)
{
   bool rectangular= true;

   for (int i=1; i<matriz.size() && rectangular; ++i)
      if (matriz[i].size()!=matriz[i-1].size())
         rectangular= false;

  return rectangular;
}
```

Observe que hemos incluido el valor booleano en la condición del bucle a fin de que el algoritmo termine al encontrar una fila de distinto tamaño, sin necesidad de seguir comprobando el resto de la matriz.

Fin ejemplo 6.4.2 ■

Ejercicio 6.16 Escriba una función que recibe un vector de valores reales que indican las calificaciones de una serie de estudiantes, y devuelve una matriz de 10 filas. La primera fila contiene los índices de las calificaciones c tal que $c < 1$, la segunda los índices de las calificaciones que cumplen $1 \leq c < 2$, y así hasta la décima, que contiene los índices de las calificaciones que cumplen $9 \leq c \leq 10$.

Ejemplo: Ordenación por casilleros (bucket sort)

El ordenamiento por casilleros (en inglés *bucket sort* o *bin sort*) es un buen ejemplo para mostrar el uso de matrices no rectangulares. En gran medida es una idea simple e intuitiva, ya que nosotros mismos podemos aplicarla en tareas de ordenación habituales. Por ejemplo: si tiene que ordenar una lista muy grande de nombres de personas “a mano” —imagine un montón de documentos con un nombre en cada uno— una forma de actuar para evitar la dificultad de manejar muchos datos es

pre-ordenar los elementos separándolos en distintos subconjuntos; por ejemplo, podemos separar los documentos según la primera letra del nombre.

La ordenación por casilleros consiste en particionar el conjunto de datos a ordenar en subconjuntos, ordenar cada uno de esos subconjuntos y finalmente unirlos. Cada uno de los subconjuntos se ordena de forma independiente —por lo que se puede realizar más eficientemente— para finalmente encadenarlos en la solución final.

Ejemplo 6.4.3 Escriba una función que ordene un vector de números reales correspondientes a la altura (en metros) de un conjunto de personas.

Para resolver el problema creamos 30 casilleros para insertar los números que corresponden a las alturas, particionadas cada 10 centímetros. La solución consiste en crear un vector de casilleros —un vector de vectores— y aplicar el siguiente algoritmo:

1. Volcar cada número a su correspondiente casillero. Para eso simplemente dividimos entre 0.1 (es decir, 10 cm) para calcular el casillero donde corresponde y lo añadimos al final.
2. Ordenamos cada casillero. En este caso hemos escogido el algoritmo de inserción. Es una elección que podría cambiarse incluso por otro algoritmo de tipo *bucket sort*.
3. Volcamos uno a uno los casilleros al vector original. Consiste en concatenar todos los elementos que hay en los casilleros uno detrás de otro en el vector original.

El código podría ser el siguiente:

```
void OrdenarAltura (vector<double>& v)
{
   vector<vector<double> > buckets(30); // 30 * 0.1 metros

   // Volcamos a los casilleros
   for (int i=0; i<v.size(); ++i) {
      int b= v[i]/0.1; // Cada 10 centímetros
      buckets[b].push_back(v[i]);
   }

   // Ordenamos casilleros. Por ejemplo, con inserción.
   for (int b=0; b<buckets.size(); ++b)
      OrdenarInsercion(buckets[b]);

   // Volcamos de nuevo en v
   int dest=0; // posición a sobreescribir
   for (int b=0; b<buckets.size(); ++b)
      for (int i=0; i<buckets[b].size(); ++i) {
         v[dest]= buckets[b][i];
         dest++;
      }
}
```

Observe que hemos usado una solución muy ligada al problema que queremos resolver. Así, sabemos que las alturas de las personas no van a llegar a los 3 metros, y que una partición en casilleros de 10 centímetros nos permite dividir el problema en distintos subconjuntos.

Ejercicio 6.17 Modifique la función anterior para hacerla más genérica. Para ello, tenga en cuenta los números almacenados pueden estar en cualquier rango y que la función también recibe como parámetro el número de casilleros que queremos usar.

Finalmente es interesante notar que que este algoritmo es más eficiente si permite dividir mejor los elementos en los distintos casilleros. Por ejemplo, si todos los elementos fueran de personas más o menos de la misma altura no conseguiríamos más que unos pocos casilleros con la mayoría de los elementos. El hecho de que haya muchos casilleros vacíos o con pocos elementos empeora el rendimiento de la función. Más adelante volveremos sobre este algoritmo para estudiar otras posibilidades.

Fin ejemplo 6.4.3 ■

6.5 Problemas

Problema 6.1 Escriba una función que recibe un vector de caracteres y una letra y que devuelve cuántas veces aparece esta letra en dicho vector.

Problema 6.2 Escriba un programa que lea un texto desde la entrada estándar y que nos diga cuántas veces aparece cada letra del alfabeto inglés. Tenga en cuenta que:

- El texto se lee carácter a carácter. Puede llegar a ser muy largo, por lo que no se almacenará todo el texto, sino que se irá procesando conforme se lee.
- La entrada de datos finaliza con el carácter `'#'`.
- No se diferenciarán mayúsculas y minúsculas.

Problema 6.3 Escriba un programa que lea una secuencia de números enteros positivos y que imprima un análisis de frecuencias de los números que aparecen. Este análisis consiste en un listado de parejas de números: el primero será el entero que aparece en la secuencia y el segundo el número de repeticiones. Además, el listado estará ordenado por el primer valor.

Problema 6.4 Escriba un programa que permita descomponer en números primos un número indeterminado de enteros positivos menores que un valor máximo M. El programa terminará cuando se pida la descomposición del número cero. Para resolverlo, incluya dos funciones:

1. Una función que calcula un vector con todos los números primos hasta el número M.
2. Una función que calcula la descomposición de un número entero haciendo uso de un vector de primos precalculado.

Problema 6.5 Escriba un programa que permita lanzar los distintos algoritmos de ordenación para distintas secuencias de elementos, considerando el peor caso, mejor caso, números aleatorios, y pocos datos con múltiples repeticiones.

Problema 6.6 Escriba un programa que lea una matriz rectangular de números reales y calcule la posición del mayor elemento.

Problema 6.7 Escriba un programa que lea dos matrices A, B bidimensionales rectangulares de números reales y realice la suma algebraica. Recuerde que dicha suma es posible sólo si las matrices tienen idénticas dimensiones y que el resultado es una matriz C con las mismas dimensiones en la que el valor de un elemento en la posición (i, j) corresponde a la suma de los dos elementos en la misma posición de las matrices de entrada. Es decir:

$$c_{i,j} = a_{i,j} + b_{i,j}$$

Problema 6.8 Escriba un programa que lea dos matrices $A_{N\times L}, B_{L\times M}$ bidimensionales rectangulares de números reales y realice la multiplicación algebraica. Recuerde que dicha multiplicación es posible sólo si el número de columnas de la primera matriz coincide con las filas de la segunda. El resultado es una matriz $C_{N\times M}$ en la que:

$$c_{i,j} = \sum_{k=1}^{L} a_{i,k} \cdot b_{k,j}$$

Problema 6.9 Escriba un programa que calcule la traspuesta de una matriz rectangular $A_{N\times M}$ de números reales. Recuerde que la traspuesta en una matriz $B_{M\times N}$ —note las dimensiones “cambiadas”— de forma que el elemento de la posición (i, j) es el que se sitúa en la posición (j, i) de la original.

7 Cadenas de la STL

Introducción 145
Tipo string 146
El tipo char: codificaciones 156
Problemas 162

7.1 Introducción

El tipo de dato cadena es fundamental en programación, ya que es el que usaremos para procesar texto. Una cadena de caracteres es una secuencia de caracteres, es decir, de datos de tipo **`char`**.

Aunque podríamos comenzar la discusión en el problema de la codificación de los caracteres, dando lugar a comentarios sobre las distinta lenguas que podrían procesarse y terminando concluyendo que el texto debería ser una secuencia de un tipo más complejo que un simple **`char`**, nosotros simplificaremos todo esto limitándonos al caso de que una letra se almacene en un único **`char`**.

Puede considerar que la codificación corresponde a la tabla *ASCII* extendida. En la última parte del tema volveremos sobre este problema, dando algunos detalles que aclaren el problema de la codificación, pero sin abordar implementaciones más generales como corresponde a un curso de fundamentos de programación.

Si consideramos el problema de procesar cadenas de caracteres, una posible solución podría ser el uso de un vector de caracteres, es decir, usar el tipo **`vector`** de la STL con el tipo **`char`** como tipo base. Realmente serviría como solución, aunque en la práctica las cadenas de caracteres son un tipo de dato muy importante, por lo que es conveniente crear un tipo especialmente para ello —el tipo **`string`**— añadiendo funcionalidad específica para problemas de procesamiento de texto.

De hecho, en la práctica, el tipo de dato **`string`** tiene mucho en común con el tipo **`vector`**, ya que no es más que una secuencia de objetos de tipo **`char`**, que además vamos a manejar también con un índice. Es decir, el tipo de dato **`string`** es un tipo de dato compuesto homogéneo que *contiene* objetos de tipo **`char`**.

Tenga en cuenta que C++ ofrece la posibilidad de definir cadenas con un tipo de dato más “primitivo” definido dentro del lenguaje. Este tipo de dato es el que se “hereda” desde el lenguaje C, por lo que lo denominaremos *cadenas-C*. Es un tipo de dato más básico, más cercano a la máquina, y que requiere un especial cuidado, ya que será el programador el que se encargue de resolver los problemas derivados de las limitaciones del tipo.

El tipo **string** de la STL nos permite manejar cadenas evitando las particularidades de las *cadenas-C* y añadiendo nuevas características que lo hacen más versátil y potente. A continuación se estudiará el tipo **string** que ofrece C++ como parte de la STL.

7.2 Tipo string

El tipo de dato **string** es un tipo de dato diseñado para manejar cadenas de caracteres, es decir, secuencias de cero o más objetos de tipo **char**. El número de caracteres que componen la secuencia es el tamaño de la cadena. Una cadena vacía es una cadena de tamaño cero.

Como ocurría con el tipo de dato **vector**, el tipo **string** de la STL no está integrado en el lenguaje, sino que se ha definido como una clase de la STL, por lo que será necesario incluir el archivo de cabecera **string** antes de poder usarlo.

7.2.1 Declaración

La declaración de un objeto puede realizarse como cualquier declaración de los tipos básicos, indicando el tipo seguido por el nombre de la variable:

string *nombre_objeto*;

Además, es posible declarar un objeto e inicializarlo. Para ello tendremos que especificar un valor inicial como una secuencia de caracteres delimitada por dobles comillas. Por ejemplo, en el siguiente listado se presenta una declaración del objeto *nombre* —de tipo **string**— con un valor inicial:

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

int main()
{
    string nombre ("Bjarne Stroustrup");
    // ...
}
```

En este listado hemos incluido no sólo la línea de declaración, sino que hemos añadido otras líneas para enfatizar que:

1. Es necesario incluir el archivo de cabecera **string**. Recordemos que el tipo no está integrado en el lenguaje, por tanto, debemos incluir este archivo para poder disponer de él.
2. El tipo **string** está incluido dentro del estándar y, por tanto, es necesario acceder a través de **std**. Si en el ejemplo anterior no hubiéramos especificado el espacio de nombres **std**, el compilador no hubiera reconocido el tipo **string**.

7.2.2 Literal de tipo string

Desde un punto de vista formal, en C++98 no es posible indicar literales de tipo **string**, ya que no está integrado en el lenguaje[1]. Es decir, el lenguaje no va a reconocer directamente un literal como en el caso de los tipos básicos. Sin embargo, es posible usar la sintaxis que nos ofrece la clase **string** para poder indicar un valor concreto como si de un literal se tratara.

Para indicar un valor concreto del tipo **string**, simplemente tenemos que usar la palabra **string** seguida, entre paréntesis, de un valor concreto entre comillas dobles. Por ejemplo, el siguiente código:

[1] En el último estándar ya se han incluido herramientas que permiten hablar de literales de un tipo definido como clase.

string("hola")

corresponde a un “literal” de tipo **string** que contiene una secuencia de caracteres de longitud 4, en concreto con los objetos de tipo **char** ’h’, ’o’, ’l’, y ’a’.

Finalmente, es importante indicar que esta forma de crear un objeto como literal no es más que un método para ofrecer una solución homogénea similar a los tipos básicos del lenguaje. Desde un punto de vista formal, realmente no debería llamarse literal. Sin embargo, lo usaremos por ahora como tal, aunque volveremos a visitar brevemente este aspecto más adelante cuando abordemos el estándar C++11 y las ampliaciones que se incluyen para la clase **string**, especialmente cuando estudie el problema de la codificación y las distintas alternativas que incluye en lenguaje.

7.2.3 El tipo *string* como contenedor de *char*

En primer lugar, es conveniente destacar el tipo **string** como un contenedor de objetos de tipo **char**, muy similar al tipo **vector**. Como es lógico, este diseño se ve reflejado con una interfaz de operaciones parecida a la de este tipo. En concreto, las siguientes operaciones, que hemos visto para el tipo **vector**, también pueden usarse para el tipo **string**:

- Tamaño de la cadena. Podemos usar la operación *size* para consultar la longitud de la cadena.
- Acceso a cada carácter con [] y *at*. Podemos referirnos a cada carácter usando estos operadores. Se usan de manera idéntica al tipo **vector**, pudiendo tomar valores que van desde el cero al tamaño menos uno. Recuerde que *at* comprueba que el índice es una posición válida, mientras que los corchetes ignoran la comprobación para conseguir una operación más rápida.
- Añadir o eliminar caracteres al final. También podemos cambiar el tamaño añadiendo o eliminando elementos al final, con las operaciones *push_back* y *pop_back*.

Por tanto, si sabe usar el tipo **vector**, ya conoce un conjunto de operaciones que tienen el mismo efecto sobre el tipo **string**, como si de un vector de objetos **char** se tratara.

Ejercicio 7.1 Considere el siguiente trozo de código con dos bucles sobre un objeto *cadena* de tipo **string**. ¿Cuáles son los errores que encuentra? Indique brevemente el efecto que podrá tener su ejecución.

```
for (int i=1; i<=cadena.size(); i++)
   cout << cadena.at(i);
for (int i=1; i<=cadena.size(); i++)
   cout << cadena[i];
```

7.2.4 E/S de objetos string

El tipo vector no incluye operaciones de E/S. Si deseamos realizar la lectura (escritura) de un vector de objetos, será necesario leerlos (escribirlos) de uno en uno. Este comportamiento era de esperar, ya que el lenguaje no puede asumir nada sobre la intención de un vector de objetos y, por tanto, será el programador con un algoritmo adecuado quien determine la forma en que se realiza la E/S. Por ejemplo, si el programador quiere escribir los datos que contiene un vector de objetos de tipo **double**, tendrá que decidir si quiere escribirlos separados con espacios, con saltos de línea, con algún carácter como la coma, etc.

Sin embargo, el sistema de E/S de C++ incluye operaciones que facilitan la lectura y escritura de objetos de tipo **string**. En este caso no existe ambigüedad, ya que es de esperar que la lectura sea de una secuencia de caracteres y la escritura sea la salida de cada uno de los **char** que componen la cadena.

E/S con los operadores < < y > >

La lectura de un objeto **string** con el operador de entrada >> se realiza de forma similar a los tipos de datos básicos. El operador de entrada asume que los datos se encuentran separados por "espacios blancos" (separadores) y por tanto la entrada de dicho objeto se realiza como sigue:

1. Descartar todos los separadores que se encuentren en la entrada hasta que lleguemos a un carácter no separador.
2. Cargar en el objeto **string** todos los caracteres que hay en la entrada hasta llegar al siguiente separador. Los caracteres se añaden al final del **string** en el mismo orden en que se extraen de la entrada.

En el siguiente programa realizamos la lectura de una cadena en un objeto **string** y a continuación la mostramos en pantalla.

```
#include <iostream> // para cin, cout, endl...
#include <string>   // para string
using namespace std;

int main()
{
   string s;
   cout << "Introduzca cadena: ";
   cin >> s;
   cout << "Cadena: " << s << endl;
}
```

Pruebe a ejecutar el programa para entender cómo funciona. Para ello, incluya ejemplos en los que la cadena introducida tiene espacios y saltos de línea iniciales, espacios después de introducir alguna palabra, o varias palabras.

> **Ejercicio 7.2** Escriba un programa que lea dos cadenas (*c1* y *c2*) y cuente el número de apariciones de cada una de las letras de *c1* en *c2*.

E/S con la función *getline*

Es habitual que en las cadenas de caracteres que lee un programa se incluyan caracteres separadores. Por ejemplo, si leemos el nombre de una persona, podemos incluir nombre y apellidos separados por espacios. Además, no podemos asumir que es un nombre y dos apellidos (3 cadenas) ya que pueden ser compuestos. Si queremos leer el nombre completo de una persona, una buena opción sería leer toda la cadena como una línea completa.

Para poder procesar la entrada como líneas, leyendo todos los caracteres incluyendo separadores, es posible usar la función `getline`. Un ejemplo de su uso es:

```
#include <iostream> // para cin, cout, endl...
#include <string>   // para string, getline
using namespace std;

int main()
{
   string nombre, direc;

   cout << "Introduzca un nombre: ";
   getline (cin, nombre);

   cout << "Introduzca dirección: ";
   getline (cin, direc);

   cout << "Nombre: " << nombre << endl;
   cout << "Dirección: " << direc << endl;
}
```

En este programa, las dos cadenas leídas pueden contener cualquier carácter excepto el salto de línea[2], ya que es el que determina el final de la entrada para cada una de ellas.

Al incluir la función `getline` que no procesa la entrada como hasta ahora —no tiene en cuenta que los *espacios blancos* son separadores y no los descarta— nos obliga a reflexionar sobre cómo se comporta el sistema de E/S.

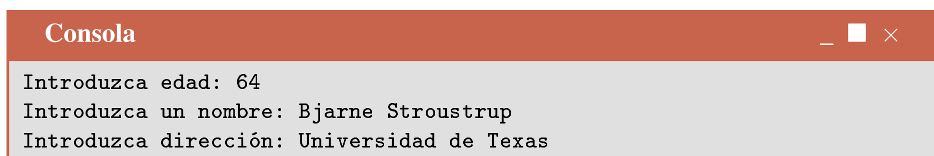
Ejemplo 7.2.1 Escriba un programa que lea y escriba la edad, nombre y dirección de una persona.

En este ejemplo es importante recordar los detalles que se estudiaron sobre cómo funciona la E/S de tipos básicos (véase página 34). La lectura de un tipo básico se realiza primero descartando separadores y luego cargando el tipo básico a partir de la entrada, terminando justo cuando un carácter deja de formar parte de la entrada. Así, si leemos un entero, primero descartamos separadores y cuando encontramos el número se carga el valor hasta que encontramos un carácter que no es parte del número, por ejemplo, otro separador.

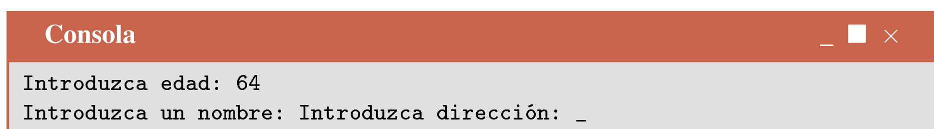
Supongamos la siguiente solución para el problema propuesto:

```
#include <iostream> // cin, cout, endl...
#include <string>   // string, getline
using namespace std;

int main()
{
   int edad;
   string nombre, direc;

   cout << Introduzca edad: ";
   cin >> edad;

   cout << "Introduzca un nombre: ";
   getline (cin, nombre);

   cout << "Introduzca dirección: ";
   getline (cin, direc);
   // ...
}
```

Confiando que una ejecución típica se comportara como sigue:



Sin embargo, el comportamiento no es ése, sino que nos encontraríamos que después de dar a la tecla *"Intro"* después del número el programa muestra:



es decir, se ha "saltado" la petición del nombre.

Realmente no ha sido así, la función `getline` se ha ejecutado y el nombre se ha leído. Para entender este problema debemos recordar cómo se realiza la lectura:

[2] Estamos presentando el uso más común de `getline`, que considera las líneas separadas por un salto de línea. También es posible leer cadenas de caracteres que incluyan saltos de línea y usen otro carácter como separador, aunque no es relevante estudiarlo en este momento.

1. El programa pide la edad. El sistema espera a que el usuario introduzca datos desde el teclado.
2. El usuario introduce los datos que desea hasta validarlos con la tecla *“Intro”*.
3. El programa recibe los datos introducidos, que en nuestro ejemplo se componen de los caracteres ’6’, ’4’ y ’\n’.
4. La lectura desde `cin` carga el valor 64 en la variable entera, y detiene la lectura en el salto de línea, ya que no es parte del dato entero.
5. El programa pide el nombre y ejecuta `getline`, para lo cual detecta que la entrada contiene un salto de línea y por tanto carga la variable `nombre` con la cadena vacía.
6. Dado que la lectura del nombre ha terminado, el programa solicita la dirección y ejecuta `getline`. En este caso no hay datos introducidos, y el programa espera a que el usuario introduzca una nueva línea.

Como puede ver, el problema está en que ahora el salto de línea es un carácter especialmente importante, ya que es el que determina el final de una línea. La función `getline` leerá caracteres hasta este carácter, incluyendo el caso de cero caracteres (línea vacía).

Para evitar este problema podríamos proponer la siguiente solución:

```
// ...
int main()
{
   int edad;
   string nombre, direc;
   cout << "Introduzca edad: ";
   cin >> edad;

   cout << "Introduzca un nombre: ";
   do {
      getline (cin, nombre);
   } while (nombre.size()==0);

   cout << "Introduzca dirección: ";
   do {
      getline (cin, direc);
   } while (direc.size()==0);

   cout << "Nombre: " << nombre << endl;
   cout << "Edad: "<< edad << endl;
   cout << "Dirección: " << direc << endl;
}
```

donde repetimos la entrada en caso de que la línea leída esté vacía.

A pesar de ello, observe que los bucles darían por válida una cadena con algún espacio blanco. Por ejemplo, introduzca un espacio después del número 64 y antes de pulsar *“Intro”*.

Podríamos haber indicado un tamaño mínimo de 3 caracteres en lugar de 1 para evitar estas entradas. Sin duda, la mejor solución podría ser crear un subalgoritmo que valide una cadena como nombre o dirección. En caso de que no sea válida, podría generar un mensaje de error y volver a pedir que se introduzca el dato correspondiente.

Fin ejemplo 7.2.1 ■

Ejercicio 7.3 Escriba un programa que lea una cadena y escriba en la salida estándar si es un palíndromo. Para ello, tendrá en cuenta que todas las letras de entrada están en minúscula, sin tildes y sin espacios.

7.2.5 Asignación de cadenas

La operación de asignación funciona de la misma forma que lo hace con los tipos básicos. Es decir, podemos usar el operador = con una variable de tipo `string` a su izquierda con el fin de asignarle un nuevo valor. En este sentido, esta sección parece innecesaria, ya que no es un caso especial, sino que se comporta como el resto de tipos. Sin embargo, es importante enfatizar que

a una variable de tipo **string** se puede asignar tanto otro objeto del mismo tipo como un valor concreto escrito entre comillas dobles, una cadena-C. Ejemplos de asignación son:

```
//...
string s1, s2;
//...
s2= string ("Ejemplo"); // Asigna un "literal"
s1= s2;                 // Los dos objetos valen: Ejemplo
s2= ".";                // Asigna una cadena−C
cout << s1 << s2;       // Imprime: Ejemplo.
//...
```

El hecho de confirmar el correcto funcionamiento de la asignación de cadenas es especialmente importante si el lector tiene en cuenta cómo funcionan las *cadenas-C*, ya que en este lenguaje la asignación de cadenas no es una operación válida[3], mientras que en el caso de tipo **string** no implica ninguna dificultad.

7.2.6 Operaciones relacionales

Los operadores relacionales que hemos usado para los tipos básicos también son válidos entre objetos de tipo **string**. Para ello, el lenguaje usa el orden lexicográfico habitual para ordenar cadenas de caracteres. Este orden es similar al que se utiliza en un diccionario, es decir, para ordenar dos cadenas se comparan uno a uno los caracteres que la componen de forma que la primera diferencia determina el orden de las cadenas.

Es importante tener en cuenta que el orden de los caracteres —de tipo **char**— es el que determinará el orden de las cadenas. Así, seguramente la cadena *"Zapatilla"* sea menor que la cadena *"alpargata"*, simplemente porque el carácter 'Z' es menor que el carácter 'a'. Por ejemplo, podemos comprobar el orden con el siguiente código:

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

int main()
{
   string s1 ("Zapatilla"), s2("alpargata"); // Dos objetos inicializados

   if (s1<s2)
     cout << s1 << " es menor que " << s2 << endl;  // Si 'Z'<'a'
   else
     cout << s2 << " es menor que " << s1 << endl;
}
```

Es muy probable que en su sistema se imprima el primer mensaje, ya que si la codificación del tipo **char** está basada —o se deriva— del código *ASCII*, los caracteres en mayúscula son menores que los caracteres en minúscula.

Por otro lado, tenga en cuenta que la comparación de cadenas de caracteres es una operación bien definida entre objetos de tipo **string**. De nuevo, recordamos la distinción entre **string** y *cadena-C*. Una comparación entre dos *cadenas-C* como la siguiente no sería válida:

```
if ("Zapatilla" < "alpargata") // Resultado indeterminado
   //...
//...
```

Si quiere hacer esa comparación, use el tipo **string** para crear dos "literales" que sí van a permitir determinar el orden correcto:

```
if (string("Zapatilla")<string("alpargata"))
   //...
//...
```

[3]Como estudiará en su momento, se puede programar la asignación de *cadenas-C*, aunque su funcionamiento a bajo nivel hace más complicado y propenso a errores el algoritmo que nos permite hacer que una *cadena-C* tenga el valor de otra.

Es importante que observe que el orden se determina comparando carácter a carácter. Eso significa que se comparará el carácter de la posición cero de las dos cadenas, si son iguales, el de la posición uno, etc. Esta operación incluye también cualquier carácter que no sea una letra, ya sean dígitos, caracteres imprimibles o incluso los que no se pueden imprimir. Por ejemplo, en el siguiente código:

```
string s1 ("Juan Aranda");
string s2 ("Juan  Zapata");

if (s1 < s2)
   cout << s1 << " es menor" << endl;
else cout << s1 << " no es menor" << endl;
```

el resultado será que la primera cadena `"no es menor"` que la segunda. La clave está en que el orden lo determina el carácter de la posición `5`, que en la primera cadena es `'A'` y en la segunda es `' '` (el espacio, con código *ASCII* 32).

7.2.7 Otras operaciones frecuentes

Además de las operaciones de las secciones anteriores, C++ ofrece otras operaciones especialmente diseñadas pensando en el procesamiento habitual de cadenas de caracteres. En esta sección añadimos las más interesantes.

Concatenación

Una operación especialmente habitual es la concatenación al final de una cadena. Es una forma natural de crear una cadena más grande: añadiendo nuevos contenidos al final.

Para facilitar esta operación el lenguaje C++ sobrecarga el operador de suma (+) para indicar concatenación de cadenas. Por ejemplo:

```
//...
string s1, s2, s;
s1= "Un ";
s2= "ejemplo.";
s= s1 + s2; // s contiene "Un ejemplo."
//...
```

Puede resultar más práctico modificar una cadena concatenando una cadena al final. Para esto, podemos concatenar con el operador +=:

```
//...
string s;  // Empieza siendo vacía
s+= "Un ";
s+= "ejemplo";
s+= '.';    // Note que añadimos un carácter, no una cadena
cout << s; // Escribe: Un ejemplo.
//...
```

donde podemos ver que también es posible concatenar caracteres en lugar de cadenas.

Es importante recordar, de nuevo, que esta operación corresponde al tipo **`string`**: es posible concatenar objetos de tipo **`string`**, no *cadenas-C*. Por ejemplo, el siguiente ejemplo no es válido:

```
string s;
s= "Esto " + "es " + "un " + "ejemplo.";
```

aunque sí sería válido escribir:

```
string s;
s= string("Esto ") + "es " + "un " + "ejemplo.";
```

> **Ejercicio 7.4** Razone brevemente por qué en el ejercicio anterior es posible concatenar múltiples *cadenas-C* cuando realmente no es válida la operación '+' entre ellas.

Operaciones con subcadenas

Las operaciones más habituales con cadenas operan con una subcadena dentro de una cadena. Hablamos de subcadena para indicar que es una cadena que ha formado parte o formará parte de otra cadena. Si entendemos la forma en que se especifica una cadena, es fácil intuir y usar las operaciones que ofrece la clase `string` para manejar cadenas.

En principio, es muy simple especificar una subcadena dentro de una cadena. Basta con indicar la posición donde comienza y donde termina, o equivalentemente, dónde comienza y su longitud. En la práctica, si tenemos una cadena de tamaño n, podemos indicar una subcadena —incluida en la anterior— de distintas formas:

1. Especificando una posición. En este caso, se entiende que la subcadena está compuesta por todos los caracteres desde esa posición hasta el final. Por ejemplo, especificando la subcadena en posición cero indicamos que se refiere a toda la cadena, es decir, la longitud de la subcadena es también n.
2. Especificando una posición y una longitud. Por ejemplo, podemos indicar la subcadena de posición 0 y longitud $n/2$ para indicar la primera mitad de la cadena original.
3. Especificando una posición y una longitud por exceso. En el caso en que la longitud sobrepase el final de la cadena, se entenderá que la subcadena llega sólo hasta el final. Por ejemplo, la subcadena de posición $n/2$ y longitud n corresponde a la segunda mitad, es decir, a la que comenzando en $n/2$ tiene una longitud de $n-n/2$.

Teniendo en cuenta esta forma de especificar una subcadena, algunas de las operaciones que podemos usar para trabajar con subcadenas son las siguientes:

- **Extraer subcadena**. La función `substr` permite extraer una subcadena. Un ejemplo de su uso es:

```
//...
string cad("Esto es una cadena de prueba");
string c1 = cad.substr(5,13); // Indicamos posición y longitud.
cout << "1: \"" << c1 << '\"' << endl;
string c2 = cad.substr(12);    // Indicamos sólo posición.
cout << "2: \"" << c2 << '\"' << endl;
```

  que obtiene como resultado, en la salida estándar, lo siguiente:

Consola

```
1: "es una cadena"
2: "cadena de prueba"
```

  Observe que hemos especificado el carácter comillas dobles con una barra invertida para que se escriba en la salida estándar. Estas comillas en el resultado no son parte de la cadena.

- **Borrar subcadena**. La función `erase` permite borrar una subcadena. Un ejemplo de su uso es:

```
//...
string cad("Esto es una cadena de prueba");
cad.erase(12,10);                  // Indicamos posición y longitud.
cout << "1: \"" << cad << '\"'  << endl;
cad.erase(7);                      // Indicamos sólo posición.
cout << "2: \"" << cad << '\"'  << endl;
cad.clear();
cout << "3: \"" << cad << '\"'  << endl;
```

  que obtiene el siguiente resultado en la salida estándar:

Consola

```
1: "Esto es una prueba"
2: "Esto es"
3: ""
```

Observe que como última operación hemos usado `clear`, que elimina todos los caracteres de la cadena, haciéndola vacía.

- **Insertar subcadena**. La función `insert` nos permite insertar una cadena en otra. En este caso tendremos que especificar la posición y la cadena a insertar. Un ejemplo de su uso es:

```
string cad("Esto es una prueba");
cad.insert(12, "cadena de ");
cout << "1: \"" << cad << '\"'  << endl;
```

que obtiene como resultado, en la salida estándar, lo siguiente:

Consola

```
1: "Esto es una cadena de prueba"
```

- **Reemplazar una subcadena**. La función `replace` nos permite reemplazar una subcadena por otra. En este caso tenemos que especificar una subcadena a eliminar y una subcadena a insertar. Un ejemplo de su uso es:

```
//...
string cad("Esto es un mal ejemplo");
cad.replace(11,3,"buen");
cout << "1: \"" << cad << '\"'  << endl;
//...
```

que obtiene como resultado, en la salida estándar, lo siguiente:

Consola

```
1: "Esto es un buen ejemplo"
```

Ejercicio 7.5 Escriba un programa que lea una línea y vuelva a escribirla sin espacios. Para ello, calcule una nueva cadena modificando la original mediante la eliminación de todos los caracteres que sean espacio.

Búsqueda

La operación de búsqueda localiza una subcadena dentro de otra cadena. Le dedicamos esta sección específica porque es importante especificar algunos detalles adicionales para garantizar que funciona correctamente.

La idea es muy sencilla: especifico la cadena a buscar y la función me devuelve la posición en la que se encuentra. El problema es que para diseñar esta función se tienen que decidir dos cosas:

1. Qué tipo de dato devuelve. Aunque es posible que el lector esté tentado a decir "tipo **`int`**" —el que hemos usado hasta ahora para indicar posiciones— lo cierto es que no es fácil decidirse. En general podría ser suficiente, pero en algunos sistemas podría ser demasiado "pequeño" (por ejemplo, sólo llegar a 32767). Además, ¿para qué usar un tipo con *signo*?, sería más lógico usar un entero positivo.

2. Qué devuelve en caso de que no se encuentre. Para responder a esto, primero hay que tener resuelta la pregunta anterior, es decir, dependiendo del tipo de dato podríamos devolver un número especial para indicar que no se encontró. Por ejemplo, un número “imposible”, como el -1 o un número muy alto.

Dado que no es una buena decisión de diseño fijar el tipo de dato para el lenguaje, sino que es mejor dejar que cada sistema implemente estas decisiones a medida, el tipo `string` se diseña de forma genérica:

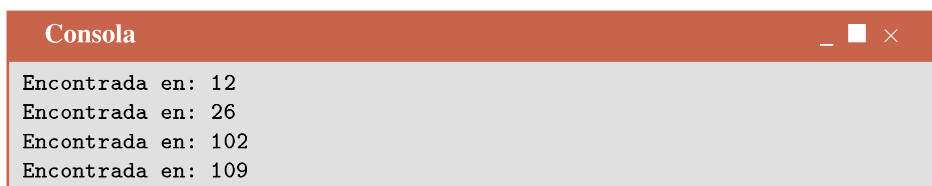
1. El tipo devuelto para la búsqueda es: `string::size_type`.
2. El valor devuelto cuando no se encuentra es: `string::npos`.

No piense que el tipo será probablemente el tipo `int` (de hecho, probablemente será algún tipo entero, pero sin signo). Si va a almacenar en una variable la posición donde se localiza una subcadena, declare la variable con el tipo especial que hemos indicado.

La función `find` de búsqueda de una subcadena en otra tendrá como parámetro una subcadena a buscar y, opcionalmente, una posición a partir de la que buscar. Un ejemplo de su uso es:

```
string cad("En un lugar de la Mancha, "
           "de cuyo nombre no quiero acordarme, "
           "no ha mucho tiempo que vivía un hidalgo "
           "de los de lanza en astillero adarga antigua, "
           "rocín flaco y galgo corredor");

string::size_type pos= cad.find("de");  // Buscamos primera ocurrencia
if (pos != string::npos) {
   do {
      cout << "Encontrada en: " << pos << endl;
      pos= cad.find("de",pos+1); // Buscamos siguiente ocurrencia
   } while (pos != string::npos);
}
else cout << "No he encontrado la palabra" << endl;
```

que obtiene como resultado, en la salida estándar, lo siguiente:



De forma similar se usa la función `rfind` que busca una subcadena desde el final. En el ejemplo anterior podemos buscar todas la ocurrencias de la cadena `"de"` desde el final. Para ello, cambiamos el código de búsqueda como sigue:

```
//...
string::size_type pos = cad.rfind("de"); // Buscamos última ocurrencia
if (pos != string::npos) {
   do {
      cout << "Encontrada en: " << pos << endl;
      pos= cad.rfind("de",pos−1); // Seguimos desde la posición anterior
   } while (pos != string::npos);
}
else cout << "No he encontrado la palabra" << endl;
```

que obtiene las mismas posiciones que en el código anterior, aunque en orden inverso.

Como puede ver, en los ejemplos anteriores hemos usado un nuevo tipo de dato entero: `string::size_type`. Es cierto que no sabemos exactamente qué números puede almacenar, pero sabemos que se comporta como un entero, así que podemos realizar operaciones como sumar o restar uno, obteniendo un nuevo entero.

Casi parece que podemos prescindir de ese tipo y declarar `pos` de tipo `int`. De hecho, si lo prueba en su sistema, tal vez funcione. Sin embargo, si queremos garantizar que es correcto en

todos los casos, tendremos que manejar las posiciones con ese tipo. De esa forma, la comparación con `string::npos` funcionará sin problemas.

7.2.8 Funciones y tipo string

La forma en que funciona el tipo `string` cuando se pasa como parámetro o se devuelve desde una función no es distinta al tipo `vector`. Recuerde la similitud que tienen estos dos tipos, ya que podemos ver al tipo `string` como un caso especializado de vector de caracteres. Por tanto, debemos tener en cuenta que:

- Podemos pasar un `string` a una función por valor o por referencia como cualquier tipo básico del lenguaje. Es decir, el paso por valor implica una copia del objeto original y el paso por referencia implica el uso del objeto original.
- Una función puede devolver un `string` mediante `return`. En principio, esta devolución se puede considerar una copia, ya que el objeto devuelto "se copia" como resultado al punto de llamada.

Teniendo en cuenta que un objeto de tipo `string` puede ser muy grande, es importante tener en cuenta que si realizamos copias nuestro código puede resultar demasiado ineficiente. Recordemos que:

- Podemos pasar un objeto de entrada por referencia añadiendo `const` para indicar que no se va a modificar.
- Podemos usar un parámetro por referencia para evitar la copia en la devolución con `return`. En el punto de llamada se incluye el nombre de un objeto que va a modificarse con el resultado de la función.

Como puede observar, el tipo `string` tiene un comportamiento idéntico al tipo `vector`, por lo que aplicamos criterios similares para el diseño de funciones.

Ejercicio 7.6 Escriba un programa que lea una línea e indique si es un palíndromo. Para ello, será necesario crear varias funciones. En concreto puede considerar las siguientes:

1. Una función que elimine los espacios y tabuladores de una cadena.
2. Una función que pase las mayúsculas a minúsculas.
3. Una función que devuelva la cadena inversa.

El resultado será, por tanto, un programa que procesa una cadena con las anteriores funciones y comprueba si la cadena es igual a su inversa.

7.3 El tipo char: codificaciones

En los temas anteriores hemos presentado el tipo `char` de una forma bastante simple y fácil de manejar, considerando que la codificación que usamos corresponde al código *ASCII* (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). De esta forma, un objeto de tipo `char` no es más que un *byte* que codifica un carácter *ASCII* (letra, dígito, carácter de control, etc.). Si revisa la sección 1.2.3 (página 9) puede recordar la introducción a este código.

La mayoría de los cursos de introducción a la programación no consideran más posibilidades, ya que los fundamentos para aprender a programar pueden adquirirse diseñando e implementando algoritmos que no requieren de más complicación. Sin embargo, si consideramos las herramientas que ahora mismo usamos en nuestros ordenadores, adaptadas a un contexto global en el que se desarrollan la mayoría de las soluciones, descubrimos que cada vez se hace más importante conocer cómo se codifican los caracteres, incluso en cursos introductorios.

En esta sección vamos a introducir algunas de las codificaciones más interesantes, aunque en general desarrollaremos el curso de programación evitando entrar en detalles técnicos sobre el

problema de la codificación. Los problemas de codificación son muy amplios, planteando soluciones que se derivan del lenguaje, de las bibliotecas usadas, o incluso del sistema operativo sobre el que se trabaja. Por tanto, presentaremos los conceptos fundamentales para entender en qué consiste la codificación, pero seguiremos trabajando con el código *ASCII* que hemos supuesto en los temas anteriores.

7.3.1 Código ASCII: Extensiones

El código *ASCII* propiamente dicho corresponde a una tabla de codificación de 128 posiciones, ya que se usan sólo 7 bits. Se empieza a diseñar hace más de medio siglo, y se enfoca a la codificación de textos que usan el alfabeto inglés. La tabla corresponde a la mitad superior de la figura 7.1. Observe que se incluyen tanto caracteres imprimibles como no imprimibles. Cualquier texto escrito en inglés se puede representar como una secuencia de caracteres de esta tabla, es decir, se puede guardar como una secuencia de tuplas de 7 bits.

	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	SGCI	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	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 7.1
Tabla de codificación ISO-8859-1.

Lógicamente, para otros lenguajes no es suficiente. Por ejemplo, en español necesitamos las tildes, la diéresis y la letra *ñ* para poder representar cualquier texto. En otros, además, podrían aparecer otros caracteres que necesiten de otros signos diacríticos. Para ampliar las posibilidades de representación se crea lo que denominamos un código *ASCII extendido* añadiendo un bit, es decir, un código de 8 bits (por tanto, una tabla de 256 posibilidades) en el que la primera mitad corresponde al código *ASCII* y la segunda a la extensión. En la figura 7.1 se presenta el estándar ISO del alfabeto latino, con el que es posible escribir cualquier texto de un conjunto de lenguas de Europa occidental.

La codificación ISO-8859-1 nos permite escribir cualquier texto almacenando cada carácter en un único objeto `char`. En este caso, el tipo `char` puede guardar únicamente un *byte* que corresponde al código *ASCII* correspondiente o al código extendido si el carácter está en la segunda mitad de la tabla. Por simplicidad, usaremos el término *tabla ASCII* para referirse a una codificación extendida, aunque formalmente corresponda sólo a los primeros 128 caracteres.

Realmente no es la única codificación posible y de hecho se han creado otras tablas que se han usado ampliamente, aunque en muchas de las codificaciones se ha respetado que las primeras 128

entradas corresponden a la tabla *ASCII* de forma que sea compatible con la codificación inglesa original. Por ejemplo, otro estándar ISO es el *8859-15* (véase la tabla 7.1 en la página 157) que corresponde a una pequeña modificación del 8859-1 con la intención de cambiar algunos caracteres, entre ellos, el carácter del euro.

Otra codificación de 8 bits ampliamente usada es la conocida como *windows-1252* (o *CP1252*). Diseñada por *Microsoft* para su sistema operativo y que también se basa en una ampliación del código *ASCII* para el alfabeto latino. La mayor parte de los caracteres del *ISO-8859-1* están situados en la misma posición de *windows-1252*. Incluye el carácter del euro, así como nuevos caracteres imprimibles.

En este curso de introducción a la programación podemos asumir que trabajamos con alguna codificación *ASCII extendida*, es decir, que cada carácter corresponde a un *byte* de una tabla de 256 posiciones. Por tanto, un `string` es una secuencia de caracteres, y por consiguiente, una secuencia de bytes que codifican un valor de esta tabla.

Ejemplo 7.3.1 Escriba un programa que lee todos los caracteres desde la entrada estándar y escribe en la salida estándar el carácter y su código *ASCII*. El programa termina cuando se introduce el carácter '#'.

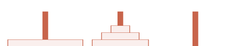
Para resolver el problema tenemos que tener en cuenta que cuando usamos un objeto de tipo `char` en una expresión de enteros, o asignamos un objeto de este tipo a un entero, realmente trabajamos con el valor *ASCII* que almacena. Por tanto, si escribimos un carácter en `cout` se imprime el carácter correspondiente, pero si lo asignamos a un entero e imprimimos el entero, se imprime su valor *ASCII*. Por tanto, una posible solución es la siguiente:

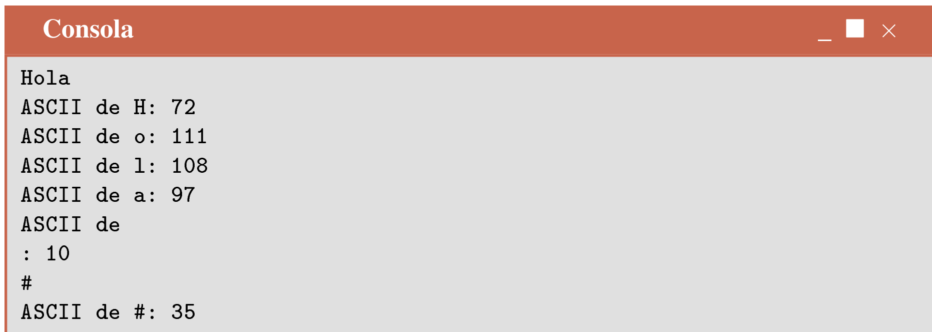
```
#include <iostream>
using namespace std;

int main()
{
   char c;      // donde leemos cada carácter
   int ascii;   // donde asignamos el carácter para escribirlo

   do {
      c= cin.get();
      ascii= c;
      cout << "ASCII de " << c << ": " << ascii << endl;
   } while (c!='#');
}
```

En este ejemplo es interesante que observe que:

- Hemos usado la función `get` para extraer desde `cin` un carácter. Esta función extrae el siguiente carácter desde la entrada estándar sin saltarse ningún carácter. Si usáramos la lectura con $>>$, los caracteres espacios se considerarían separadores y se ignorarían. Al usar `get`, si el siguiente carácter es un espacio, se leerá y devolverá dicho espacio (como carácter *ASCII* 32).
- En la línea de salida se escriben las dos variables. Realmente las dos contienen el mismo número. Por ejemplo, si se lee un espacio, ambas tendrán el número 32. Lógicamente, `cout` considera que imprimir `c` consiste en imprimir el carácter correspondiente mientras que imprimir `ascii` consiste en imprimir los dos caracteres '3' y '2' para formar el número 32 en la salida.

En realidad, este programa se podría escribir de una forma más simplificada, aunque el listado que presentamos deja más claro el sentido de las entradas y salidas. Una posible ejecución de este programa sería la siguiente:



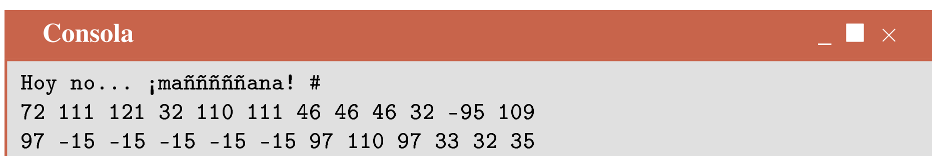


Donde podemos ver que al llegar al bucle se detiene para que introduzcamos un carácter. Lógicamente, el programa no sigue hasta que pulsamos la tecla *"Intro"*. Al haber escrito 4 caracteres y la tecla *"Intro"*, el bucle itera 5 veces escribiendo tanto el carácter como el código *ASCII* de cada uno de ellos. Observe que en el último carácter —que corresponde al salto de línea— la escritura del carácter implica que salta a la siguiente línea, donde escribe el código 10 correspondiente. Finalmente, introducimos el carácter '#' para indicar final de la entrada, haciendo que el bucle itere por última vez indicando que su código es el 35.

En esta ejecución hemos usado sólo caracteres de la primera mitad de la tabla. Vamos a probar un ejemplo con más caracteres para incluir algunos de la segunda mitad. Además, para tener una entrada más larga cambiamos el código del bucle a lo siguiente:

```
//...
   do {
      ascii= cin.get();
      cout << ascii << ' ';
   } while (c!='#');
//...
```

de forma que se escriba una secuencia de códigos *ASCII* separados por espacios. Si ejecuto el programa en mi sistema, usando la codificación *ISO-8859-1*, el resultado podría ser:



donde podemos observar que:

- Aparecen números negativos. Esto no es ningún error. El problema es que el tipo `char` ocupa un byte, pero no se sabe si es con signo o no. Así, si el sistema implementa este tipo con signo el rango de posibilidades no es $[0,255]$, sino $[-128,127]$. Es como si la segunda mitad de la tabla la hubiéramos colocado a la izquierda del cero. El estándar de C++ no especifica si tiene que tener signo, por tanto, en su sistema puede que los números sean positivos.
- Los caracteres de la segunda mitad son la apertura de exclamación y las letras 'ñ'. Estos caracteres no se usan en el alfabeto inglés, por lo que se han añadido en la parte extendida.

Ejecutemos el programa obteniendo el código *ASCII* en el rango $[0,255]$ (sería como sumar 256 a los número negativos para ponerlos a la derecha de la primera mitad). Además, para presentar el resultado, lo formateamos para separar las líneas con caracteres de la parte extendida. El resultado se podría escribir así:

```
Consola
Hoy no... ¡maññññññana! #
72 111 121 32 110 111 46 46 46 32
161
109 97
241 241 241 241 241
97 110 97 33 32 35
```

Observe que los códigos 161 (segunda línea de números) y 241 (cuarta línea) son los que corresponden a los caracteres de la segunda mitad de la tabla. Como puede comprobar, corresponden a la "exclamación española" —signo de apertura de exclamación— y a la letra *'ñ'*.

Fin ejemplo 7.3.1 ■

7.3.2 Unicode

El problema de la adaptación del software a distintos idiomas, la transmisión y visualización de textos de otros idiomas, o incluso la posibilidad de usar varios idiomas en un mismo sistema hace que la solución que hemos indicado en las secciones anteriores no se pueda usar. Para resolver este problema se crea el estándar *Unicode*, donde se incluyen todos los caracteres asociando a cada uno de ellos un código único.

El estándar ha ido creciendo, añadiendo nuevos caracteres hasta alcanzar decenas de miles. Aunque inicialmente un valor de 16 bits parecía más que suficiente para codificar todos los caracteres actuales, pronto se amplió el estándar para poder usar 32 bits, pudiendo alcanzar millones de entradas. Ahora bien, una vez determinados los caracteres y asignados los códigos correspondientes, será necesario seleccionar una forma de tratarlos en el ordenador.

Codificación de longitud fija y variable

La solución más simple para realizar una codificación es mediante longitud fija. La idea es que determinamos cuántos bits queremos usar para cada carácter y creamos una tabla para realizar cada una de las asignaciones. Por ejemplo, si decidimos usar 32 bits, cada carácter tendrá asignado una secuencia única de 32 bits. Se denomina de longitud fija porque todos y cada uno de los caracteres se codifican con un número fijo de bits.

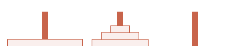
Sin embargo, no siempre es la mejor solución. Por ejemplo, si usamos una codificación de 32 bits, cada carácter que queramos almacenar necesitará 4 bytes. Eso ocurre tanto para una letra común —como alguna del alfabeto inglés— como para un carácter de una determinada lengua muy local que incluso puede que no se esté usando en nuestro sistema.

Podemos mejorar la eficacia de nuestra codificación si aplicamos una longitud variable. En este caso, los caracteres más habituales se codifican con menos bits. Por ejemplo, las letras del alfabeto inglés se pueden codificar con un único byte, mientras que un carácter de una lengua muerta se codifica con 4 bytes. Lo importante es que no se puedan confundir las codificaciones, es decir, que se pueda codificar y decodificar de forma única.

UTF

Dado que hay distintas formas de codificación, *Unicode* define distintas alternativas para que podamos seleccionar una de ellas como la codificación estándar de nuestro *software*, de forma que sea compatible con cualquier otro que la use. Usando el acrónimo *UTF* (**U***nicode* **T***ransformation* **F***ormat*) se establecen distintas formas de codificación. Podemos destacar las siguientes:

- *UTF32*. Una codificación fija de 32 bits. Esta resulta simple e intuitiva, ya que si el estándar Unicode usa 32 bits para clasificar todos los caracteres, lo más sencillo es crear una codificación de la misma longitud para que un ordenador los procese.



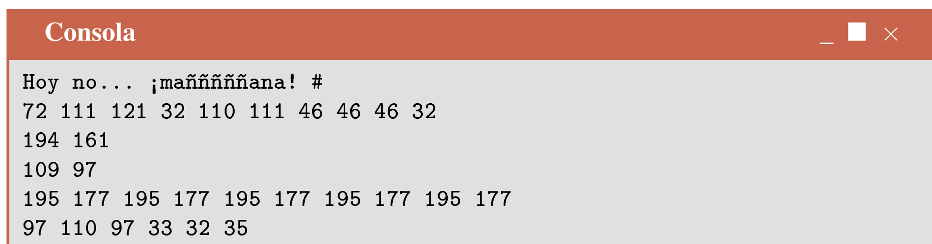
- *UTF16*. Una codificación de longitud variable de 16 bits mejora la eficacia de la codificación. Tenga en cuenta que el número de caracteres que resuelven la mayoría de los caracteres habitualmente usados en el mundo se pueden codificar con menos de 16 bits. Si queremos evitar un gasto excesivo podemos codificar cada uno de ellos con un grupo único de 16 bits, y dejar los caracteres "menos habituales" codificados con dos grupos de 16 bits.
- *UTF8*. Una codificación de longitud variable de 8 bits permite tener caracteres codificados con un único byte. Lógicamente, un carácter cualquiera puede codificarse con 1, 2, 3, o 4 bytes.

La codificación *UTF16* es un buen compromiso entre tener un tamaño fijo en la codificación y no desperdiciar mucho espacio para almacenar un carácter. Podríamos incluso crear sistemas que trabajaran sólo con el subconjunto de caracteres codificados con un único grupo de 16 bits[4], que serían válidos prácticamente en cualquier lugar del mundo. Este sistema es el usado por *Microsoft Windows*.

La codificación *UTF8* está especialmente diseñada para ser compatible con la tabla *ASCII*. Dado que es posible asignar códigos de 1 byte, podemos asignar los mismos códigos para la primera mitad de la tabla *ASCII*. Por ejemplo, cualquier texto inglés codificado en *ASCII* es idéntico en *UTF8*. Para otros caracteres, como por ejemplo los que hemos añadido en la parte extendida para Europa occidental, se opta por utilizar más de un byte.

Para que sea más eficaz, los más habituales requieren solamente dos bytes y los menos probables requieren hasta 4 bytes. La ventaja de esta codificación es que gran parte de los documentos y software que se han creado para trabajar con el código *ASCII* siguen siendo válidos prácticamente sin modificación. Este sistema es ampliamente usado, por ejemplo, en los sistemas *GNU/Linux*.

Para mostrar cómo se almacenan estos caracteres, hemos usado el programa anterior que imprime los códigos *ASCII* para mostrar todos los bytes usados en un mensaje. Usamos el mismo mensaje del ejemplo anterior, pero suponiendo que se ha codificado en UTF8, por lo que el resultado ha sido el siguiente:



Cada vez que hemos leído un **`char`**, hemos extraído un byte del mensaje. Tenga en cuenta las consecuencias de esta lectura. Si un carácter ocupa dos bytes, al leer un **`char`** realmente hemos leído la mitad del carácter. Aquí, es más fácil pensar en un **`char`** como un *byte* en lugar de como un carácter individual.

Observe las diferencias con respecto a la última salida:

- Tenemos el mismo mensaje, pero ahora tiene más bytes. Los caracteres de la parte extendida *ASCII* del mensaje se han codificado con dos bytes y, por tanto, tenemos 6 bytes más.
- Los caracteres de la primera mitad de la tabla son idénticos.
- El carácter apertura de admiración español se ha codificado con el par $194, 161$.
- La línea que presenta el grupo de 5 caracteres `'ñ'` repite 5 veces el par $195, 177$.

[4]Lo que se denomina el plano básico multilingüe.

7.3.3 Editor de texto y codificación usada

Si analiza el editor de texto que habitualmente está usando para escribir los programas, es probable que descubra opciones que le permiten cambiar la codificación usada. Es decir, puede escribir un texto y establecer que el editor lo almacene usando una determinada codificación, aunque no sea la habitual de su sistema.

No sólo los editores. El terminal donde ejecuta los programas que compilamos también tiene una codificación establecida, por lo que si escribimos una secuencia de **`char`**, la va a interpretar según dicha codificación. Por ejemplo, si su terminal funciona con *UTF8*, al escribir una secuencia de bytes codificados por ejemplo con *ISO-8859-1*, dará lugar a caracteres extraños debido a los errores de decodificación.

Como puede ver, el problema de la codificación a usar no es trivial, ya que tiene múltiples posibilidades y no todas resueltas con el tipo de dato **`char`**. El tipo **`string`** es una secuencia de objetos **`char`** y lógicamente también deberá tener en cuenta esta característica.

Para un curso de introducción a la programación, en el que no deseamos crear aplicaciones multilingües, no es necesario incluir código para tratar este problema. En el resto de este libro seguiremos suponiendo que tenemos una codificación del tipo *ASCII extendido*, por tanto, cada carácter se almacena como un único objeto de tipo **`char`**. A pesar de ello, también puede usar la codificación *UTF8* ya que al ser en gran medida compatible con la anterior el código que desarrolle puede seguir siendo válido.

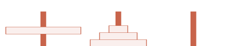
Si lo desea, pruebe sus programas con ejemplos donde use sólo la primera mitad de la tabla *ASCII*. De esta forma podrá avanzar en el curso obviando este problema y dejando la dificultad añadida para cuando tenga más experiencia de programación.

Ejercicio 7.7 Reconsidere el programa del palíndromo resuelto en el ejercicio 7.6 (página 156). Suponga que hemos resuelto el programa suponiendo una codificación *ISO-8859-1*. ¿Qué puede ocurrir si ejecutamos el programa en un sistema con codificación *UTF8*?

7.4 Problemas

Problema 7.1 Implemente una función que reciba una cadena que contiene el nombre y apellidos de una persona y que devuelva otra con las iniciales. Por ejemplo, ante la entrada “Juan Carlos Martínez Pérez” devolverá “J. C. M. P.”.

Problema 7.2 Implemente una función que reciba una cadena y que devuelva el número de palabras que contiene. Las palabras pueden estar separadas entre sí por espacios en blanco (uno o más) o signos de puntuación.



8 Estructuras y pares

Introducción 163
Estructuras 164
Pares con la STL 175
Problemas 183

8.1 Introducción

En los temas anteriores hemos ampliado las posibilidades para definir las estructuras de datos de nuestros programas incorporando un *tipo de dato compuesto* —el tipo `vector`[1]— donde almacenamos varios objetos de un mismo tipo base. En este tema proponemos la creación de nuevos tipos de datos compuestos, aunque en este caso no nos limitaremos a definirlo con un único tipo *base*.

Las estructuras nos permiten definir nuevos tipos de datos *compuestos* que denominamos *heterogéneos*, es decir, son tipos de datos compuestos de varios objetos que pueden ser de distintos tipos. Por ejemplo:

- Podemos unir un número entero y una letra para representar una posición del tablero de ajedrez.
- Podemos crear un objeto que tiene dos números reales para representar un punto del plano.
- Podemos unir tres enteros para crear un único objeto que representa una fecha.
- Podemos unir un título, nombre, editorial y año para representar un libro.

Como puede ver, tenemos múltiples posibilidades: unir varios objetos del mismo tipo, de distinto tipo, simples, compuestos, etc.

En el caso de un vector indicamos el número de componentes y accedemos a ellos con un índice. Para las estructuras, al ser de distintos tipos, tenemos que listar —uno a uno— los componentes con sus tipos. A estos componentes los denominamos *miembros* de la estructura. Cada miembro tiene un nombre asociado que nos permitirá poder referenciarlo cuando operemos con la estructura.

Es posible que en otras referencias encuentre este tipo de datos con el nombre de *registro* en lugar de *estructura* y el nombre *campo* para referirse a un *miembro*. Esta nomenclatura se utiliza intensivamente en el contexto de las bases de datos.

En este capítulo presentamos las estructuras como una forma de agregar distintos tipos sin incluir nuevas características que incluye C++. Se podría decir que se presentan las estructuras tipo

[1] Realmente hemos hablado del tipo `vector` y `string`, aunque este segundo se podría considerar una especialización del primero.

C. Cuando avance en C++ comprobará cómo una estructura puede ser mucho más complicada, tanto como una clase.

8.1.1 Tipos de datos definidos por el usuario

Con la presentación de las estructuras aparece un nuevo concepto: *tipo definido por el usuario*. Hasta ahora, todos los tipos estaban definidos en C++. Por un lado, los tipos simples estaban definidos en el lenguaje y, por otro, los tipos **vector** y **string** estaban definidos en la STL[2]. Con las estructuras podemos crear tipos nuevos. Cada vez que definimos una estructura, estamos definiendo un nuevo tipo.

Básicamente la idea es agrupar un conjunto de tipos —los miembros o campos de la estructura— y darles un nuevo nombre. Este nombre es un identificador que corresponde a un nuevo tipo. Declarar un objeto de dicho tipo nos permitirá disponer de un objeto que incluye en su interior un objeto de todos y cada uno de los tipos que hemos agrupado.

8.2 Estructuras

El formato de la definición de una estructura es:

```
struct <nombre de la estructura> {
   <declaración miembro 1>;
   <declaración miembro 2>;
   ...
   <declaración miembro n>;
};
```

donde cada declaración de un miembro tiene la sintaxis que ya conocemos para la declaración de una variable. En lugar de declarar una variable, se declara el nombre asociado al componente correspondiente, es decir, el tipo del miembro y el nombre con el que podremos referirnos a él. Por ejemplo:

```
struct Punto {
   double x;
   double y;
};
```

define una nueva estructura de nombre `Punto` con dos campos, `x` e `y` que son del mismo tipo: **double**. Además, es posible declarar varios miembros en una misma línea, de forma similar a cuando declaramos varias variables de un mismo tipo. Por ejemplo, podríamos haber definido el tipo `Punto` como:

```
struct Punto {
   double x, y;
};
```

que obtiene una declaración más simplificada pero equivalente.

Otro ejemplo que agrupa distintos tipos es el siguiente:

```
struct Estudiante {
   string nombre;
   int curso;
   char grupo;
};
```

donde se define una nueva estructura `Estudiante` —es un nuevo tipo— que contiene tres miembros de distintos tipos, con nombres `nombre`, `curso` y `grupo`.

[2]Note que el tipo **vector** y **string** son realmente una familia de tipos de datos ya que se pueden definir múltiples tipos si cambiamos el tipo *base* sobre el que se define el contenedor.

8.2.1 Declaración de variables

Una vez definida la estructura el programador dispone de un nuevo tipo. El nombre de este tipo es el que hemos dado a la estructura, es decir, el identificador después de la palabra `struct`. Para usar una estructura, usaremos su nombre para declarar una variable que tenga ese tipo. Por ejemplo:

```
Punto p,q;
Estudiante alumno;
```

que define tres nuevos objetos: dos de tipo `Punto` y un tercero de tipo `Estudiante`.

Una diferencia importante con C es que en este lenguaje es necesaria la palabra clave `struct` para identificar la estructura. Así, si queremos declarar las variables anteriores, en C es necesario hacer:

```
struct Punto p,q;
struct Estudiante alumno;
```

mientras que en C++ anteponer `struct` es opcional. En la práctica, lo habitual es prescindir de la palabra `struct`. Si encuentra código que la incluya, probablemente se deba a que fue escrito pensando en la compilación en C.

Inicialización

Cuando se define un nuevo objeto, se puede inicializar una estructura especificando los valores de cada uno de sus campos. Para ello, escribimos entre llaves cada uno de esos valores separados por comas. Por ejemplo:

```
Punto origen= { 0.0, 0.0 };
```

define un nuevo objeto de tipo `Punto` donde hemos especificado los valores de inicialización. El primer valor se asignará a `x` y el segundo a `y`, siguiendo el orden especificado en la definición de la estructura.

Definición y declaración de variables

También es posible declarar variables junto con la definición de una estructura. El formato es:

```
struct <nombre de la estructura> {
   <declaración miembro 1>;
   <declaración miembro 2>;
   ...
   <declaración miembro n>;
} <variables>;
```

Por ejemplo, podemos definir `Fecha` y declarar dos variables `f1` y `f2` de ese tipo como sigue:

```
struct Fecha {
   int dia;
   int mes;
   int anio;
} f1, f2;
```

Además, se puede prescindir del nombre de la estructura en su definición (el nombre o la declaración de variables es opcional). Aunque esta definición de *estructuras anónimas* no la usaremos en nuestros códigos ya que siempre pondremos nombres a los nuevos tipos.

8.2.2 El operador punto

El operador fundamental para manejar estructuras es el operador punto. Con este operador podemos seleccionar uno de los campos de la estructura. La forma de uso consiste en escribir primero el nombre de un objeto y a continuación seleccionar el campo de la estructura incluyendo un punto seguiro por el nombre del campo. Es decir, el formato es:

```
<nombre>.<miembro>
```

donde *nombre* se refiere a un objeto de tipo estructura y *miembro* es el nombre de uno de sus campos. Por ejemplo, podemos usar el tipo `Punto` anterior de la siguiente forma:

```
Punto p= { 0.0, 0.0 };

cout << "Punto inicial: (" << p.x << ',' << p.y << ')' << endl;
p.x= 1.0;
p.y= p.y + 5;
cout << "Punto final: (" << p.x << ',' << p.y << ')' << endl;
```

que escribiría lo siguiente:

Una vez que hemos especificado el nombre del objeto y seleccionado uno de sus campos, podemos realizar cualquier operación válida para ese objeto miembro. Por ejemplo, podemos ver en el ejemplo anterior que `p.y` es el nombre de un objeto de tipo **`double`**. Por consiguiente, la operación de añadir cinco a su valor es correcta con el efecto que ya conocemos.

Observe que respecto a la dificultad de uso de uno de sus campos, sólo podría considerarse más complicado porque el nombre es *"más raro"*, pues contiene un punto. Sin embargo, una vez que se dé cuenta de que no es más que el nombre de un objeto de tipo **`double`**, podrá realizar las mismas operaciones de siempre sin encontrar ningún problema.

Esta capacidad para abstraer los detalles de una expresión y poder ver claramente a qué se *refiere* el resultado de una expresión es la clave para comprender fácilmente complejas líneas de código. Conforme avance en el lenguaje, irá descubriendo esta forma de entender el código en C++.

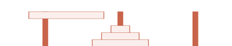
Para entender mejor el uso del operador punto podemos proponer un ejemplo un poco más complicado mediante la definición de estructuras que contienen estructuras. Por ejemplo, una vez que hemos definido la estructura `Punto` podemos definir:

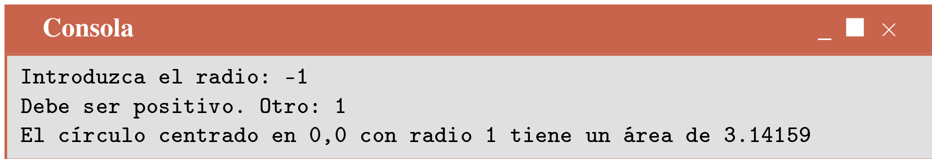
```
struct Circulo {
   Punto centro;
   double radio;
};
```

Con estas dos estructuras podemos usar el operador punto para acceder a cada uno de sus campos como sigue:

```
Circulo c;

cout << "Introduzca el radio: ";
cin >> c.radio;
while (c.radio<=0) {
   cout << "Debe ser positivo. Otro: ";
   cin >> c.radio;
}

c.centro.x= c.centro.y= 0.0;

cout << "El círculo centrado en " << c.centro.x << ',' << c.centro.y
     << " con radio " << c.radio
     << " tiene un área de " << 3.14159 * c.radio * c.radio << endl;
```

que podría dar lugar a una ejecución como la que sigue:





Observe que *c.centro* es el nombre de una estructura de tipo *Punto*. Por eso, cuando ponemos ese nombre podemos volver a usar el operador punto para acceder a su interior. De alguna forma, el programador ha escrito *c* y ha pensado en una estructura con sus dos componentes, a la que ha añadido *.centro* para situarse en una estructura de tipo *Punto*, enmarcada dentro de *c* y —obviando que tiene un nombre extraño especificado como una expresión— ha añadido *.radio* para seleccionar un objeto **double**, con el que se pueden hacer la variedad de operaciones que ya conocemos.

8.2.3 Copia y asignación de estructuras

Una estructura puede definir un objeto de un tamaño muy grande. Aunque se pueden definir estructuras muy simples, con pocos datos, de tamaño muy pequeño, en muchos casos resultan objetos de un tamaño considerable por lo que hacer una copia de uno de ellos puede resultar costoso.

En C++ es válida la asignación de una estructura a otra. El efecto es la asignación campo a campo de una estructura en otra. Por ejemplo, podemos escribir el siguiente código:

```
Circulo c1 = { {0.0,0.0}, 1.0 },
        c2 = { {1.0,1.0}, 2.0 };

c1= c2;  // Asigna todos los campos

cout << "Radio " << c1.radio
     << " centrado en "
     << '(' << c1.centro.x << ',' << c1.centro.y << ')' << endl;
```

cuya ejecución da como resultado el siguiente mensaje en la salida estándar:

Observe que en una única asignación se ha actualizado *c1* con todos los valores de *c2*. Se han asignado todos sus campos, incluido el objeto *centro* que también implica, a su vez, una asignación de estructuras. Por otro lado, es interesante que note cómo se han incluido llaves anidadas para inicializar los objetos de tipo *Circulo*: una llave para especificar el círculo y otra interna donde especificar el punto central.

Funciones y estructuras

El uso de estructuras para pasar o devolver datos a una función no implica ningún comportamiento especial. El paso de un objeto de tipo estructura —por valor o por referencia— así como su devolución mediante **return** tiene un comportamiento idéntico a los tipos simples.

Sin embargo, es interesante revisar las implicaciones de pasar un objeto que potencialmente puede llegar a ser de un tamaño considerable. Efectivamente, al igual que hemos indicado para la asignación, es importante tener en cuenta que la copia de un objeto grande puede ser costosa. Tenga en cuenta que:

- Cuando se pasa un objeto por valor, se hace una copia. Si la estructura es muy grande, será necesario un tiempo y espacio que pueden ser considerables.

- Cuando devolvemos un objeto con **return** se hace una copia del objeto devuelto[3]. Por tanto, devolver una estructura puede ser una operación muy costosa.

La solución más directa y sencilla para resolver este problema es aprovechar el paso por referencia. La solución es similar a la propuesta para el tipo **vector** (véase sección 6.2.3 en la página 117). Básicamente, con el paso por referencia podemos evitar la creación de copias que harían nuestro código menos eficiente. Recordemos que:

- Si queremos pasar un objeto como dato de entrada, lo pasamos como referencia constante. De esta forma el código usa el objeto original —evita la copia— aunque la palabra **const** del parámetro garantiza que sólo se usará para consultar, sin posibilidad de modificarlo.
- Si queremos evitar la copia de un objeto en una sentencia **return**, podemos pasar un objeto por referencia de forma que la función sobreescriba el valor de este objeto con el resultado de la función.

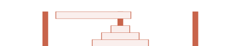
Esta discusión no está completa sin comentar —brevemente— que aunque en general podemos asumir que el comportamiento es el indicado, en la práctica podemos encontrarnos que la situación es mejor de lo esperado. En primer lugar, los compiladores de C++ están diseñados con mecanismos de optimización que pueden resolver o mejorar algunas de estas situaciones. Por otro lado, también es interesante recordar que en C++11 aparecen nuevas posibilidades para obtener mejor código. En cualquier caso, en este nivel de programación es importante entender este funcionamiento y saber enfrentarse a este problema proponiendo soluciones que garanticen una buena eficiencia.

Ejemplo 8.2.1 Escriba un programa que lea los datos de una persona desde la entrada estándar y vuelva a escribirlos en la salida estándar. Para ello, incluya funciones de lectura y escritura con sus correspondientes mensajes interactivos. Tenga en cuenta datos como la fecha de nacimiento, correo electrónico, nombre y sexo.

Sin entrar en más detalles sobre el tipo de solución deseada, a continuación se propone una solución donde aparecen distintos casos de paso y devolución de parámetros.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  // ------ Definición de tipos ------
6
7  struct Fecha {
8     int dia;
9     int mes;
10    int anio;
11 };
12
13 struct Persona {
14    string apellidos;
15    string nombre;
16    string email;
17    char sexo;          // M(asculino) / F(emenino)
18    Fecha nacimiento;
19 };
20
21 // ------ Funciones de E/S para fechas ------
22
23 Fecha LeerFecha()
24 {
25    Fecha f;
26    char dummy;
27
28    cin >> f.dia >> dummy >> f.mes >> dummy >> f.anio;
29
30    return f;
31 }
```

[3] Realmente no tiene por qué ser así, ya que el compilador podría generar código para evitarla, pero es posible que se realice.



```
32 void EscribirFecha(const Fecha& f)
33 {
34    cout << f.dia << '/' << f.mes << '/' << f.anio;
35 }
36
37 // ------ Funciones de E/S para personas ------
38
39 void LeerPersona (Persona& p)
40 {
41    cout << "Introduzca apellidos: ";
42    getline(cin,p.apellidos);
43    cout << "Introduzca nombre: ";
44    getline(cin,p.nombre);
45    cout << "Introduzca e-mail: ";
46    getline (cin,p.email);
47    cout << "Introduzca sexo (M/F): ";
48    cin >> p.sexo;
49    cout << "Introduzca fecha de nacimiento: ";
50    p.nacimiento= LeerFecha();
51 }
52
53 void EscribirPersona (const Persona& p)
54 {
55    cout << "Nombre: " << p.apellidos << ',' << p.nombre << endl;
56    cout << "e-mail: " << p.email << endl;
57    cout << "Sexo: " << p.sexo << endl;
58    cout << "Fecha de nacimiento: ";
59    EscribirFecha(p.nacimiento);
60    cout << endl;
61 }
62
63 // ------ Programa test ------
64 int main()
65 {
66    Persona p;
67
68    LeerPersona(p);
69    EscribirPersona (p);
70 }
```

En este programa podemos observar varios casos relacionados con la copia y asignación de estructuras. Podemos ver que:

- La función para leer una fecha —líneas 23-31— se ha implementado devolviendo la nueva fecha con `return`. En el punto de llamada —línea 50— podemos ver que el objeto devuelto se usa para asignar la estructura al campo `nacimiento`. Esta línea asigna los tres campos. A pesar de esta forma de devolver el objeto, el tamaño de un objeto de este tipo es relativamente pequeño. Además, observe que la lectura desde `cin` es realmente la parte de la función que más tiempo necesita.
- En la función de lectura de una fecha se ha declarado una variable —línea 26— para eliminar un carácter de la lectura. Hemos supuesto que el formato de entrada es *día/mes/año*, por lo que debemos eliminar el carácter '/' para leer el siguiente dato. Como sabe, todavía no estamos incorporando controles de detección de errores en E/S, por lo que simplemente ignoramos el carácter leído.
- En la función para leer un objeto de tipo `Persona` hemos pasado la estructura por referencia. En este caso no lo devolvemos con `return` —aunque podríamos— ya que es un objeto relativamente grande, por lo que pasamos por referencia el objeto que queremos modificar. Como puede ver en la línea 68, hay una llamada a esta función. El objeto `p` que pasamos será el que finalmente tenga el valor leído. Lógicamente, no existe ninguna copia de objetos de tipo `Persona`, pues las operaciones de la función `LeerPersona` se hacen directamente sobre `p`.
- Las funciones de escritura reciben un objeto como referencia constante. Tal vez considere posible realizar un paso por valor para escribir una fecha, ya que el objeto es relativamente pequeño. Sin embargo, no tiene ningún sentido ya que el paso por referencia garantiza que

funciona perfectamente y a menor costo. De hecho, en la práctica evitaremos el paso por valor de estructuras si no son de un tamaño pequeño.

Fin ejemplo 8.2.1 ■

Ejercicio 8.1 Considere el programa anterior, donde se leen y escriben los datos de una persona. Modifíquelo para que esa lectura/escritura se haga dos veces (por ejemplo, duplique las dos últimas líneas de **`main`**). ¿Qué ocurre? Proponga una solución.

Ejercicio 8.2 Escriba una función `Distancia` que recibe dos parámetros de tipo `Punto` y devuelve la distancia euclídea entre ellos. Recuerde que la distancia entre dos puntos p,q es $\sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$.

Ejercicio 8.3 Considere un nuevo tipo de dato `BoundingBox` definido para delimitar un rectángulo controlado por la esquina superior izquierda e inferior derecha. Use el tipo `Punto` anterior para definir esta nueva estructura. Una vez definida, escriba una función que lee un entero `n` y a continuación `n` objetos de tipo `Punto` para obtener el `BoundingBox` mínimo que los contiene. Observe que no es necesario definir ningún vector de puntos.

8.2.4 Ordenar estructuras

El aspecto más relevante de la ordenación de estructuras se refiere al problema de determinar el orden entre objetos. Lógicamente, una vez que lo hayamos determinado no tenemos más que seleccionar el algoritmo (véase sección 6.2.5, página 121) y aplicarlo sin necesidad de ningún cambio adicional.

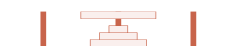
Supongamos que tenemos un vector `v` de estructuras a ordenar. El problema está en determinar, dados dos objetos —`v[i]` y `v[j]`— qué orden tienen. En la implementación con tipos simples podemos aplicar el operador `v[i]<v[j]` para ordenar los elementos según el orden preestablecido en dicho tipo, o establecer otro criterio mediante una función `orden(v[i],v[j])` que devuelva cuál es el orden de los elementos: `v[i]<v[j]`, `v[i]>v[j]` o `v[i]==v[j]`.

En principio, en C++ no se puede aplicar el operador relacional para conocer el orden de dos estructuras. Este comportamiento es lógico, ya que si definimos un nuevo tipo con **`struct`**, el compilador no puede conocer el sentido del nuevo tipo y, por tanto, la forma de ordenarlo. Un código como el siguiente:

```
struct  Persona {
   string apellidos;
   string nombre;
   string email;
   char sexo;        // M(asculino) / F(emenino)
   Fecha nacimiento;
};

//...
vector <Persona> v; // Vector a ordenar
//...

if (v[i]<v[j]) // Error, no se puede hacer la comparación
```

provoca un error de compilación en la última línea. Más adelante descubrirá que hay formas de resolverlo (de hecho, si aplica el operador < entre dos estructuras verá que el error de compilación no indica que sea algo desconocido, sino que no ha definido su significado.).

En su lugar, podemos definir una función que ordena dos estructuras. Esta función recibe dos estructuras y devuelve el orden que tienen. Hay tres posibilidades: menor, mayor o igual. La función puede devolver un entero como sigue:



```
int orden= OrdenPersonas (v[i],v[j]);

if (orden<0)
   cout << i << " es menor" << endl;
else if (orden>0)
   cout << j << " es menor" << endl;
else cout << "Son iguales" << endl;
```

Para implementar la función `OrdenPersonas` podemos considerar la solución más simple: establecer un campo por el que ordenar la estructura. Por ejemplo:

```
int OrdenPersonas (const Persona& l, const Persona& r)
{
   int orden;

   if (l.apellidos<r.apellidos)
      orden= −1;
   else if (l.apellidos>r.apellidos)
      orden= 1;
   else orden= 0;

   return orden;
}
```

establece que el campo `apellidos` determina el orden de dos estructuras.

Probablemente, ya se habrá dado cuenta de que es posible que un solo campo no sea suficiente. Otra forma es establecer varios campos junto con un orden que indica qué campos son prioritarios. Por ejemplo, para ordenar nombres, es decir, una estructura con al menos dos campos (`nombre` y `apellidos`) fijamos el campo `apellidos` como primer campo y el campo `nombre` como secundario. Es decir, podemos hacer lo siguiente:

```
int OrdenPersonas (const Persona& l, const Persona& r)
{
   int orden;

   if (l.apellidos<r.apellidos)
      orden= −1;
   else if (l.apellidos>r.apellidos)
      orden= 1;
   else if (l.nombre<r.nombre) // Apellidos iguales: miramos nombre
      orden= −1;
   else if (l.nombre>r.nombre)
      orden= 1;
   else orden= 0;

   return orden;
}
```

donde hemos establecido el campo `apellidos` como el principal y, en caso de que coincidan, ordenamos según el nombre.

Note que cuando dos estructuras tienen los campos `nombre` y `apellidos` idénticos, la función devuelve que son dos estructuras iguales. A pesar de ello, es probable que realmente sean distintas ya que tienen otros campos que seguramente no coinciden. Lógicamente, si incluimos todos los campos en este criterio, tendremos un orden completo para todos los objetos de este tipo. Por ejemplo:

```
int OrdenFecha (const Fecha& l, const Fecha& r)
{
   int orden;

   if (l.anio<r.anio)
      orden= −1;
   else if (l.anio>r.anio)
      orden= 1;
   else if (l.mes<r.mes) // Años iguales: miramos meses
      orden= −1;
   else if (l.mes>r.mes)
      orden= 1;
```

```
      else if (l.dia<r.dia) // Meses iguales: miramos días
         orden= -1;
      else if (l.dia>r.dia)
         orden= 1;
      else orden= 0;

      return orden;
   }
```

establece un criterio de ordenación de fechas habitual de forma que si devolvemos el valor cero es porque todos sus campos son idénticos.

Ejercicio 8.4 Suponga que disponemos de la función `OrdenFecha`. Escriba la función `OrdenPersonas` usando como único campo de ordenación la fecha de nacimiento. Se ordenarán por edad, por lo que la fecha de nacimiento posterior implica un orden menor. Una vez resuelto, escriba el algoritmo de ordenación por selección para ordenar un vector de objetos de tipo `Persona` según su edad.

Es importante observar que las soluciones que estamos aportando en esta sección son las más directas y simples para aprender a manejar estructuras. Si piensa en el código de ordenación (por ejemplo, el ejercicio anterior, 8.4) se dará cuenta de que necesitamos una función de ordenación para cada criterio que establezcamos. Por ejemplo, si en otra ocasión necesitamos ordenar por nombre, tendremos que definir una nueva función.

Estará de acuerdo en que esta solución aparenta ser muy redundante —de hecho lo es— ya que el código esencialmente es el mismo, aunque cambia un pequeño detalle. Más adelante encontrará mejores formas de realizarlo para eliminar esta redundancia.

Algoritmos de ordenación estables

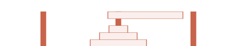
La ordenación de estructuras es una excelente motivación para la introducción de un concepto muy importante en los algoritmos de ordenación: la *estabilidad*.

Definición 8.1 — Ordenación estable. Un algoritmo de *ordenación es* **estable** si, considerando cualesquiera dos objetos *(l,r)* iguales y con *l* situado a la izquierda de *r* antes de ordenar, el algoritmo garantiza que el objeto *l* sigue estando a la izquierda del objeto *r* después de ordenar.

En principio parece que un algoritmo estable no ofrece ninguna ventaja, ya que si dos elementos son iguales no importa dónde se muevan. Tanto si el primero queda a la izquierda del segundo o al revés, la secuencia de elementos quedará ordenada. Efectivamente, en muchos casos no importa que el algoritmo sea estable o no, pero podemos encontrarnos aplicaciones donde la estabilidad es fundamental. Tal vez queramos ordenar los elementos pero respectando el orden previo en caso de elementos iguales.

Imagine que tenemos una lista de personas —usamos `struct Persona` anterior— que se han ido añadiendo a un vector por orden de llegada. Por ejemplo, podemos suponer una inscripción en alguna competición en la que tenemos un límite de participantes. Si queremos realizar una ordenación de personas por año de nacimiento podemos usar el siguiente criterio:

```
   int OrdenPersonas (const Persona& l, const Persona& r)
   {
      int orden;

      if (l.nacimiento.anio<r.nacimiento.anio)
         orden= -1;
      else if (l.nacimiento.anio>r.nacimiento.anio)
         orden= 1;
      else orden= 0;

      return orden;
```



```
}
```

donde dos personas son iguales si el año de nacimiento coincide.

Con esta función podemos aplicar un algoritmo de ordenación[4] para obtener una lista de personas que estarán agrupadas por año de nacimiento. Si el algoritmo es estable, cada persona estará ordenada junto con las del mismo año de nacimiento pero en el mismo orden inicial, es decir, el orden de llegada que queríamos respetar.

Ejercicio 8.5 Considere una estructura con dos campos `c1` y `c2`. Suponga que aplicamos primero un algoritmo de ordenación para ordenarlas sólo por el campo `c2` y posteriormente otro algoritmo de ordenación sólo por el campo `c1`. ¿Qué resultado obtendrá si usamos para ello algoritmos de ordenación estables?

Como puede ver, la estabilidad de un algoritmo de ordenación es una característica muy importante ya que puede determinar su elección para nuestras aplicaciones. Por eso, es importante tener en cuenta lo siguiente:

- El análisis de un algoritmo de ordenación también debe incluir un estudio de su estabilidad.
- El diseño e implementación de un algoritmo de ordenación debe realizarse con cuidado, ya que es posible que con una pequeña modificación podamos obtener un algoritmo estable, lo que resulta un resultado más deseable.

Ejercicio 8.6 Indique si los algoritmos de *selección*, *inserción* y *burbuja* son estables. En caso de que sea estable, ¿podría modificarse para que, aun obteniendo un resultado ordenado, deje de ser estable?

Algoritmo del cartero (postman'sort)

El algoritmo del cartero es un caso especial del algoritmo de *"ordenación por casilleros"* que aprovecha la división de la clave de ordenación en distintos campos. Aunque en la bibliografía puede encontrar este algoritmo descrito en otros términos, relacionado con otros algoritmos, o especializado en algún tipo de clave de ordenación, en esta sección es interesante presentarlo como un algoritmo que especializa la ordenación por casilleros (véase 6.4.3 en página 141) al caso de datos con distintos campos.

La idea está basada en la forma en que los carteros ordenan el correo tradicional para poder enviarlo. Como es de esperar, el correo se ordena dividiendo los envíos según el país de destino, después la ciudad, la localidad, etc. Es decir, la dirección de destino tiene una estructura jerárquica que se puede aprovechar para la ordenación.

Un caso sencillo de aplicación es la ordenación de estructuras. Cuando queremos ordenar una estructura usando varios campos establecemos un orden jerárquico indicando desde el campo principal hasta el campo menos significativo. El algoritmo del cartero se puede aplicar mediante una ordenación por casilleros a cada uno de los campos, desde el menos significativo hasta el principal.

Por ejemplo, para ordenar una estructura de tipo `Fecha` podemos realizarla en tres fases: ordenar por día, luego por mes y finalmente por año. Las fechas quedan ordenadas en el tiempo como si hubiéramos aplicado un algoritmo clásico como el de inserción apoyado en una función que establece el orden de las fechas (como la presentada en la página 171).

Ejemplo 8.2.2 Considere el problema de ordenar las personas por fecha de nacimiento. Escriba una función que implementa el algoritmo del cartero.

[4] Por ejemplo, los algoritmos *bucket sort* o *counting sort* pueden resultar muy adecuados en este problema.

La función de ordenación se puede aplicar en tres fase. Para resolver el problema, implementamos tres funciones independientes que ordenan un vector de objetos de tipo `Persona` por cada uno de los campos. La función que buscamos se podría escribir por tanto como sigue:

```
void OrdenarPersonas (vector<Persona>& v)
{
   OrdenarPorDia(v);
   OrdenarPorMes(v);
   OrdenarPorAnio(v);
}
```

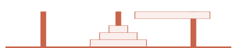
A primera vista, tal vez el lector quede un poco sorprendido con este código. Aplicamos un algoritmo de ordenación e inmediatamente aplicamos otro que vuelve a cambiar los elementos de lugar. Parece que estamos deshaciendo lo hecho. Sin embargo, es fundamental darse cuenta de que los algoritmos de ordenación que se aplican son *estables* y, por tanto, el orden establecido en un paso se respeta en el siguiente para objetos que sean iguales. Por ejemplo, dos personas nacidas el mismo año quedan con el orden establecido en la fase de ordenación por mes.

La función más interesante de entre las tres que se llaman es la del año, ya que el rango de años no está establecido. Para resolverlo podemos proponer la siguiente solución:

```
void OrdenarPorAnio (vector<Persona>& v)
{
   // Buscamos rango de años
   int min_anio= v[0].nacimiento.anio;
   int max_anio= v[0].nacimiento.anio;

   for (int i=1; i<v.size(); ++i) {
      int anio= v[i].nacimiento.anio;
      if (anio<min_anio)
         min_anio= anio;
      else if ((anio<max_anio)
            max_anio= anio;
   }

   // ------ Aplicamos bucket sort ------

   // Un casillero por año
   vector<vector<Persona> > buckets(max_anio-min_anio+1);

   // Volcamos a los casilleros
   for (int i=0; i<v.size(); ++i) {
      int b= v[i].nacimiento.anio - min_anio;
      buckets[b].push_back(v[i]);
   }

   // Volcamos de nuevo en v
   int dest= 0; // posición a sobreescribir
   for (int b=0; b<buckets.size(); ++b)
      for (int i=0; i<buckets[b].size(); ++i) {
         v[dest]= buckets[b][i];
         dest++;
      }
}
```

Es interesante que observe algunos detalles del algoritmo, especialmente comparándolo con la solución del algoritmo de ordenación por casilleros de la página 142 (sección 6.4.3):

- El número de casilleros es tan grande como el rango de posibles fechas que tenemos. Por ejemplo, si todas las fechas están situadas en una década, no necesitamos más de 10 casilleros. Imagine que las personas tienen fechas que se distribuyen a lo largo de milenios: el conjunto de casilleros sería muy alto. Sin embargo, esta implementación está considerando que el rango no es demasiado grande (observe que estamos interesados en procesar, por ejemplo, las direcciones de correo).
- En esta implementación no hemos ordenado cada uno de los casilleros (recuerde que este paso sí lo aplicamos en nuestra implementación de la sección 6.4.3). En este ejemplo el problema es mucho más simple, puesto que tenemos que ordenar un conjunto de valores



enteros. Si el rango de enteros es relativamente pequeño, es muy fácil crear un conjunto de casilleros que "empaqueten" valores idénticos, de forma que el vuelco de todos los casilleros garantiza que están ordenados. Cada casillero es el destino de un único valor entero. La velocidad de ordenación es muy alta: con dos pasadas, una para volcar a los casilleros y otra para volcar al vector original es suficiente.

- En ningún caso comparamos dos objetos de tipo `Persona`. La ordenación por casilleros permite una implementación que vuelca los elementos uno a uno en cada casillero y los devuelve al vector original ordenados. Por ello, el algoritmo final es muy eficiente. Recuerde los algoritmo clásicos, donde para situar un elemento teníamos que comparar con muchos otros a lo largo de un rango de posiciones del vector.

Ejercicio 8.7 Complete el ejemplo con las dos funciones de ordenación por días y meses.

Finalmente, es interesante observar que aunque el algoritmo es muy eficiente, es fácil proponer cambios que afectan al tiempo de ejecución. Por ejemplo, las copias de elementos pueden ser costosas; un algoritmo basado en la ordenación de un vector de índices simplificaría esas operaciones. Podemos, incluso, prescindir del vector de vectores. En las siguientes secciones volvemos sobre este problema.

Fin ejemplo 8.2.2 ■

8.3 Pares con la STL

En las secciones anteriores hemos descrito algunos ejemplos donde se han definido nuevos tipos de datos compuestos por dos objetos. Por ejemplo, el tipo `Punto` lo componen dos valores reales o el tipo `Circulo` que lo componen un número real y un punto. Podríamos incluso proponer otros ejemplos, como el tipo `Complejo` que contendría dos números reales. En todos ellos se crea un nuevo tipo para poder declarar objetos, así como funciones para realizar operaciones con ellos. Una vez definidos, se pueden usar en múltiples ocasiones, implementando distintos programas que aprovechan esos nuevos tipos.

Es bastante frecuente encontrarse con código donde se deben manejar pares de objetos, sin que sea necesario crear un nuevo tipo de dato explícitamente. Por ejemplo, tal vez queremos devolver un par de valores desde una función, y queremos hacerlo con la sentencia **`return`** en lugar de usar dos parámetros por referencia. En este caso podemos devolverlo como una estructura de dos campos, aunque tal vez no es fácil justificar la creación de un nuevo tipo de dato sólo para ello. Sólo queremos devolver los dos valores, sin incluir nuevos identificadores de tipo en nuestro programa.

Si necesitamos simplemente *"empaquetar"* dos objetos en nuestro programa, es muy sencillo realizarlo sin necesidad de crear una nueva "entidad" en nuestro diseño. Es esta sección se estudian los detalles.

8.3.1 El tipo *Pair*

Para resolver el problema, la STL incorpora un tipo genérico —disponible si incluye **`utility`** — que sirve para manejar cualquier par de valores en un mismo objeto. No tiene ningún objetivo concreto distinto a empaquetarlos para poder manejarlos como una única entidad. Para declarar un objeto como un par usamos la siguiente sintaxis:

pair<*tipo_1, tipo_2*> *nombre_objeto*;

que declara un nuevo objeto como un par que tiene dos campos, uno de tipo *tipo_1* y otro de tipo *tipo_2*. El acceso a cada uno de ellos se hace con los identificadores `first` y `second`. Es decir, puede considerar que el objeto se ha creado como la siguiente estructura:

```
struct {
   tipo_1 first;
   tipo_2 second;
};
```

Por otro lado, también es posible declarar un objeto de tipo `pair` incluyendo una inicialización para cada uno de sus campos. Para ello, no tenemos más que escribir los valores iniciales como sigue:

pair<*tipo_1, tipo_2*> *nombre_objeto* (expresión_1, expresión_2);

donde hemos declarado un nuevo objeto *nombre_objeto* que se inicializa con las dos expresiones incluidas entre paréntesis.

Por ejemplo, imagine que queremos escribir una función que devuelve las dos soluciones de una ecuación de segundo grado (véase sección 5.2.3 en la página 95). Se pueden devolver como un único objeto con una función como la siguiente:

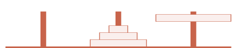
```
#include <cmath>   //  sqrt
#include <utility> //  pair
using namespace std;

// Suponemos ax^2+bx+c=0 tiene solución
pair<double,double> Ecuacion2Grado (double a, double b, double c)
{
   pair<double,double> sol;

   double discriminante= b*b - 4*a*c;
   sol.first= (-b+sqrt(discriminante)) / (2*a);
   sol.second= (-b-sqrt(discriminante)) / (2*a);

   return sol;
}
```

de forma que podríamos declarar un objeto de tipo `pair` para recibir el resultado de la función. Por ejemplo, en el siguiente código:

```
#include <cmath>    // sqrt
#include <utility>  // pair
#include <iostream> // cout
using namespace std;
// ...

int main()
{
   double a, b, c;
   //...

   pair<double,double> soluciones;

   soluciones= Ecuacion2Grado(a, b, c); // Suponemos que existen

   cout << "Soluciones: " << soluciones.first  << " y "
                          << soluciones.second << endl;
}
```

Es interesante notar que el tipo de dato puede incluir como componentes cualquier otro tipo, tanto si es del lenguaje, como de la STL, como definido por el usuario. Por ejemplo, podemos definir un período temporal como la unión de dos fechas definidas con las estructura que hemos visto en las secciones anteriores:

```
#include <iostream> // cin, cout
#include <utility>  // pair
using namespace std;
```



```
struct Fecha {
   int dia;
   int mes;
   int anio;
};

Fecha LeerFecha()
{
   // ...
}
void EscribirFecha(const Fecha& f)
{
   // ...
}

pair<Fecha,Fecha> LeerPeriodo()
{
   pair<Fecha,Fecha> periodo;

   periodo.first= LeerFecha();
   periodo.second=  LeerFecha();

   return periodo;
}

void EscribirPeriodo(const pair<Fecha,Fecha>& p)
{
   EscribirFecha(p.fist);
   cout << ' ';
   EscribirFecha(p.second);
}
```

8.3.2 Operaciones con *pair*

El tipo de dato **pair** ha sido diseñado de forma genérica, por lo que no incluye ningún tipo de operación especial. Lógicamente, el hecho de haberse definido de una forma simple mediante una estructura con dos campos implica su uso como se muestra en las secciones anteriores (copias, asignaciones y acceso a los campos). Para hacer más fácil su uso, se han incluido algunas capacidades adicionales pero sin darle significado al nuevo tipo.

Literales de tipo *pair*

La forma de expresar un "literal" para un tipo **pair** concreto se realiza escribiendo el tipo junto con los dos valores que queremos asignar a sus campos. Por ejemplo, podemos hacer:

```
pair<double,double> p;
// ...

p= pair<double,double>(1.5,2.5);
```

para asignar un literal con dos campos **double** inicializados con los valores `1.5` y `2.5`.

Además, se ha añadido una función para crear un objeto de tipo **pair** con los dos valores, sin necesidad de escribir explícitamente los nombres de los tipos. En el ejemplo anterior, podríamos haber hecho:

```
pair<double,double> p;
// ...

p= make_pair(1.5,2.5);
```

con idéntico resultado. Observe que en este caso es el compilador el que determina el tipo de objeto que tiene que devolver a partir de los parámetros de la función.

Asignación

Como hemos visto, es posible asignar una estructura a otra (se asigna campo a campo). El los ejemplos mostrados, hemos asignados dos objetos de tipo **pair** del mismo tipo.

En principio, no resulta en absoluto novedoso, ya que es consecuencia de su definición como estructura. Sin embargo, es importante tener en cuenta que en el caso del tipo **pair** se pueden asignar pares que sean de *"tipos compatibles"*. Dos pares son asignables si campo a campo se pueden asignar. Esto implica que aunque los tipos base de los pares sean distintos, la asignación es válida si el compilador acepta las asignaciones de sus campos. Por ejemplo, considere el siguiente ejemplo:

```
#include <utility>  // pair
#include <iostream> // cout
using namespace std;

int main()
{
   pair<double,double> reales(1.5,2.5);
   pair<int,int> enteros;

   enteros= reales;

   cout << "Enteros: " << enteros.first << " y " << enteros.second << endl;
}
```

donde los dos tipos **pair** son distintos. A pesar de ello, la asignación se realiza sin ningún problema ya que un objeto **double** se puede asignar a un objeto **int** (aunque perdiendo la parte fraccionaria). El este caso se escribiría el mensaje:

Operadores relacionales

Como se ha indicado en las secciones anteriores, por defecto, dos estructuras no se pueden comparar. Si intenta comprobar la relación entre dos estructuras mediante un operador de relación descubrirá que el compilador genera un mensaje de error[5].

Dado que el tipo **pair** se ha creado para empaquetar dos objetos, sin intención de crear un nuevo tipo de dato, podemos pensar que la operación de comparación entre dos pares no tiene ningún sentido. Sin embargo, es fácil encontrarse casos en los que conviene tener el tipo con algún orden. Por ejemplo, podemos tener múltiples pares en un vector e intentar buscar si incluye cierto par de valores. Si estuviera ordenado, podríamos aplicar un algoritmo rápido de búsqueda que nos indicaría si el elemento está.

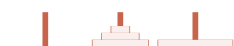
El tipo **pair** incluye la posibilidad de aplicar los operadores relacionales siempre que los tipos que lo componen permitan dichos operadores[6]. Para ello asume que el campo *first* es prioritario. En caso de que el campo *first* sea igual, el campo *second* determina el orden. Por tanto, si define pares de valores, recuerde que el orden de los campos determinará el orden de las parejas (en caso de que desee aplicarlo).

8.3.3 Diccionarios

La introducción del tipo **pair** nos permite presentar un concepto muy importante: el de *diccionario*. El lector conoce bien los diccionarios habituales, donde hay un conjunto ordenado de entradas que corresponden a pares con una *clave* por un lado, y una *información asociada* a dicha clave por otro. El tipo **pair** nos puede ayudar a gestionar diccionarios, es decir, un conjunto de pares *(clave,valor)* almacenados en los campos *first* y *second*, respectivamente.

[5] Como comprobará cuando profundice en C++, en realidad se puede programar el comportamiento de estos operadores de forma que la operación sea válida.

[6] Realmente el orden existe siempre que los campos admitan, concretamente, los operadores == y <.



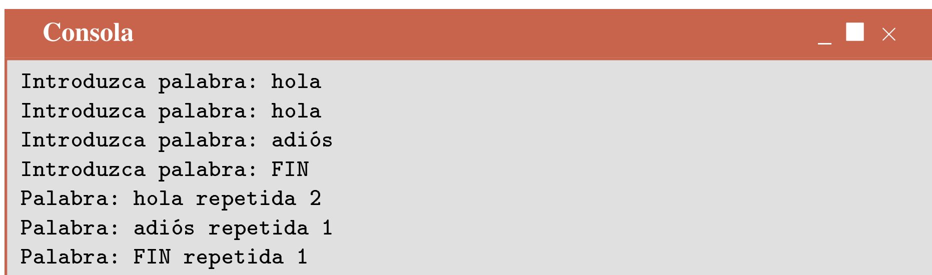
En este capítulo no vamos a introducir complejas y elaboradas estructuras que nos facilitan el manejo de diccionarios. Sin embargo, podemos crear soluciones que mantienen diccionarios como vectores de pares, de forma que afiance sus conocimientos sobre vectores, pares y el concepto de diccionario.

Ejemplo 8.3.1 Se desea obtener un listado de todas las palabras que se introducen por teclado junto con la frecuencia asociada, es decir, el número de veces que se han introducido. Escriba un programa que lea una serie de palabras hasta que se introduzca la palabra *"FIN"* y obtenga un listado de todas ellas junto con su frecuencia.

Una posible solución del problema puede ser la siguiente:

```
#include <utility>  // pair
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Busca la clave en el diccionario y devuelve su localización
int BuscarInsertar (vector<pair<string,int> >& diccionario,
                                 const string& clave)
{
   for (int i=0; i<diccionario.size(); ++i)
      if (diccionario[i].first == clave)
         return i;
   diccionario.push_back(pair<string,int>(clave,0));
   return diccionario.size()−1; // No estaba, la pusimos al final
}

int main()
{
   vector<pair<string,int> > dicionario;
   string palabra;

   while (palabra!="FIN") {
      cout << "Introduzca palabra: ";
      cin >> palabra;
      int indice= BuscarInsertar(dicionario,palabra);
      dicionario[indice].second++;
   }

   for (int i=0; i<diccionario.size(); ++i)
      cout << "Palabra: " <<  diccionario[i].first
           << " repetida " << diccionario[i].second << endl;
}
```

que daría lugar, por ejemplo, a una ejecución como la siguiente:



Observe que la función `BuscarInsertar` no sólo busca, sino que si no encuentra la palabra en el diccionario la inserta al final con un contador de cero. Esta función de búsqueda puede resultar ineficiente al realizar una búsqueda lineal que puede necesitar recorrer todo el vector.

Por otro lado, note que en la ejecución las palabras que se listan están en el orden en que se introdujeron, pues se añaden al vector al final. Además, podríamos cambiar el programa principal para evitar introducir la palabra `"FIN"`, por ejemplo, con una lectura adelantada.

Resulta especialmente interesante mantener los diccionarios ordenados por la clave correspondiente (el campo `first` en nuestro ejemplo). De esta forma, la búsqueda podría resultar mucho más rápida. Además, un listado de todos los elementos del diccionario aparecería ordenado por la clave.

Ejercicio 8.8 Modifique la función `BuscarInsertar` del ejercicio anterior de forma que el vector de pares esté siempre ordenado. Para implementar la función, use el algoritmo de búsqueda dicotómica (sección 6.2.4 en página 120) a fin de mejorar la eficiencia.

Fin ejemplo 8.3.1 ■

Si analizamos la idea de un **`vector`** no es más que una estructura que organiza los datos asignando un índice a cada objeto. Por tanto, podemos interpretar que no es más que un caso particular de *diccionario*, donde la clave de búsqueda es un entero que empieza en cero y termina en el número de elementos menos uno. Desde otro punto de vista, un diccionario es una generalización de un vector donde se han ampliado las posibilidades del índice para no limitarse a enteros. Por ello, en algunas referencias puede encontrar los diccionarios como *arrays asociativos*, ya que no son más que *arrays* —vectores— que asocian un contenido a cada *clave*.

Algunos lenguajes contienen directamente implementaciones que facilitan estas asociaciones. El lenguaje C++ ofrece tipos desarrollados específicamente para manejar diccionarios (véase el tipo **`map`** de la STL) de forma que la eficiencia de operaciones como la búsqueda o la inserción son muy buenas. Por supuesto, no tan rápidas como el acceso a una posición en un **`vector`** a partir de un entero.

Ordenación por conteo (counting sort)

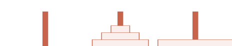
El algoritmo de ordenación por conteo —en inglés *counting sort*— generalmente se presenta como un algoritmo de ordenación de enteros pequeños. Imaginemos que queremos ordenar un conjunto de enteros en un rago relativamente pequeño[7]. Por ejemplo, tenemos una serie de miles de enteros, todos con valores del 0 al 99. En lugar de compararlos entre sí, podemos realizar un conteo de cuántos datos hay de cada valor:

1. Creamos 100 índices y contamos cuántos datos hay de cada número. Por ejemplo, tal vez haya 10 ceros, 20 unos, 5 doses, etc.
2. Calculamos en qué posición del vector ordenado se situaría cada una de las secuencias de valores 0, 1, 2, etc. Por ejemplo, los ceros deben situarse a partir del índice cero, los unos a partir del índice 10 (saltándose los 10 ceros que hemos contado que hay), los doses a partir del índice 30 (saltándose los 10 ceros + 20 unos), los treses a partir del índice 35 (saltándose los 10 ceros + 20 unos + 5 doses), etc.
3. Recorremos el vector original volcándolo al vector ordenado. Cada vez que encontramos un valor k, lo situamos en su correspondiente tramo, y avanzamos el índice para cualdo aparezca un nuevo k.

Esta idea se muestra gráficamente en la figura 8.1, donde podemos ver el vector original desde donde se copian los elementos a un segundo vector resultado. Para ello, se ha usado un vector auxiliar de conteo. Observe en la figura que es fácil deducir que el algoritmo es *estable*.

La implementación de este algoritmo es bastante simple. La parte más interesante, probablemente, es el volcado de los elementos originales al vector ordenado. Para ello, se puede usar un vector de índices que indica para cada entero a qué posición le toca copiarse. Así, el índice cero

[7] El rango de posibles valores no debería ser mucho más grande que el número de datos a ordenar.



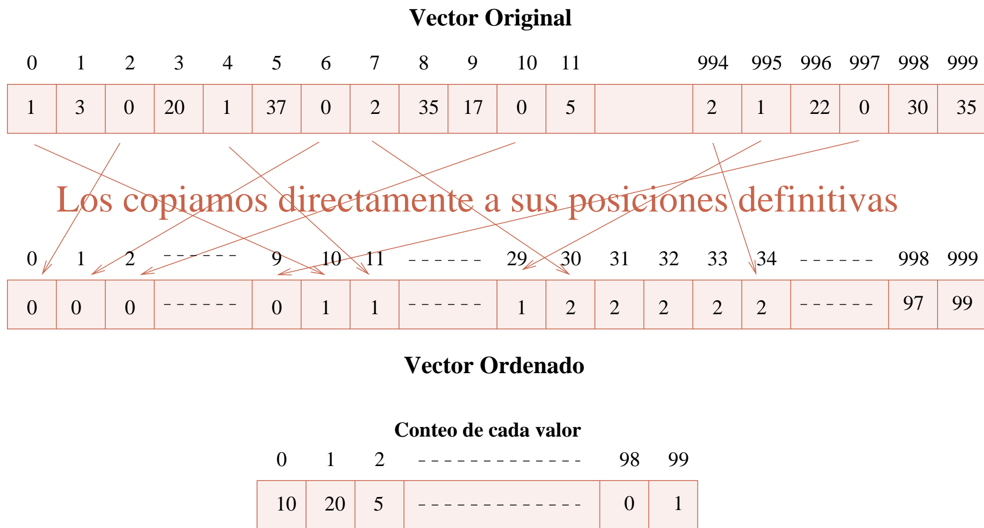


Figura 8.1
Ordenación por conteo.

indica a dónde irá el siguiente cero, el índice uno a dónde irá el siguiente uno, etc. Este índice se puede calcular a partir del vector de contadores como sigue:

```
vector<int> indice(100,0); // Un contador a cero por entero

// Contamos
for (int i=0; i<v.size(); ++i)
   ++indice[v[i]];

// Preparamos contadores para actuar como índice destino
int acumulado= 0; // Para cada i, acumula los anteriores a i
for (int i=0; i<indice.size(); ++i) {
   int solo_en_i= indice[i];
   indice[i]= acumulado;
   acumulado= acumulado + solo_en_i;
}
```

Este algoritmo recuerda al algoritmo de ordenación por casilleros (sección 6.4.3, página 141). La diferencia es que aquí creamos un casillero para cada entero, pero sólo apuntamos cuántos elementos le corresponden. Esta simplificación se debe a que queremos ordenar en base a un número pequeño de números enteros. Si recuerda, en la sección 8.2.4 (página 174) hemos usado la ordenación por casilleros para ordenar una serie de estructuras por año. Para ordenar estos objetos también podemos usar el algoritmo por conteo, ya que la clave de ordenación corresponde a un número entero y el número de año se supone relativamente pequeño. Una implementación equivalente a la anterior podría ser:

```
void OrdenarPorAnio (vector<Persona>& v)
{
   // Buscamos rango de años
   int min_anio= v[0].nacimiento.anio;
   int max_anio= v[0].nacimiento.anio;

   for (int i=1; i<v.size(); ++i) {
      int anio= v[i].nacimiento.anio;
      if (anio<min_anio)
         min_anio= anio;
      else if (anio<max_anio)
               max_anio= anio;
   }
```

```
    // ------ Aplicamos counting sort ------

    // Un contador por año, inicializado a 0
    vector<int> indice(max_anio-min_anio+1, 0);

    // Contamos cuántos datos de cada año
    for (int i=0; i<v.size(); ++i) {
       int b= v[i].nacimiento.anio - min_anio;
       ++indice[b];
    }
    // Preparamos contadores para actuar como índice destino
    int acumulado= 0;
    for (int i=0; i<indice.size(); ++i) {
       int solo_en_i= indice[i];
       indice[i]= acumulado;
       acumulado= acumulado + solo_en_i;
    }

    // Volcamos los datos en un vector ordenado
    vector<Persona> ordenado(v.size()); // Tanto elementos como v
    for (int i=0; i<v.size(); ++i) {
       int b= v[i].nacimiento.anio-min_anio;
       ordenado[indice[b]]= v[i];
       ++indice[b];
    }

    v= ordenado; // El resultado sobreescribe el original
}
```

En esta función hemos calculado el mínimo y máximo de los años a fin de determinar el rango de enteros válidos, de forma que se pueda manejar un vector de índices para cada uno de ellos.

Finalmente, es interesante que observe que el vector v final no está totalmente ordenado por fecha de nacimiento, sino únicamente por año. Si estamos interesados en el orden completo, teniendo en cuenta mes y día de nacimiento, podríamos usar el vector v semi-ordenado en otro algoritmo de ordenación. Por ejemplo, si aplicamos la ordenación por inserción el número de desplazamientos de la clave estaría más contenido, ya que las claves están agrupadas. Recuerde que el algoritmo de ordenación por inserción es especialmente bueno cuando tenemos un vector ordenado o casi ordenado.

Ordenación por conteo generalizada

Si piensa en la solución para ordenar un vector de personas por año —propuesta en la sección anterior— descubrirá que el algoritmo de ordenación por casilleros anteriormente descrito se ha adaptado al conteo para poder ordenar los elementos sin necesidad de casilleros. Lo único necesario ha sido poder acumular para cada año un contador de repeticiones. Ha sido fácil, pues era un *"rango pequeño"* de enteros, con el único inconveniente de no empezar en cero.

Realmente este conteo se puede generalizar para otros tipos de datos que no sean enteros. De hecho, el algoritmo de ordenación por casilleros no se ha diseñado para un grupo de enteros (véase 6.4.3, en página 141).

Si el conjunto de datos a ordenar es relativamente pequeño y con muchos repetidos, es muy fácil y eficiente establecer un conjunto de pares $(dato, frecuencia)$ para obtener una serie de contadores que nos permitan aplicar el algoritmo de ordenación por conteo.

```
void Ordenar(vector<string>& v)
{
   vector<pair<string,int> > conteo; // Diccionario ordenado por first

   for (int i=0; i<v.size(); ++i) {
      int indice= BuscarInsertar (conteo,v[i]);
      conteo[indice].second++;
   }

   // Preparamos contadores para actuar como índice destino
   int acumulado= 0;
```

```
        for (int i=0; i<conteo.size(); ++i) {
            int solo_en_i= conteo[i].second;
            conteo[i].second= acumulado;
            acumulado= acumulado + solo_en_i;
        }

        // Volcamos los datos en un vector ordenado
        vector<string> ordenado(v.size()); // Tanto elementos como v
        for (int i=0; i<v.size(); ++i) {
            int indice= BuscarInsertar (conteo,v[i]);
            ordenado[conteo[indice].second]= v[i];
            conteo[indice].second++;
        }

        v= ordenado; // Reescribimos el vector v
    }
```

Observe que en este ejemplo es fundamental que el diccionario corresponda a un vector ordenado por el campo `first`, ya que ese orden determina la forma en que se empaquetarán las secuencias de cadenas en el vector ordenado final.

8.4 Problemas

Problema 8.1 Defina una estructura `Tiempo` que contenga tanto la hora como la fecha. Tenga en cuenta que éstas pueden ser definidas, a su vez, como estructuras.

Problema 8.2 Defina una estructura `Segmento` a partir de dos puntos. Escriba una función que recibe un segmento y devuelve su longitud.

Problema 8.3 Se desea programar la gestión de las cuentas de un banco. Para resolverlo, se crean dos estructuras:

- `Movimiento`. Está compuesta por una hora, una fecha, una cantidad (con signo) y un concepto (anotación asociada al movimiento).
- `Cuenta`. Está compuesta por varias cadenas que componen la cuenta (entidad, sucursal, número de cuenta y dígitos de control). Además, contiene todos los movimientos realizados en la cuenta desde su creación.

Teniendo en cuenta esta información, resuelva los siguientes puntos:

1. Defina las estructuras necesarias para poder disponer del tipo `Movimiento` y `Cuenta`.
2. Escriba una función que recibe la información de una cuenta y devuelve el saldo actual de la cuenta.

Problema 8.4 Se desea gestionar el conjunto de productos de una tienda. Para ello, defina una estructura `Producto` que está compuesta por un nombre, número de unidades disponibles y precio. Con esta estructura:

1. Escriba una función que recibe un vector de productos y el nombre de uno de ellos y devuelve si existe tal producto.
2. Escriba una función que recibe un vector de productos y un nuevo producto y lo añade al conjunto. En caso de que ya exista, deberá actualizar las unidades disponibles y su precio.

Para ambas funciones tenga en cuenta que el vector de productos está ordenado por el nombre del producto.

Problema 8.5 Para gestionar un almacén se dispone de un vector de productos como el indicado en el problema 8.4 indicando el stock disponible. Un pedido se almacena como un vector de pares. Cada par incluye información del nombre del producto y número de unidades. Con estos tipos de datos, resuelva lo siguiente:

1. Escriba una función que recibe un vector de productos (stock) y un pedido. Como resultado devuelve si se puede servir el pedido y en caso de poder realizarse, se actualiza el estado del almacén.

2. Unir los contenidos de dos almacenes. Tenga en cuenta que debe realizarse eficientemente, aprovechando que los dos almacenes tienen sus productos ya ordenados (véase sección 6.2.5 en página 131).

9 Recursividad

Introducción 185
Ejemplos de funciones recursivas 186
Gestión de llamadas: la pila 189
Diseño de funciones recursivas 193
Recursivo vs iterativo 201
Ejemplos de funciones recursivas 203
Problemas 208

9.1 Introducción

El concepto de recursividad no es nuevo. En el campo de la matemática podemos encontrar muchos ejemplos relacionados con él. Por ejemplo, muchas definiciones matemáticas se realizan en términos de sí mismas. Considérese el clásico ejemplo de la función factorial:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases} \tag{9.1}$$

donde la definición se realiza, por un lado, para un caso que denominamos base ($n = 0$) y por otro, para un caso general ($n > 0$). En este último, la formulación es claramente recursiva, es decir, en su definición utilizamos la misma función factorial.

Consideremos, por otro lado, el método de demostración por inducción. Generalmente, sólo necesita realizar: primero, una demostración para un caso base; segundo, suponiendo que está demostrado para casos menores, un caso general de tamaño n (aprovechando la ventaja de saberlo demostrado para cualquier caso menor que n).

De una manera similar, veremos que cuando programamos una función recursiva podemos suponer que está resuelta para un tamaño menor. Se podría afirmar que la gran virtud de la programación recursiva radica en que, para resolver un problema, usar esta técnica implica que ya "tenemos resuelto" el problema (de menor tamaño como ya veremos). De esta forma, la recursividad constituye una de las herramientas más potentes en programación.

Básicamente, un problema podrá resolverse de forma recursiva si es posible expresar una solución al problema —algoritmo— en términos de él mismo, es decir, que para obtener la solución será necesario resolver previamente el mismo problema sobre un conjunto de datos o entrada de menor tamaño.

En principio, podemos decir que los problemas recursivos que son más fáciles de resolver son aquellos que de forma natural se formulan recursivamente. Estos problemas ofrecen directamente una idea no sólo del problema sino también de cómo debe ser el algoritmo recursivo que lo resuelve. Por ejemplo, considere el caso del factorial: por definición nos dice que hay un caso base ($n = 0$) y

un caso recursivo ($n > 0$) que además se puede resolver si, habiendo resuelto $(n-1)!$, realizamos una multiplicación por n.

Por otro lado, hay problemas que no se expresan de forma recursiva y, por tanto, para resolverlos con funciones recursivas será necesario formular un algoritmo que se exprese en términos de sí mismo. Por tanto, este tipo de problemas requiere un esfuerzo adicional, ya que tenemos que idear esta solución recursiva. Lógicamente, está especialmente indicado para los que la solución no recursiva es especialmente compleja.

Aunque crea que se encontrará pocos casos de soluciones recursivas, descubrirá que son muy numerosos, especialmente aplicando algún tipo de técnica de diseño de algoritmos que nos ofrezca soluciones recursivas de forma natural (por ejemplo *divide y vencerás*, ver Brassard[3]).

De hecho, ya se ha encontrado con posibles algoritmos recursivos. Un ejemplo que hemos visto y se adecúa a esta descripción es el problema de localizar un elemento dentro de un vector de elementos ordenados. Es claro que el algoritmo se podría resolver fácilmente —como ya sabe— de forma iterativa. Diseñar este algoritmo aplicando la idea de *divide y vencerás* nos lleva de forma natural a expresar de forma recursiva el algoritmo de búsqueda binaria o dicotómica. Éste consiste en la división del vector en dos partes para seleccionar el subvector donde debe estar el elemento a buscar —según la ordenación— para continuar buscando habiendo descartado la mitad de los elementos. Es decir, estamos expresando el algoritmo en términos de resolver el mismo problema sobre el subvector.

9.1.1 Recursión directa e indirecta

La definición formal de función recursiva es el de una función que se llama a sí misma; la función *Factorial* es recursiva porque contiene una llamada a la función *Factorial*. En este caso decimos que es un caso de recursión directa, ya que podemos ver la llamada a la función directamente en el cuerpo que la define.

Otra forma de recursión es la denominada recursión indirecta; consiste en la llamada indirecta de una función a sí misma, es decir, sin que aparezca una llamada directamente en el cuerpo que la define. Por ejemplo, la función *A* llama a la función *B* que llama a la función *A*. En la práctica podría consistir en un ciclo mucho más complicado que pasa por varias funciones antes de desembocar en una llamada a la función original.

En este tema nos centraremos en problemas de recursividad directa. A pesar de ello, la mayor parte de las ideas que se presentan —si no todas— son útiles para el diseño de funciones recursivas indirectas, ya que básicamente son funciones recursivas diseñadas de una forma más elaborada.

9.2 Ejemplos de funciones recursivas

Antes de estudiar en profundidad la forma de obtener una función recursiva para solucionar un problema, es importante que el lector se familiarice con las funciones recursivas. En esta sección, presentamos algunos ejemplos muy simples para mostrar el mecanismo de funcionamiento y los conceptos fundamentales que en las siguientes secciones se analizarán.

9.2.1 Factorial de un número entero

El cálculo del factorial de un número entero se puede resolver fácilmente de forma iterativa como sigue[1]:

[1] No vamos a discutir sobre la posibilidad de que el valor obtenido desborde la capacidad del tipo **int**. En una implementación real se deberá tener cuidado para que el valor a calcular no sea demasiado grande o, en su caso, usar otro tipo entero distinto.

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

Sin embargo, la definición recursiva de `Factorial` (ecuación 9.1, página 185), sugiere claramente un algoritmo recursivo, ya que para resolverlo se debe resolver un problema idéntico, aunque de menor tamaño. Directamente, la función es la siguiente:

```
int Factorial (int n)
{
   if (n==0)
      return 1;
   else
      return n * Factorial(n-1);
}
```

Aunque se use la misma función que se define, el funcionamiento de las llamadas no difiere del que presentamos en las lecciones anteriores. Cuando se llama a esta función, se crea un nuevo objeto de tipo **int** como copia del argumento en la llamada —lógicamente, se crea en la pila—. Si este valor es cero termina la función, y si no, se vuelve a "abrir" una nueva llamada con un nuevo argumento. Si representamos esta situación para el valor *2*, con el mismo esquema gráfico que se ha mostrado en temas anteriores, el resultado sería el de la figura 9.1.

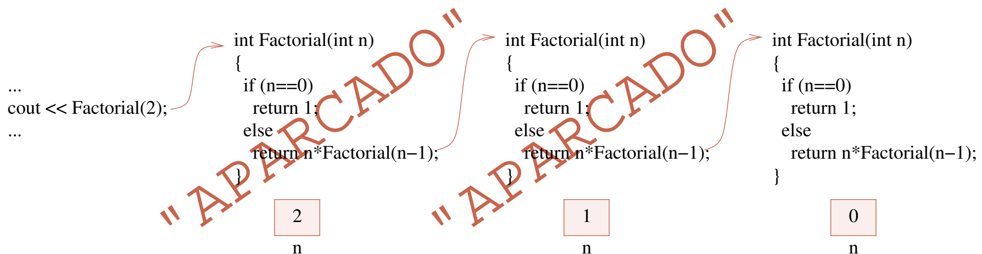


Figura 9.1
Situación tras la tercera llamada.

La situación representa el estado que se alcanza cuando se han realizado tres llamadas recursivas. En la última, el valor de `n` vale cero y no seguirán realizándose más, ya que termina devolviendo el resultado *1*. En este punto decimos que se ha alcanzado un caso base, puesto que no "avanza" más y se devuelve la llamada recursiva para seguir con el código de llamada. En nuestro ejemplo, después de esa vuelta, sólo es necesaria una multiplicación para seguir retornando. En la figura 9.2 se representa gráficamente esta situación, incluyendo los valores que se devuelven como resultado a la llamada original.

Hemos presentado este esquema para mostrar que el mecanismo es el mismo que se lleva a cabo en cualquier llamada a función. No obstante, una representación más realista no mostraría varios listados de la función `Factorial` —en el código ejecutable sólo hay una— sino que las llamadas se gestionarían a través de la pila, de forma que cada una de ellas provocaría un nuevo apilamiento.

Ejercicio 9.1 Considere el problema de calcular la potencia m^n, donde m y n son dos valores enteros. En el caso general, ese cálculo se puede definir como $m \cdot m^{n-1}$. Escriba una función recursiva que lo implemente. Recuerde que debe incluir un caso base.

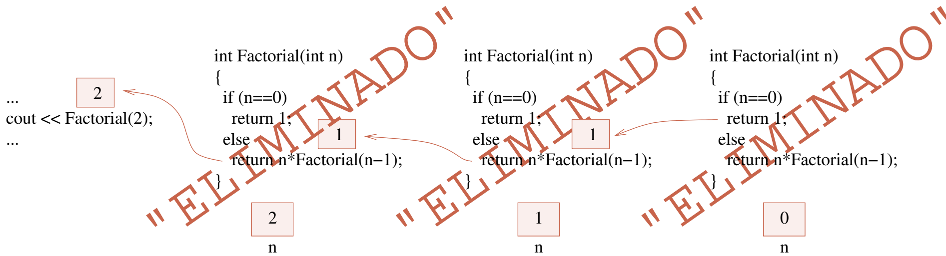


Figura 9.2
Situación tras la vuelta de las llamadas.

9.2.2 Suma de los elementos de un vector

La suma de los elementos de un vector se puede resolver con un bucle que recorre los elementos y los va acumulando en una variable auxiliar. En esta sección, presentamos otro algoritmo distinto para que se escriba en términos de sí mismo (sea recursivo). Lógicamente, en la práctica, será el primero el que se utilice.

En general, la suma de los elementos de un vector v con n elementos se puede resolver mediante la suma de dos cantidades: la primera, la suma de los elementos desde el 0 al elemento central; y la segunda, la suma de los elementos desde el central al último. Como vemos, el problema de calcular la suma de los elementos desde la posición 0 a la posición $n-1$ se escribe en términos de la suma de los elementos de un segmento del vector (el mismo problema, más pequeño). Por tanto, es posible escribir una función recursiva que siga este algoritmo.

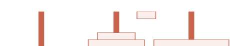
Recordemos que las llamadas recursivas avanzan desde un caso general hasta llegar a un caso base que también debe ser incluido. En nuestro caso, el segmento del vector a sumar se va haciendo más corto, hasta llegar a un problema simple que no requiere llamadas recursivas. Para nuestro problema, podemos seleccionar como caso base que el segmento de vector a sumar sea de un solo elemento. La solución para este caso es trivial, ya que la suma es, precisamente, ese elemento.

Con estas consideraciones, una posible solución al problema es la siguiente:

```
int Suma(const vector<int>& v, int desde, int hasta)
{
   if (desde==hasta)
      return v[desde];
   else {
      int centro= (desde+hasta)/2;
      return Suma(v,desde,centro) + Suma(v,centro+1,hasta);
   }
}
```

donde vemos que se realizan dos llamadas recursivas, ya que la solución del problema requiere resolver dos "subproblemas".

Es especialmente interesante que observe la cabecera de la función. Como puede ver, no es simplemente una función que recibe un vector y devuelve la suma, sino que además recibe dos valores enteros para indicar la parte del subvector que queremos sumar. La llamada que obtiene la suma de todo el vector pasará como índices el 0 y el tamaño menos 1. Aunque la primera idea pueda ser que una función que suma los elementos de un vector sólo tiene un parámetro, no podemos olvidar que nuestra solución recursiva se usa no sólo para resolver el problema original, sino también cada uno de los subproblemas. Para éstos, es fundamental determinar el rango de valores a sumar.



9.2.3 Listado de los elementos de un vector

Se desea escribir —en la salida estándar— los elementos de un vector de enteros, desde el último al primero (es decir, al revés). La solución obvia —que todos usaríamos— sería un bucle que recorriera los elementos del vector en ese orden. Sin embargo, en esta sección forzamos una solución distinta basada en un algoritmo recursivo.

Para escribir los elementos al revés, podemos escribir los elementos del subvector derecho (sin contar el primer elemento) al revés, y luego escribir el primer elemento. Esta solución requiere resolver el mismo problema para un tamaño más pequeño. Podemos expresarla con una función como la siguiente:

```
void VectorAlReves(const vector<int>& v, int desde)
{
   if (desde<v.size()) { // Hay elementos a escribir
      VectorAlReves(v,desde+1);
      cout << v[desde] << endl;
   }
}
```

En este ejemplo hemos seleccionado un caso base muy simple: cuando no hay que escribir ningún elemento. Si el valor del índice indica que no quedan elementos, la escritura del subvector es, simplemente, no hacer nada. Así, el caso base se da cuando `desde==v.size()`, y el caso general cuando hay, al menos, un elemento a escribir (`desde<v.size()`).

Ejercicio 9.2 Escriba una función recursiva `VectorAlReves` que tenga como caso base que sólo hay un elemento a imprimir.

Ejercicio 9.3 Un algoritmo recursivo, también válido para este listado, consiste en listar los elementos de la mitad derecha del vector al revés, seguidos de los elementos de la mitad izquierda al revés. Escriba una función que implemente este algoritmo.

9.3 Gestión de llamadas: la pila

La *pila* es la estructura básica responsable de que la recursividad funcione. Realmente ya hemos trabajado con funciones y el lector debería tener una idea más o menos intuitiva de cómo funciona el mecanismo de llamada y retorno de funciones. En esta sección vamos a mostrar con detalle cómo el sistema gestiona las llamadas entre funciones a fin de tener una idea muy precisa del mecanismo responsable de que la recursividad funcione correctamente.

Una pila es una estructura de datos que nos permite organizar múltiples datos de una forma muy concreta. Al igual que cuando apilamos platos situamos uno encima de otro, añadir uno es situarlo encima de los demás y retirar uno es eliminar el último añadido, la pila nos permite añadir y eliminar datos con ese mismo esquema. Básicamente, una pila está especialmente diseñada para:

- Añadir un elemento. La operación consiste en apilarlo, es decir, situarlo encima del último que se añadió.
- Eliminar un elemento. La operación consiste en desapilarlo, es decir, eliminar el último añadido.

A la posición donde se añade —o de donde eliminamos un elemento— se denomina el *tope de la pila*.

9.3.1 La pila del sistema

Para que un programa funcione se necesita una *pila*. La pila se sitúa en una posición de memoria y crece o decrece automáticamente conforme el programa necesita y libera memoria. En concreto, la gestión de las llamadas a función se resuelve fácilmente con este tipo de estructura.

Aunque el apilamiento sugiere el crecimiento de la pila, y éste crecimiento sugiere que se van añadiendo nuevos datos en posiciones de memoria crecientes, la implementación de la pila no tiene que realizarse así. En muchos casos, se sitúa la pila en una posición de memoria y conforme se van añadiendo elementos el tope se mueve hacia posiciones de memoria más bajas. En la práctica, este detalle de implementación no nos afecta. En los ejemplos sucesivos iremos mostrando el contenido de la pila simplemente añadiendo valores en un gráfico que simula el apilamiento de datos desde el fondo de la pila hasta el tope.

Para verlo desde un punto de vista práctico, vamos a analizar el comportamiento de las llamadas sucesivas en el siguiente código:

```
1  #include <iostream>
2  using namespace std;
3
4  int Maximo(int a, int b) {
5     return (a>b) ? a : b;
6  }
7
8  int MaximoDe3(int a, int b, int c) {
9     int m1 = Maximo(a, b);
10    int m2 = Maximo(m1,c);
11    return m2;
12 }
13
14 int main() {
15    int x, y, z;
16    cout << "Deme 3 números: ";
17    cin >> x >> y >> z;
18    cout << MaximoDe3(x,y,z) << endl;
19 }
```

Supongamos que el usuario se encuentra en la última línea del programa —línea 18— tras haber leído los números *3,7,6* en las variables `x`, `y`, `z`, respectivamente. Podemos decir que la pila se encuentra como indica la figura 9.3.

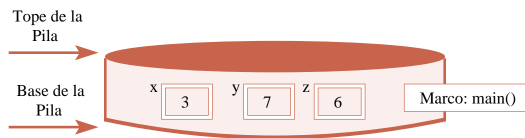


Figura 9.3
Estado de la pila antes de la primera llamada.

Como se puede observar, el sistema ha reservado tres enteros en la pila, asignándoles los nombres de las variables correspondientes. Este segmento de la pila corresponde al *contexto* de la función **`main`**. Cuando accedemos o modificamos el valor de cualquier variable de **`main`** se accede a la posición correspondiente de la pila.

Para resolver la línea *18*, el programa tiene que resolver la llamada a la función `MaximoDe3`. En este caso, el programa deja en la pila el estado actual de la función **`main`** y abre un nuevo marco o contexto para la función `MaximoDe3`. En este caso, decimos que cambiamos de contexto: una referencia a los objetos `x`, `y` o `z` son un error, ya que no se encuentran en el contexto actual. El estado de la pila se puede representar como se muestra en la figura 9.4.

Observe que los valores de las variables `x`, `y`, `z` se han transferido a los objetos correspondientes mediante copia. Además, en la función actual tenemos nuevos objetos locales: los objetos `m1` y `m2`,

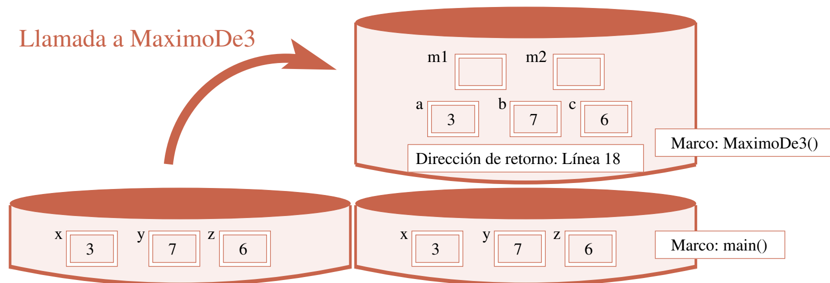


Figura 9.4
Llamada a *MaximoDe3*.

que aún no han recibido ningún valor. Lógicamente, aunque no les incluimos ningún valor, es de esperar que contengan algún número aleatorio que consideramos "basura".

Adicionalmente, junto a estos objetos, hemos añadido una nueva casilla que hemos llamado *"Dirección de retorno"*. Téngase en cuenta que la pila no se utiliza únicamente para las variables locales, sino que el sistema puede usarlo para crear los objetos e informaciones que necesite para gestionar las llamadas a función. En este caso, hemos supuesto que el compilador ha añadido la dirección que corresponde a la llamada que ha generado ese nuevo contexto. Para entenderlo mejor, supongamos que avanzamos a la línea 9 y el compilador realiza la llamada a la función `Maximo` para calcular el primer valor `m1`. El estado podría ser el mostrado en la figura 9.5.

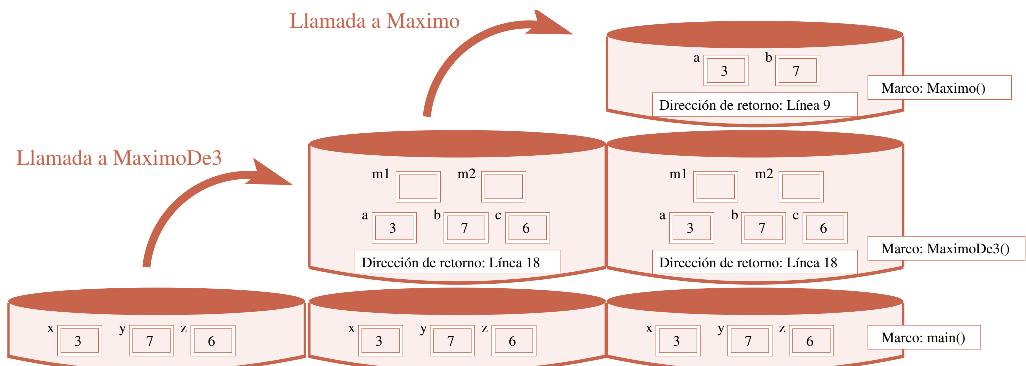


Figura 9.5
Primera llamada a *Maximo*.

Observe que ahora hemos creado un nuevo contexto: el de la función `Maximo`. En este caso necesitamos dos objetos locales `a` y `b` a los que se les copia los dos parámetros de la llamada. Es interesante notar que existen con los mismo nombres que en la función anterior. Sin embargo, si modificamos cualquiera de sus valores, el cambio sólo afectará a las variables locales, y no a las de la llamada que están en otro contexto.

Por otro lado, podemos ver que hemos anotado la *dirección de retorno* (línea 9). Cuando la función termine, este valor le indicará el lugar donde continuar con la ejecución. Si revisa el programa, verá que a continuación se volverá a llamar a la función, pero desde la línea 10. En esa nueva llamada el lugar de retorno será distinto.

En la figura 9.6 puede observar la nueva situación. Al volver de la primera llamada a `Maximo`, hemos asignado el valor `7` a `m1`. Al apilar la nueva llamada —desde la línea 10— copiamos los valores de `m1` y `c` a la función y anotamos la nueva dirección de retorno.

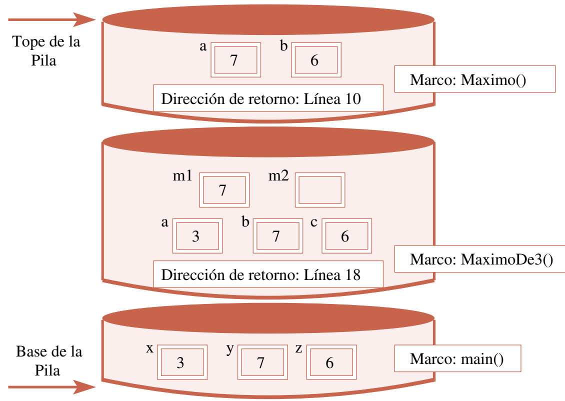


Figura 9.6
Segunda llamada a *Maximo*.

El resto de la ejecución consiste en el desapilamiento de las llamadas. Para eso se devuelve el máximo de `7` y `6`, se almacena en *`m2`* y se devuelve como resultado a la función **`main`**. Se desapila la función *`MaximoDe3`* devolviendo el valor que se escribe como resultado.

9.3.2 Pila y recursividad

La gestión de la recursividad no requiere un tratamiento especial. Una llamada recursiva no es más que una llamada a una función que se gestiona con la pila de la misma forma que cualquier otra llamada. Los detalles que hemos incluido en la sección anterior con distintas funciones son los mismos a tener en cuenta para trazar la ejecución de funciones recursivas.

En las figuras 9.1 y 9.2 se han dibujado repetidamente las funciones *`Factorial`* para que el lector pueda intuitivamente entender en qué consiste su ejecución anidada. En la práctica, lo que antes hemos indicado como *"aparcado"*, o como *"eliminado"*, indicando que una función queda pendiente de concluir y una función ha terminado su ejecución no son más que casos concretos de apilamiento y desapilamiento de llamadas en la pila.

Cuando estamos ejecutando la función *`Factorial`* y realizamos una llamada recursiva, se crea un nuevo contexto o marco en la pila, dejando el anterior contexto apilado para cuando terminemos la ejecución del actual. Por ejemplo, la ejecución de *`Factorial`*`(5)` hace que se llame recursivamente a *`Factorial`*`(4)`, de `3`, `2`, `1` y `0`. Por tanto, cuando estamos resolviendo el caso base `0` tenemos 6 contextos apilados. La profundidad de recursión hace referencia al número de llamadas que se apilan.

Observe cómo es más fácil entender ahora que el código ejecutable de una función sólo aparece una vez en el programa. Aunque en secciones anteriores lo hayamos repetido para mostrar su funcionamiento, lo que realmente aparece de forma múltiple son los contextos que gestionan cada una de las llamadas. Una llamada recursiva implica la creación de un contexto y un salto en el punto de ejecución al comienzo de la función. El fin de una llamada recursiva implica el desapilamiento de un contexto y el salto al punto de la función desde donde se hizo la llamada.

9.3.3 Gestión de la memoria

En la sección anterior hemos mostrado una idea simple de los conceptos básicos y el mecanismo de funcionamiento de la pila. Las variables que hemos mostrado se pueden denominar *automáticas* ya que el compilador las reserva automáticamente en la pila y las libera cuando ya no son necesarias. Como sabemos, en un programa podemos encontrar otros objetos, como son las variables globales,

que no se encuentran alojadas en la pila. Estas variables existen desde el comienzo del programa y se sitúan en una zona de memoria fija, que no se modifica. Más adelante estudiará en mayor detalle cómo se gestiona la memoria, especialmente cuando estudie la gestión de memoria dinámica.

Por otro lado, resulta especialmente interesante que entienda cómo se gestiona la memoria para conocer la facilidad con que el compilador puede usar la pila para resolver la creación y destrucción de variables locales. Recuerde que una variable no existe mientras no lleguemos a ella, por tanto, el compilador puede generar un nuevo contexto sin incluir las variables que aún no se han alcanzado. Por ejemplo, en la llamada mostrada en la figura 9.4 hemos creado un nuevo contexto con los objetos `m1` y `m2`. Sin embargo, el compilador aún no ha alcanzado la declaración de `m2` cuando está resolviendo la primera llamada a `Maximo`.

El compilador puede hacer crecer y decrecer la pila estando dentro de un mismo contexto, sin salir de una función, cuando encuentra nuevos objetos que sitúa automáticamente en la pila, o cuando sale de un bloque donde hay un objeto y lo elimina de la pila. Como puede ver, la estructura de la pila es algo que puede ser aún más dinámico que lo mostrado en la sección anterior. Todo el misterio y la potencia está en algo tan simple como la posibilidad de apilar encima de lo último y desapilar lo último que se ha creado.

Por ejemplo, tal vez se haya preguntado cómo retornar los valores de una función. Cuando hacemos un **`return`** no parece sencillo ver en qué consiste eso de *"copio al punto de llamada"* el resultado de la función. Lo podemos resolver de forma muy simple con la pila. Al terminar una función, no sólo desapilo su contexto sino que también copio el resultado y lo dejo en el tope de la pila. El código que sigue a una llamada a función sabe que el resultado de lo que ha calculado está en una posición de memoria bien determinada: el tope de la pila. Sólo tenemos que usar ese valor y desapilarlo.

Como puede ver, el funcionamiento es extremadamente simple y por eso resulta muy eficiente. Por ejemplo, no se da el caso de que se fraccione la memoria. Cuando necesitamos más espacio lo localizamos justo a continuación del último usado, y cuando liberamos, eliminamos justo el trozo que acabamos de reservar. Como resultado, la memoria usada no tiene huecos con zonas de memoria libres, sino que están ocupando todo el espacio que va desde el fondo de la pila hasta el tope. Además, los datos que probablemente usemos son precisamente los últimos que se han gestionado.

Finalmente, no podemos terminar esta sección sin indicar que el compilador puede implementar el comportamiento del estándar C++ de distintas formas. El comportamiento implementado en un compilador no tiene por qué ser exactamente el presentado aquí, aunque en general responde al esquema que hemos indicado. Puede tener múltiples detalles adicionales que permitan generar un código más eficiente en tiempo y espacio. Por ejemplo, podría optimizar el código generado para ahorrar alguna de las variables que declaramos.

9.4 Diseño de funciones recursivas

El primer paso para obtener la implementación de la solución de un problema por medio de una función recursiva es la *identificación del algoritmo recursivo*, es decir, tenemos que identificar un algoritmo que resuelve el problema original en términos de la solución de subproblemas de la misma naturaleza. Más concretamente: los casos base, casos generales y la solución en términos de ellos.

1. **Los casos base**. Son los casos del problema que se resuelven con un segmento de código sin recursividad. Normalmente corresponden a instancias del problema simples y fáciles de implementar, para los cuales es innecesario expresar la solución en términos de un subproblema, es decir, de forma recursiva.

Es obvio que las llamadas recursivas para la solución de un problema no pueden darse de forma indefinida, sino que debe llegar un caso en el que no es necesario expresar la solución como un problema de menor tamaño, es decir, tienen que existir casos base. Por tanto, una regla fundamental a tener en cuenta con respecto al diseño de una función recursiva es:

> **Regla 9.1 — Casos base.** Siempre debe existir, al menos, un caso base.

Por ejemplo, si consideramos una definición de factorial del tipo $n! = n(n-1)!$, necesitamos añadir el caso $0! = 1$ para indicar una condición de parada en la aplicación de la ecuación recursiva. Por supuesto, esta regla es bastante obvia, pues básicamente estamos afirmando que la recursividad debe parar en algún momento.
Por otro lado, el número y forma de los casos base son hasta cierto punto arbitrarios, ya que no sólo dependen del problema, sino también del programador y su habilidad para identificar un conjunto de casos, en términos de simplicidad y eficiencia.
En general, no es difícil proponer y resolver un conjunto de casos base ya que, por un lado corresponden a instancias simples y, por otro, en caso de problemas muy complejos, siempre es posible proponer varios casos para asegurar que el conjunto es suficientemente amplio como para garantizar el fin de la recursión. Por supuesto, la solución será mejor cuanto más simple y eficiente resulte el conjunto de casos seleccionados.

2. **Los casos generales.** Cuando el tamaño del problema es suficientemente grande o complejo, la solución se expresa de forma recursiva, es decir, es necesario resolver el mismo problema, aunque para una entrada de menor tamaño. Es justo aquí donde se explota la potencia de la recursividad, ya que un problema que inicialmente puede ser complejo, se expresa como la unión de:
   a) La solución de uno o más subproblemas (de igual naturaleza pero menor tamaño).
   b) Posiblemente, un conjunto de pasos adicionales que permiten unir los resultados anteriores para formar la solución final.

   Las soluciones a los subproblemas, junto con estos pasos adicionales, componen la solución al problema general que queremos resolver. Una regla básica, por tanto, es:

> **Regla 9.2 — Avanzar hacia caso base.** Los casos generales siempre deben avanzar hacia un caso base. Es decir, la llamada recursiva se hace a un subproblema más pequeño que se encuentra más cerca de alcanzar un caso base.

Si consideramos los subproblemas como algo resuelto —de hecho generalmente sólo requieren de llamadas recursivas— el problema inicial queda simplificado a resolver sólo el conjunto de pasos adicionales.
Por ejemplo, para el caso del factorial, es necesario calcular el factorial de $n-1$ (la llamada recursiva) y, adicionalmente, realizar una multiplicación que es mucho más simple que el problema inicial. Obviamente, esto nos llevará a la solución si sabemos resolver el subproblema, lo que está garantizado, pues en última instancia se reduce hasta uno de los casos base.
Por tanto, una regla básica a tener en cuenta con respecto al diseño de los casos generales de una función recursiva es:

> **Regla 9.3 — Alcanzar los casos base.** Los casos generales siempre deben desembocar en un caso base. Es decir, la llamada recursiva no sólo llama a casos más pequeños, sino que en última instancia debe alcanzar un caso base.

Por ejemplo, cuando diseñamos la función del cálculo de factorial, el caso general hace referencia al subproblema $(n-1)!$ que está más cerca de alcanzar el caso base $n = 0$. Además,

es fácil ver que si los subproblemas avanzan bajando en uno el tamaño n, seguro que, en última instancia, alcanzan el caso base cuando se realice la llamada desde el valor $n = 1$.

9.4.1 Implementación de funciones recursivas

Es este apartado vamos a presentar algunos consejos y reglas prácticas para implementar funciones recursivas.

En primer lugar, se plantean las posibles consideraciones a la hora de especificar la cabecera de la función recursiva (sintaxis y semántica). Para ello, una regla básica a tener en cuenta es:

> **Regla 9.4 — Cabecera de la función.** La cabecera debe ser válida tanto para llamar al problema original como a cualquiera de los subproblemas.

Supongamos que queremos resolver el siguiente problema: dado un vector de 100 enteros ordenados, devolver la posición de un elemento a buscar, *x*. Inicialmente, este problema puede ser resuelto de la siguiente forma:

```
int Buscar (const vector<int>& v, int x)
{
   for (int i=0; i<100; i++) // Note la imposición del tamaño 100
      if (v[i]==x)
         return i;

   return -1;
}
```

En cambio, si nuestra intención es resolver este mismo problema —de forma recursiva— mediante el algoritmo de búsqueda binaria, la cabecera de la función debe de ser modificada, ya que es imposible expresar la llamada a un subproblema sin añadir algún parámetro adicional. Dado que un subproblema consiste en continuar la búsqueda en una parte del vector, cuando se llama recursivamente, será necesario especificar el subconjunto de elementos del vector donde buscar. Así, una cabecera válida podría ser:

```
int Buscar (const vector<int>& v, int izqda, int drcha, int x)
```

que localiza el elemento *x* en las posiciones comprendidas entre *izqda* y *drcha*, ambas inclusive, del vector *v*.

En segundo lugar, el cuerpo de la función debe implementar los casos base y generales. Ambos están muy relacionados, de manera que tenemos que tener en cuenta cómo programamos los casos generales para poder definir los casos base, y al revés.

Para la programación de los casos generales, una regla básica que puede ser de gran utilidad es:

> **Regla 9.5 — Implementación de casos generales.** Suponer que las llamadas recursivas a subproblemas funcionan.

Es poco recomendable intentar entender el funcionamiento y el progreso de las distintas llamadas recursivas cuando se están programando. Generalmente, el programador debe asumir que las llamadas van a funcionar, es decir, tenemos que asumir que la llamada a la función responde a las especificaciones que hayamos determinado para la función, sin más consideraciones referentes al funcionamiento interno. Es decir, pensar en la función recursiva como usuarios de ellas, y no como programadores.

Para personas que se intentan iniciar en la programación de funciones recursivas, puede ser de utilidad considerar que *la función que se pretende resolver ya está programada para problemas de menor tamaño* (la llamada recursiva funciona). El objetivo de esta suposición es, fundamentalmente, conseguir que el programador no intente imaginar el funcionamiento de sucesivas llamadas recursivas.

Un problema típico en programadores novatos que no consiguen dominar la recursividad radica en que no son capaces de separar los dos papeles que desempeñamos al diseñar la función: programadores y usuarios de la función. Cuando estamos programando la función debemos imaginar que nuestro conjunto de herramientas disponibles —por ejemplo, **sqrt**, **pow**, etc.— incluye una nueva función: la que estamos programando, disponible para cualquier subproblema, sin más consideraciones que el manual de uso (como en el resto de funciones).

Por ejemplo, supongamos que queremos programar la función *Factorial* para un valor *n*. Podemos considerar que disponemos de una función *Resuelto* que implementa el factorial para valores menores que *n*, y además, de la que desconocemos su implementación. Sólo conocemos que su especificación es idéntica a la función *Factorial* que estamos resolviendo, aunque sólo se puede usar para valores menores que *n*. La primera parte de la implementación de la función recursiva sería:

```
int Factorial (int n)
{
   if (n==0)
      return 1;
   else
     //... Caso general
}
```

a falta de construir el caso general. Ahora bien, sabemos que tenemos disponible una función *ya implementada y que funciona*, que se llama *Resuelto*, con la misma especificación —sintaxis `int Resuelto (int n)` y semántica idénticos, es decir, hace exactamente lo mismo pero con otro nombre—. La única restricción es que esta función no se puede usar para el tamaño original *n* de la función *Factorial*, pero sí para cualquier caso más pequeño.

Con este planteamiento, la implementación del caso general de la función *Factorial* es muy simple, pues ya tenemos resuelto el caso de cualquier factorial más pequeño que *n*. Si tenemos que calcular el factorial de *n*, sólo necesitamos realizar una multiplicación de este valor por el factorial de $n-1$, que precisamente ya tenemos en la función *Resuelto*, quedando:

```
int Factorial (int n)
{
   if (n==0)
      return 1;
   else
      return n*Resuelto(n-1);
}
```

Por supuesto, la solución definitiva se obtiene cambiando el nombre *Resuelto* por el de la misma función:

```
int Factorial (int n)
{
   if (n==0)
      return 1;
   else
      return n*Factorial(n-1);
}
```

Es importante destacar que el hecho de haber usado otra función —*Resuelto*— en la programación del factorial tiene como objetivo alejar al lector de pensar que estamos programando una función recursiva. Así, puede pensar que estamos programando una función iterativa, con la gran ventaja de que la mayor dificultad del problema —resolver el subproblema— está ya resuelta en una función auxiliar. Note que incluso la función *Resuelto* no tendría que ser recursiva, es decir, podríamos pensar que implementa el factorial de forma iterativa. Por tanto, pierde todo el sentido considerar el funcionamiento y relación interna de las distintas llamadas recursivas.

No se sorprenda si un programador novato que aún no haya entendido cómo funciona la recursividad modifica el valor de una variable local con la intención de que se use en otra llamada recursiva independiente. Recuerde, si es una variable local, es porque es un detalle interno de la

función y, por tanto, si estoy llamando a un subproblema sólo me interesa saber el efecto de la llamada, no los detalles de cómo se resuelve.

Finalmente, es importante comprobar que, efectivamente, los casos generales desembocan en un caso base. Como hemos indicado anteriormente, para el caso del factorial es fácil ver que las llamadas recursivas acabarán en el caso base.

Un ejemplo: La búsqueda binaria

En primer lugar es interesante recordar *la idea o algoritmo recursivo* que resuelve este problema de búsqueda: la búsqueda de un elemento `x` en un vector ordenado se puede realizar comparando `x` con el elemento central, si es menor, se realiza una búsqueda del elemento en el subvector mitad izquierda, y si es mayor, en la mitad derecha.

Es decir, la solución se puede exponer en términos del mismo problema para un tamaño menor (la búsqueda en una de sus dos mitades).

La cabecera de la función a implementar ya se ha presentado antes (página 195). Note que en el algoritmo recursivo que presentamos en el párrafo anterior ya hacemos referencia a que la búsqueda se realiza en una parte del vector original. De ahí, que la cabecera tenga que permitir referenciar la parte de éste donde buscar. Recordemos que la cabecera es:

```
int Buscar (const vector<int>& v, int izqda, int drcha, int x)
```

que busca el elemento `x` en las posiciones delimitadas por `izqda`, `drcha` del vector ordenado `v`.

Inicialmente, podemos considerar que un caso muy simple del problema —y al que podemos llegar a través de llamadas recursivas— es el caso de que busquemos el elemento en un segmento del vector de tamaño 1, es decir, que $izqda == drcha$. Por tanto, por ahora fijamos éste como el único caso base.

Por otro lado, tenemos que resolver el caso general. Éste consiste en realizar búsquedas en los subvectores correspondientes. Insistiendo de nuevo en la idea de obviar el hecho de que tratamos con una función recursiva, supongamos que las llamadas recursivas funcionan, considerando que tenemos una función `Resuelto` con la misma cabecera:

```
int Resuelto (const vector<int>& v, int izqda, int drcha, int x)
```

Esta función resuelve el problema de la búsqueda por algún método que no nos interesa. Sólo es necesario tener en cuenta que funciona sólo en caso de un subproblema (de menor tamaño). Si no fuera así, la solución podría ser:

```
int Buscar (const vector<int>& v, int izqda, int drcha, int x)
{
   return Resuelto (v,izqda,drcha,x);
}
```

que implicaría que no avanzamos hacia el caso base —*recursión infinita*— cuando cambiemos el nombre `Resuelto` por `Buscar`.

Ahora el problema puede parecer más sencillo, al considerar que ya tenemos una función que nos resuelve el mismo problema. El caso general se puede formular como la comparación con el elemento central, si es menor, la función `Resuelto` nos da la solución sobre la parte izquierda y, si es mayor, sobre la parte derecha (esta función es aplicable, pues resuelve el problema sobre la mitad del vector, que está más cerca del caso base).

Una primera solución al problema de búsqueda es:

```
int Buscar (const vector<int>& v, int izqda, int drcha, int x)
{
   if (izqda==drcha)
      return (v[izqda]==x) ? izqda : -1;
   else {
      int centro= (izqda+drcha)/2;
      if (v[centro]==x)
         return centro;
```

```
        else if (v[centro]>x)
                return Resuelto (v,izqda,centro,x);
             else return Resuelto (v,centro,drcha,x);
    }
}
```

Note cómo hemos incluido un segundo caso base: que el elemento central sea la solución del problema. Al considerar si es menor o mayor nos damos cuenta de que el caso de que sea igual es obvio y resuelve directamente el problema. En este caso no son necesarias llamadas recursivas, y como es de esperar se resuelve de la manera más sencilla.

Por supuesto, la versión que buscamos se obtiene cambiando el nombre `Resuelto` por el de la función. En cualquier caso, es recomendable comprobar que nuestra función sigue las distintas reglas expuestas a lo largo de esta introducción a la recursividad. Concretamente, siempre es recomendable asegurarnos que los casos base son alcanzables (véase regla 9.3, página 194).

Es claro que si el vector tiene muchos elementos, el cortar por la mitad implica un avance hacia el caso base, pero ¿qué ocurre en los casos límite, es decir, cuando estamos cerca de llamar a los casos base? Es bien sabido en ingeniería del software que el estudio de los casos límite son una prueba interesante para intentar localizar errores de programación. Es en estos casos, donde sí se recomienda hacer una traza del funcionamiento de la recursividad, aunque obviamente es mucho más simple, pues sólo consideramos uno o, como mucho, dos pasos anteriores al caso base.

Supongamos que estamos en el paso anterior a la llamada del caso base, es decir, con dos posiciones del vector tal como muestra el caso *(a)* de la figura 9.7.

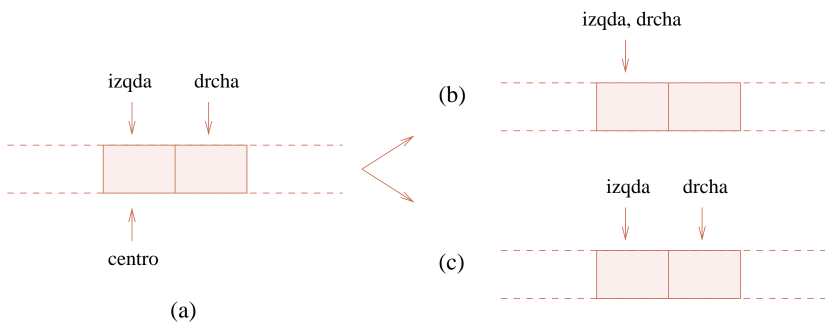


Figura 9.7
Efecto de la llamada recursiva cuando el problema se limita a dos elementos.

Si elemento es menor que el central, se produce la situación del caso *(b)* y si es mayor, la del caso *(c)*. Como podemos observar, la de este último no es válida, pues no se ha avanzado hacia el caso base, es decir, la recursividad fallaría.

Una forma muy simple de mejorar el problema que tenemos sería incluir el caso de dos elementos como un nuevo caso base, pues si comprobamos el comportamiento de esta función cuando hay más de dos, seguro que no se daría ningún problema para alcanzar un caso base. En lugar de eso, notemos que cuando se hace una llamada recursiva, ya se ha comprobado el caso del elemento central, por tanto, parece más interesante solucionar el problema eliminando el elemento central de las llamadas recursivas. De esta manera, garantizamos el avance hacia el caso base, pues al menos eliminamos un elemento (el central). El nuevo algoritmo quedaría:

```
int Buscar (const vector<int>& v, int izqda, int drcha, int x)
{
   if (izqda==drcha)
      return (v[zqda]==x) ? izqda : −1;
   else {
      int centro= (izqda+drcha)/2;
      if (v[centro]==x)
```

```
            return centro;
         else if (v[centro]>x)
                 return Buscar (v,izqda,centro−1,x);
              else return Buscar (v,centro+1,drcha,x);
      }
   }
```

De nuevo, supongamos que estamos en el paso anterior a la llamada del caso base, es decir, con dos posiciones del vector, tal como muestra el caso *(a)* de la figura 9.8.

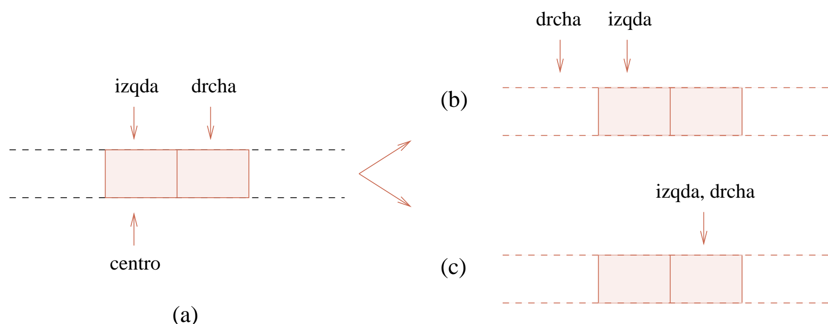


Figura 9.8
Efecto de la llamada recursiva cuando el problema se limita a dos elementos eliminando el elemento central en la llamada.

Ahora se produce un error en caso de que el elemento sea menor, es decir, en el caso *(b)*. Podemos solucionar el problema haciendo que en la primera llamada recursiva no se reste uno a la variable centro, o añadiendo el caso base $izqda > drcha$ a la función. Si añadimos este caso, podemos comprobar que el caso $izqda == drcha$ no es necesario ya que también se reduce, en la siguiente llamada recursiva, al caso $izqda > drcha$.

Por tanto, podemos proponer la siguiente función como definitiva:

```
int Buscar (const vector<int>& v, int izqda, int drcha, int x)
{
   if (izqda>drcha)
      return −1;
   else {
      int centro= (izqda+drcha)/2;
      if (v[centro]==x)
         return centro;
      else if (v[centro]>x)
              return Buscar (v,izqda,centro−1,x);
           else return Buscar (v,centro+1,drcha,x);
   }
}
```

9.4.2 Descomposición en números primos

La descomposición en números primos de n se puede formular recursivamente si tenemos en cuenta que dicha descomposición consiste en encontrar un número primo p que divide a n y descomponer el número n/p: el mismo problema, de menor tamaño. En esta sección proponemos implementar una función que devuelve un vector de enteros con todos los primos de la descomposición.

Podemos considerar que el caso base es que el número n es primo. En este caso la descomposición no necesita llamar recursivamente ya que ese primo es la solución del problema.

El caso general consiste en que el primer número primo p que divide a n es menor que n. En este caso ya tenemos un número de la descomposición —p— aunque nos queda la descomposición de n/p, que podría requerir todavía muchos primos más.

Una primera solución al problema puede incluir una cabecera como la siguiente:

```
vector<int> Descomponer (int n);
```

que es válida tanto para resolver el problema original como cualquier subproblema. La implementación consiste en localizar el primer primo que divide a *n* para poder distinguir si estamos en el caso base o el general. Tenga en cuenta que para buscar el primer primo que divide a *n* podemos buscar el primer entero que lo divide. Por tanto, la primera parte se puede implementar con el siguiente código:

```
int p= 2;
while (n%p!=0)
   p++;
// Aquí seguro que p divide a n y es primo
```

donde el número *p* será primo. Por supuesto, en el código anterior también comprobamos números que no son primos[2], pero seguro que no serán divisores de *n*. Si *p* fuera divisor sin ser primo, cualquiera de los primos que divide a *p* hubiera parado el bucle antes.

Una vez localizado *p* el código es bastante simple. La implementación final puede ser la siguiente:

```
vector<int> Descomponer (int n)
{
   int p= 2; // Será primo
   while (n%p!=0)
      p++;

   vector<int> resultado;
   if (p<n)
      resultado= Descomponer(n/p);
   resultado.push_back(p);
   return resultado;
}
```

Observe que cada una de las llamadas a la función requiere crear un vector `resultado` que se devuelve con **`return`**. Dado que este tipo de objetos puede ser costoso de copiar[3], podemos pensar en modificar la solución para no tener que devolverlo. Para ello podemos implementarlo de la siguiente forma:

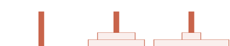
```
void Descomponer (int n, vector<int>& vec)
{
   int p= 2;
   while (n%p!=0)
      p++;

   vec.push_back(p);
   if (p<n)
      Descomponer(n/p,vec);
}
```

Ejercicio 9.4 En la solución anterior buscamos un divisor siempre desde el número 2. ¿Cómo podríamos evitarlo? Modifique la función para incluir la mejora.

En el código anterior hemos cambiado el sentido de la función. En este caso la función no consiste en hacer que el vector `vec` tenga la descomposición de *n*, sino que el efecto de la llamada provoca la adición de la descomposición de *n* al final de `vec`. Incluso sería adecuado modificar su nombre; por ejemplo, llamarla `AnadirFactores`, para que refleje la idea de que `vec` aumenta

[2]Este código es mejorable. Por ejemplo, comprobando el 2 y luego sólo los impares, aunque ahora mismo no es el problema que nos interesa.

[3]Esta función podría ser un buen ejemplo para analizar con el nuevo estándar C++11, donde se han añadido nuevas características que harían que el resultado de esta función fuera tan eficiente como si evitamos devolver el vector con **`return`**.



de tamaño sin modificar los elementos ya almacenados. Tener esta idea clara es fundamental para entender por qué funciona la implementación anterior.

Por ejemplo, supongamos que la idea que tenemos es crear una función que no añada los elementos a `vec`, sino que modifica `vec` eliminando los elementos que tuviera antes. Podemos modificar la función de la siguiente manera:

```
void Descomponer (int n, vector<int>& vec)
{
   vec.clear(); // Limpiamos valores anteriores de vec

   int p= 2;
   while (n%p!=0)
      p++;

   vec.push_back(p);
   if (p<n)
      Descomponer(n/p,vec);
}
```

Sin embargo, esta función no es válida. Piense que si la función realiza esa operación, la llamada al subproblema sobre el número `n/p` realiza exactamente la misma operación. Concretamente, al llamar recursivamente el valor de `vec` se limpiará para poder albergar los números que forman la descomposición de `n/p`. Esta limpieza elimina el valor `p` que acabamos de añadir.

Para resolverlo podemos modificar ligeramente el código anterior. Si la función realiza esa operación, un algoritmo válido consiste en actualizar el valor de `vec` con la descomposición de `n/p` y después añadir el valor de `p` al final del vector. El código podría ser:

```
void Descomponer (int n, vector<int>& vec)
{
   int p= 2;
   while (n%p!=0)
      p++;

   vec.clear(); // Limpiamos valores anteriores de vec
   if (p<n)
      Descomponer(n/p,vec);
   vec.push_back(p);
}
```

donde también hemos movido la posición de la primera línea para hacerlo más legible.

Ejercicio 9.5 Considere un programa que llama a las dos funciones —`AnadirFactores` y la versión final de la función `Descomponer`— para escribir los resultados de cada una de ellas para un mismo número `n`, ¿qué diferencias habrá?

9.5 Recursivo vs iterativo

En muchos casos, para resolver el problema la recursividad requiere más recursos (tiempo y memoria) que una versión iterativa por lo que, independientemente de la naturaleza del problema, la recursividad tiene su gran aplicación para problemas complejos, cuya solución recursiva es más fácil de obtener, más estructurada y sencilla de mantener. Así por ejemplo, los problemas de cálculo de factorial o de búsqueda binaria son fáciles de programar iterativamente, por lo que generalmente no se utilizan las versiones recursivas:

- En el caso del factorial, no necesitamos más que un bucle para calcular el producto, mientras que en la versión recursiva el número de llamadas que se anidarían —*profundidad de recursión*— es igual al valor de entrada. De manera que usar la versión recursiva es más un error que algo poco recomendable.
- Para la búsqueda binaria, el máximo número de llamadas anidadas —profundidad de recursión— es del orden del logaritmo del tamaño del vector, por lo que este factor es

menos importante. A pesar de ello, y considerando que la versión iterativa es muy simple, también para este caso podemos rechazar el uso de la recursividad. Tenga en cuenta que la versión iterativa no incluye el coste de las llamadas y la memoria adicional para gestionarlas.

Incluso para casos más complejos, pero correspondientes a funciones que se llaman muchas veces en un programa, puede ser interesante eliminar la recursividad, ya sea de una manera sencilla —como es el caso de la recursión de cola, como veremos más adelante— o de forma más compleja mediante la introducción de estructuras adicionales como *pilas* definidas por el programador.

Además, cuando se resuelven problemas de manera recursiva hay que tener especial cuidado, ya que el que la función tenga varias llamadas recursivas puede hacer que el número de subproblemas a resolver crezca de forma exponencial, haciendo incluso que el resultado obtenido sea mucho más ineficiente. Por tanto, podemos plantear una última regla en la programación recursiva:

> **Regla 9.6 — No repetir subproblemas.** No resolver el mismo subproblema varias veces en distintas llamadas recursivas.

Para ilustrar esta idea, véase más adelante el ejemplo de la sucesión de Fibonacci.

A pesar de estas consideraciones, existen multitud de problemas que se ven radicalmente simplificados mediante el uso de la recursividad, y que cuya diferencia en recursos necesarios con respecto a la solución iterativa puede ser obviada a efectos prácticos.

Recuerde los comentarios que hemos incluido en la sección 9.3.3 (página 192), donde indicamos que la gestión de la pila puede realizarse de una forma muy eficiente. En la práctica, la ejecución de funciones recursivas puede resultar muy rápida, ya que el compilador implementa de una forma muy eficiente esa gestión. Si desea evaluar el uso o no de la recursividad tenga en cuenta un estudio más formal de la eficiencia del algoritmo que está usando, y no tanto si un apilamiento en la pila puede resultar más o menos eficiente.

9.5.1 Recursión de cola

La recursión de cola hace referencia a la llamada recursiva que se realiza al final de la función. Por ejemplo, la búsqueda binaria tiene recursividad de cola ya que cuando hemos determinado el subvector donde se encuentra la solución, lo único que resta por hacer es volver a llamar a la función sobre ese nuevo subproblema.

El interés de la recursividad de cola radica en la facilidad con que puede eliminarse. Realizando una descripción un poco burda, la eliminación se podría realizar simplemente haciendo que los datos se refieran al subproblema y transfiriendo la ejecución al comienzo de la función. Lógicamente, en la práctica se podrá realizar con algún tipo de algoritmo iterativo que vuelve a repetir el código para el subproblema correspondiente.

En la mayoría de los casos de recursión de cola es muy fácil escribir un algoritmo iterativo para resolver el problema. Esta versión suele ser más eficiente, ya que nos ahorramos el espacio del apilamiento y el tiempo para realizar —y devolver— la llamadas. Por ejemplo, funciones para calcular el factorial, la búsqueda binaria, máximo común divisor, etc. siempre se implementarán de forma iterativa, aunque en este libro las usemos para practicar proponiendo versiones recursivas.

La recursividad de cola cobra aún más interés cuando el lenguaje no permite las llamadas recursivas, o cuando permitiéndolas, el espacio que se reserva para la pila es muy limitado. Los compiladores pueden incluir procedimientos automáticos de optimización para analizar las funciones recursivas con recursión de cola y eliminarla de forma automática.

9.6 Ejemplos de funciones recursivas

El objetivo de esta sección es fijar los conceptos desarrollados mostrando algunos ejemplos prácticos. Por supuesto, el lector puede encontrar mucha más información en la literatura. Véase por ejemplo otros algoritmos en Brassard[3], Peña Marí[21], Sedgewick[28], o ejemplos más específicos sobre estructuras de datos más complejas en Weiss[43].

9.6.1 Sucesión de Fibonacci

La sucesión de Fibonacci viene dada por:

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \text{ o } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases} \tag{9.2}$$

Como vemos, es muy similar al ejemplo del cálculo del factorial. Obtener una solución no debería ser ningún problema para el lector. Por ejemplo:

```
int Fibonacci (int n)
{
   if (n<2)
      return 1;
   else
      return Fibonacci(n-1)+Fibonacci(n-2);
}
```

El interés en esta función no es tanto la dificultad de implementarla, sino la discusión sobre la conveniencia de una implementación recursiva. Como indicábamos anteriormente, es importante tener cuidado de no resolver varias veces el mismo subproblema. En este caso podemos observar que para cada n hay que calcular *F(n-1)* y en segundo lugar *F(n-2)*, que ya está incluido en el primero. Observe que si $n-1>1$:

$$F(n-1) = F(n-2) + F(n-3) \tag{9.3}$$

Por tanto, el resultado será mucho más ineficiente al estar duplicando trabajo.

9.6.2 Torres de Hanoi

El problema de las torres de Hanoi consiste en que inicialmente tenemos un conjunto de tres torres. La primera de ellas tiene apiladas n fichas de mayor a menor tamaño. El problema es pasar las n fichas de la primera a la tercera torre, teniendo en cuenta que:

1. En cada paso sólo se puede mover una ficha.
2. Una ficha de un tamaño no puede apilarse sobre otra de menor tamaño.

La situación inicial, en el caso de tener 4 fichas, se muestra en la figura 9.9. El problema consiste en pasar las fichas de la torre 1 a la torre 3, teniendo en cuenta las dos restricciones anteriores.

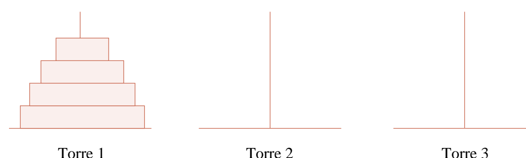


Figura 9.9
Situación inicial de las torres de Hanoi con 4 fichas.

Inicialmente, el problema no parece trivial, pues no es sencillo proponer un algoritmo iterativo que lo resuelva. Ahora bien, si descomponemos el problema para que su solución se establezca en términos de la solución de un subproblema, la situación es mucho más simple.

La *"idea recursiva"* que debemos considerar es que el problema de pasar 4 fichas, de la torre 1 a la torre 3, se puede descomponer en pasar 3 fichas de la torre 1 a la torre 2, pasar la ficha más grande de la 1 a la 3, y volver a pasar las 3 fichas de la torre 2 a la torre 3. Por lo tanto, para resolver el problema de tamaño 4 hemos de resolver dos subproblemas de tamaño 3.

En primer lugar tenemos que diseñar la cabecera de la función. Lógicamente, no basta con pasar como parámetro el número de fichas, sino que además tenemos que especificar la torre inicial y final, pues estos valores son diferentes en las llamadas a subproblemas. Por tanto, la función tendrá la siguiente cabecera:

```
void Hanoi (int m, int inic, int final)
```

que como vemos, refleja los parámetros que queremos introducir y es válida para expresar una llamada a un subproblema: sólo tendremos que cambiar el número de fichas a una menos, así como los números de las torres inicial y final.

Es decir, vamos a diseñar una función `Hanoi` que tome como parámetros un entero `m`, que indica el número de fichas a pasar, y dos enteros (`inic`, `final`) que indiquen la torre inicial —donde se encuentran las `m` fichas— y la torre final —donde acabarán—. Como resultado, la función escribirá en la salida estándar una línea por cada movimiento, indicando la torre origen y destino, desde la que parte y a la que llega la ficha, respectivamente.

En segundo lugar, es necesario determinar cuáles serán los casos base y casos generales. El caso base corresponde a un valor del número de fichas a pasar que sea bajo, y por tanto, sea un problema con fácil solución sin necesidad de usar recursividad. Un buen ejemplo es hacer que el caso base corresponda al problema de traspasar una sola ficha de la torre `inic` a la torre `final`. En esta caso, la solución es muy sencilla pues sólo es necesario escribir un mensaje indicando que el movimiento es pasar la ficha de la torre `inic` a la torre `final`.

El caso general corresponderá a tener más de una ficha. En este caso, tenemos que resolver dos subproblemas de tamaño `m−1`. Insistamos de nuevo en la idea de que al programar no debemos pensar en el funcionamiento interno de las llamadas recursivas sino que debemos asumir que las llamadas funcionan. Sabemos que si llamamos a la misma función con un tamaño de torre menor, se escribirán correctamente en la salida estándar todos los pasos necesarios.

Recordemos que el caso general consiste en escribir todos los pasos para:

1. Llevar `m−1` fichas de la torre `inic` a la torre intermedia,
2. pasar una ficha de la torre `ininc` a la torre `final` y
3. llevar `m−1` fichas de la torre intermedia a la torre `final`.

La primera parte la resuelve una llamada recursiva, la segunda es sólo escribir una línea con ese movimiento, y la tercera también la resuelve una llamada recursiva. La solución completa, por lo tanto, podría ser la siguiente:

```
void Hanoi (int m, int inic, int final)
{
   if (m==1)
      cout << "Ficha de " << inic << " a " << final << endl;
   else {
      Hanoi (m−1, inic, 6−inic−final);
      cout << "Ficha de " << inic << " a " << final << endl;
      Hanoi (m−1, 6−inic−final, final);
   }
}
```

donde la torre auxiliar (la que no es ni `inic` ni `final`) se obtiene como `6−inic−final`. Efectivamente, el caso general expresa la *"idea recursiva"* que antes se indicaba.

Las distintas reglas que comentábamos a lo largo de este tema son válidas en este ejemplo. El caso general avanza hacia casos base y, además, éstos son alcanzables. Si trazamos el comportamiento cuando el número de fichas es 2, el resultado de las distintas operaciones es correcto. Por lo tanto, esta función resuelve correctamente nuestro problema.

Por último, es interesante ver que el caso base podría haberse determinado como $m = 0$, obteniéndose una versión más simple de la función:

```
void Hanoi (int m, int inic, int final)
{
   if (m>0) {
      Hanoi (m−1, inic, 6−inic−final);
      cout << "Ficha de " << inic << " a " << final << endl;
      Hanoi (m−1, 6−inic−final, final);
   }
}
```

El lector puede comprobar que la función es válida y que obtiene la salida correcta al llamar al caso de $m = 1$.

9.6.3 Ordenación por selección

Algunos algoritmos de ordenación pueden ser fácilmente considerados como algoritmos recursivos, ya que en muchos casos se pueden expresar en términos de la ordenación de algún subvector del problema original. Como caso ilustrativo, consideremos la ordenación por selección que seguro que conoce bien, aunque en la práctica se implementará en base a un algoritmo iterativo, que será más rápido y menos costoso en recursos que el presentado aquí[4].

Como muestra la figura 9.10, la *"idea recursiva"* que subyace en este algoritmo es que la ordenación de un vector de tamaño *n* es la *selección* del elemento más pequeño (*m*); el intercambio de éste con el elemento de la primera posición (*A*); y finalmente la *ordenación del subvector* de tamaño *n-1* comprendido entre los elementos 2 y *n*, ambos incluidos.

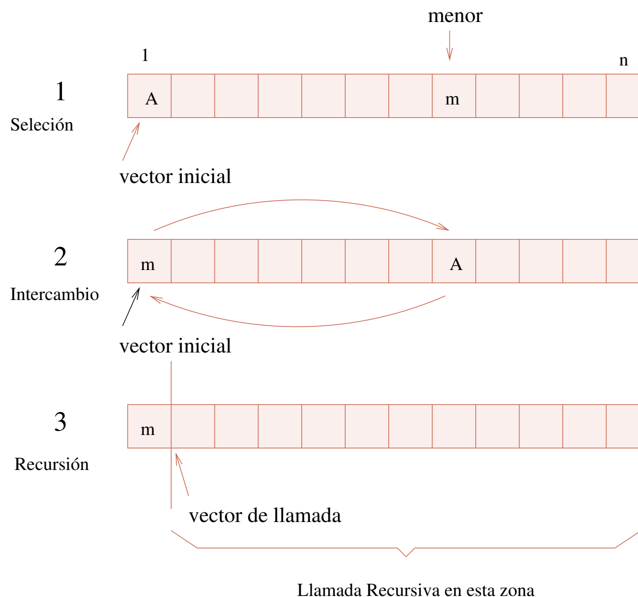


Figura 9.10
Algoritmo de ordenación por selección, recursivo.

La cabecera de la función debe incluir la posibilidad de llamar a uno de los subproblemas. Éstos consisten en la ordenación de los elementos desde una posición en adelante. Por tanto, podemos indicarlo pasando, junto al vector, la posición desde la que ordenar.

[4]Note que la profundidad de recursión para este algoritmo llega a ser del orden del número de elementos del vector.

En segundo lugar, es necesario determinar cuáles serán los casos base y casos generales. El caso base corresponde a un valor del número de elementos pequeño de forma que sea simple de solucionar, sin necesidad de recursividad. En este problema podemos determinar el caso de un elemento como el más simple. La solución para este tamaño es realmente simple: no hacer nada, puesto que un vector de 1 elemento está ya ordenado. El caso general corresponderá a tener 2 o más elementos. En este caso tendremos que llevar a cabo los tres pasos anteriormente expuestos:

1. Selección del menor valor. Mediante un bucle que recorre el vector, seleccionamos el más pequeño de sus elementos (notemos la posición `minimo`).
2. Intercambio del elemento. El elemento en posición `minimo` se intercambia con el de la primera posición.
3. Recursión. De forma recursiva se resuelve el problema, de igual naturaleza pero de menor tamaño ($n-1$), sobre el vector que comprende los elementos desde la posición siguiente a la inicial (ver figura 9.10).

En lenguaje C++ se traduce en lo siguiente:

```
void SeleccionRecursivo (vector<int>& vec, int desde)
{
   if (desde<vec.size()-1) { // Al menos dos elementos
      int minimo= desde;
      for (int i=desde+1; i<vec.size(); i++)
         if (vec[i]<vec[minimo])
            minimo= i;
      int aux= vec[desde];
      vec[desde]= vec[minimo];
      vec[minimo]= aux;
      SeleccionRecursivo(vec,desde+1);
   }
}
```

En este ejemplo es interesante que observe el parámetro que pasa por referencia. Es de esperar este paso, pues el vector original debe modificarse. Note que al pasar siempre por referencia todas las modificaciones que se están haciendo en cualquiera de las llamadas recursivas están actuando sobre el vector de la llamada original.

9.6.4 Ordenación por mezcla

En esta sección mostramos un ejemplo de algoritmo de ordenación en el que puede estar más justificado el uso de la recursividad, ya que su solución es bastante simple y clara con respecto a una posible solución iterativa.

El algoritmo lo podemos considerar una solución a la aplicación directa de la técnica de *divide y vencerás* (ver Brassard[3]) a la ordenación de vectores. Básicamente, la idea consiste en dividir el vector en dos partes, realizar la ordenación de esas dos partes, y finalmente componerlas en una solución global. Es obvia la posible aplicación de la recursividad, ya que cada una de esas dos partes se convierte en el mismo problema, aunque de menor tamaño.

La composición de la solución global se realiza mediante mezcla. Si tenemos un vector en el que cada una de las dos mitades que lo componen están ordenadas, y pretendemos unirlas en una sola secuencia ordenada, la solución consistirá en ir recorriendo ambos subvectores desde los dos elementos más pequeños —desde el primer elemento al último— e ir añadiendo el más pequeño de los dos a la solución final (véase sección 6.2.5 en página 131).

En la figura 9.11 se presentan distintos momentos del estado de un vector que se está ordenando con este algoritmo. En la primera gráfica podemos ver los elementos desordenados. Conforme avanza el algoritmo, se puede observar cómo quedan ordenados distintos subvectores. Por ejemplo, en la cuarta gráfica ya se ha conseguido ordenar la primera mitad y se está ordenando la segunda.

La función debe ser válida para la llamada sobre todo el vector y para las llamada recursivas que resuelven la ordenación de un subvector. Para implementarlo no tenemos más que añadir dos índices que indican la parte a ordenar. Por tanto, la cabecera podría ser:

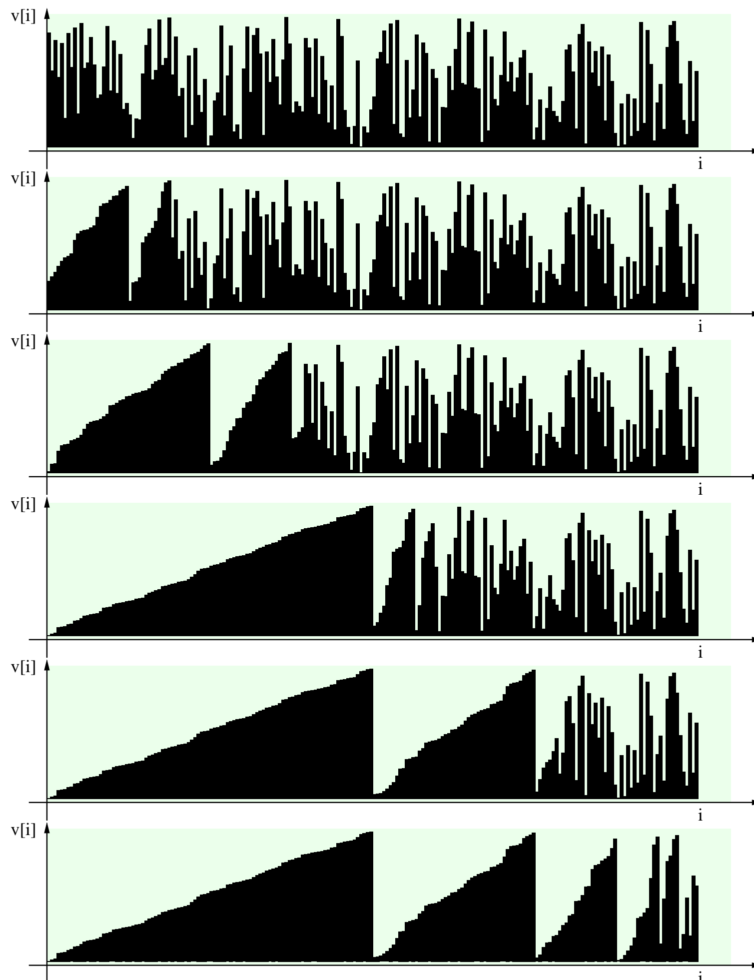


Figura 9.11
Evolución del algoritmo de ordenación por mezcla.

```
void OrdenMezcla (vector<int>& vec, int izqda, int drcha)
```

Por otro lado, es necesario determinar los casos base y general. Al igual que en el apartado anterior, consideramos el caso de ordenar un sólo elemento como caso base, donde no hay que realizar ninguna operación. El caso general tendrá que implementar:

1. Ordenación de las dos mitades.
2. Mezcla de las dos mitades.

La primera de estas dos etapas corresponde a resolver el mismo problema pero de menor tamaño. La segunda no es más que una operación —de menor complejidad que la ordenación— en la que únicamente tenemos que ir reescribiendo los elementos resultantes de forma ordenada. Sin más discusión, la solución completa podría ser:

```
void OrdenMezcla (vector<int>& vec, int izqda, int drcha)
{
   if (drcha-izqda>0) { // Al menos dos elementos
      int centro= (izqda+drcha)/2;
      OrdenMezcla (vec,izqda,centro);
```

```
        OrdenMezcla (vec,centro+1,drcha);

        // Declaramos un vector auxiliar para mezclar
        vector<int> auxiliar(drcha-izqda+1);

        // Volcamos primera mitad al principio de auxiliar
        int paux= 0;
        for (int i=izqda; i<=centro; i++) {
           auxiliar[paux]=vec[i];
           paux++;
        }

        // Volcamos segunda mitad al final de auxiliar, pero hacia atrás
        paux=auxiliar.size()-1;
        for (int j=centro+1; j<=drcha; j++)  {
           auxiliar[paux]= vec[j];
           paux--;
        }

        // Devolvemos elementos de auxiliar a vec, pero mezclando ordenadamente
        int i= 0, j= auxiliar.size()-1;
        for (int k=izqda; k<=drcha; k++)
           if (auxiliar[i]<auxiliar[j]) {
              vec[k]= auxiliar[i];
              i++;
           }
           else {
              vec[k]= auxiliar[j];
              j--;
           }
     }
  }
```

De nuevo tenemos un ejemplo con un objeto —`vec`— que pasa por referencia en todas las llamadas. Podemos decir que todas las llamadas comparten exactamente el mismo objeto. Podría incluso imaginar que la variable `vec` es global, no pasa como parámetro, sino que se accede desde cualquiera de las funciones.

Ejercicio 9.6 Los elementos de las dos mitades se disponen en el vector auxiliar de manera ascendente, en el primer caso, y descendente en el segundo. ¿Por qué?

Ejercicio 9.7 Para realizar la mezcla se crea un vector auxiliar cada vez que se llama a la función, tanto en la llamada principal como para cualquier llamada recursiva. Dado el número tan alto de llamadas, puede resultar aconsejable usar la misma variable auxiliar repetidamente en todas las mezclas. Proponga una modificación de la función, incluyendo la cabecera, para usar un único vector auxiliar (que tendrá un tamaño igual al del vector original) en todas las llamadas.

Para más detalles sobre este algoritmo se puede consultar, por ejemplo, Brassard[3] y Sedgewick[28].

9.7 Problemas

Problema 9.1 Implemente dos funciones (iterativa y recursiva) para el cálculo del máximo común divisor, usando para ello el algoritmo de Euclides.

Problema 9.2 Implemente una función recursiva para calcular el máximo de un vector. Considere que este valor puede ser obtenido a partir del máximo de dos valores:

1. El máximo de la primera mitad.
2. El máximo de la segunda mitad.

Problema 9.3 Para aproximar el valor de la raíz de un número real a, podemos usar la siguiente ecuación:

$$x_i = \begin{cases} 1 & \text{si } i = 0 \\ \frac{1}{2}\left(x_{i-1} + \frac{a}{x_{i-1}}\right) & \text{si } i > 0 \end{cases} \quad (9.4)$$

que define una sucesión de valores que tiende a $\sqrt{a}$ cuando i tiende a infinito. Escriba dos funciones (iterativa y recursiva) para calcular el valor de x_n, dados los valores a y n.

Problema 9.4 Desarrolle un programa para pasar de base 10 a una nueva base. El programa recibe dos parámetros en la línea de órdenes,

1. Un número entero que indica la nueva base (por ejemplo, hasta base 25).
2. El número entero a transformar.

Considere dos programas distintos:

1. Desarrolle el programa para que escriba en la salida estándar el resultado de la transformación, implementando para ello una función recursiva que imprime en la salida estándar el número (considere que los caracteres a usar para las bases superiores a 10 son las letras 'A','B', etc.)
2. Modifique la función recursiva para que obtenga un `string` con el resultado, en lugar de imprimirlo en la salida estándar.

Problema 9.5 Supongamos que `a,b` son dos números enteros positivos. ¿Qué devuelve la siguiente función?

```
int Funcion(int a, int b)
{
   return (a>=b)? funcion(a-b,b)+1 : 0;
}
```

Implemente una función similar para calcular un entero que aproxime el logaritmo en base 2 de un número.

Problema 9.6 Denominaremos *fractal* de tamaño n al resultado de dibujar repetidamente un cuadrado de la siguiente forma:

1. Dibujar en el centro un cuadrado.
2. Dibujar en cada esquina del cuadrado un nuevo fractal de tamaño *n-1*.

En la figura 9.12 se presenta un fractal de tamaño 3.

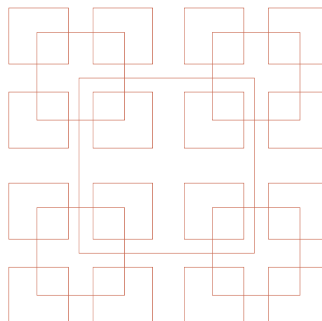


Figura 9.12
Ejemplo de fractal.

Supongamos que disponemos de una función `Rectangulo` que dibuja un rectángulo (con 4 valores `double` de entrada indicando las esquinas superior izquierda e inferior derecha). Implemente una función recursiva que dibuje un fractal de tamaño `n`.

Problema 9.7 Se desea estudiar detalladamente el comportamiento de la función recursiva *Hanoi*. Para ello, necesitamos obtener la traza de todas las llamadas recursivas que se efectúan para un determinado problema de tamaño *m*. Por ejemplo, con dos fichas podríamos obtener el siguiente listado

Consola

```
Entro Hanoi (prof: 1): 2,1,3
  Entro Hanoi (prof: 2): 1,1,2
    Entro Hanoi (prof: 3): 0,1,3
    Salgo Hanoi (prof: 3):0,1,3
  Ficha de 1 a 2
    Entro Hanoi (prof: 3): 0,3,2
    Salgo Hanoi (prof: 3):0,3,2
  Salgo Hanoi (prof: 2):1,1,2
Ficha de 1 a 3
  Entro Hanoi (prof: 2): 1,2,3
    Entro Hanoi (prof: 3): 0,2,1
    Salgo Hanoi (prof: 3):0,2,1
  Ficha de 2 a 3
    Entro Hanoi (prof: 3): 0,1,3
    Salgo Hanoi (prof: 3):0,1,3
  Salgo Hanoi (prof: 2):1,2,3
Salgo Hanoi (prof: 1):2,1,3
```

donde se puede observar que se especifica el momento de entrada y salida de la función para cada llamada, así como los parámetros de la llamada. Además, para que sea más sencillo de estudiar, se sangran los mensajes según el nivel de profundidad de la recursión.

Modifique la función *Hanoi* y desarrolle un programa que lea de la línea de órdenes el número de fichas *m* e imprima todos los mensajes de traza según el formato que se ha especificado.

Problema 9.8 Se desea dibujar una regla en la que se deberán trazar las marcas correspondientes a sus divisiones. En un intervalo, se dibujarán marcas para dividirlo en 10 partes iguales y, si es posible (las marcas son de una altura mayor que uno y están separadas más de 5 unidades), se volverá a dibujar una nueva regla en cada una de esas divisiones (las nuevas marcas se reducirán a la mitad en altura). Supongamos que disponemos de una función:

```
void Marcar(double x, double h);
```

que dibuja una marca de altura *h* en la posición *x*. Implemente una función para dibujar una regla en un intervalo dado.

10 Introducción a flujos de E/S

Introducción 211
Flujos de E/S 212
Operaciones básicas con flujos 216
Flujos asociados a ficheros 225
Flujos y C++98 232
Problemas 233

10.1 Introducción

Una vez que se dispone de las estructuras básicas de control del flujo de ejecución, conocimientos sobre modularización y se saben manejar los tipos de datos compuestos básicos, ya podemos resolver problemas más complejos. La cantidad de datos que se pueden llegar a manejar en este tipo de aplicaciones puede resultar muy grande. Normalmente, estos datos estarán almacenados como ficheros en dispositivos de almacenamiento externo. Para que nuestras aplicaciones puedan procesarlos, es necesario estudiar las herramientas que C++ nos ofrece para poder leer o escribir archivos.

En este nivel de programación no incluiremos todos los detalles y posibilidades del sistema de E/S que ofrece el lenguaje. El objetivo a este nivel es doble:

1. Entender mejor cómo funcionan las operaciones de E/S que hemos usado en los capítulos anteriores —con `cin` y `cout`— ampliando sus posibilidades.
2. Tener la posibilidad de manejar varias fuentes de entrada y salida de datos. En concreto, poder leer/escribir datos desde/hacia uno o varios archivos.

Para ello, estudiaremos los conceptos más básicos para manejar los flujos de E/S, así como la creación de flujos asociados a archivos que almacenan texto.

Se deja para más adelante un estudio más amplio sobre la E/S en C++. En concreto, será especialmente interesante la ampliación de los contenidos estudiando los distintos tipos de datos que se ofrecen, así como otras posibilidades como son la E/S de archivos en formato binario o el acceso aleatorio a los contenidos de un archivo.

En los temas anteriores se ha presentado C++ según el estándar del 98, dejando para después el estudio de algunas características interesantes incorporadas en C++11. En este capítulo lo hacemos de otra forma: vamos a incorporar directamente algunos detalles desde el principio, ya que en este caso no queremos ampliar conocimientos, sino cambiar la forma en que se realizan algunas operaciones. Al final del tema revisamos con más detalle estas diferencias incorporando una sección que, en lugar de añadir las nuevas características de C++11, indique los detalles que no hemos incluido del anterior estándar.

10.2 Flujos de E/S

En C++, el problema de la E/S se resuelve con el uso de *flujos*. Un flujo (*stream*) es una secuencia de *bytes*[1] que nos permite comunicar nuestro programa de forma que:

- Hablamos de *flujo de entrada* cuando los caracteres fluyen, secuencialmente, desde un dispositivo o fichero externo hacia nuestro programa.
- Hablamos de *flujo de salida* cuando los caracteres fluyen, secuencialmente, desde nuestro programa a un dispositivo o fichero externo.

Este concepto de flujo es una abstracción del problema de E/S que nos permite resolverlo sin que nuestro programa deba tener en cuenta las particularidades asociadas a la fuente o destino de nuestros datos.

Por ejemplo, cuando leemos datos desde un teclado, podemos interpretar que se establece un canal —entre el teclado y nuestro programa— de forma que se recibe una secuencia de caracteres. Por otro lado, si leemos datos desde un fichero en disco, aunque los detalles del dispositivo son muy distintos, también podemos considerar que se establece un canal o flujo de caracteres entre el fichero y nuestro programa.

El lenguaje C++ define tipos para manejar estos flujos. Una operación de entrada de datos tendrá asociada la extracción de caracteres desde un flujo; una operación de salida tendrá asociada la salida de caracteres hacia un flujo. En la figura 10.1 se muestra un esquema de un programa que tiene un flujo de entrada y uno de salida. El programa es independiente de los dispositivos externos, ya que su relación con ellos se establece a través de los flujos.

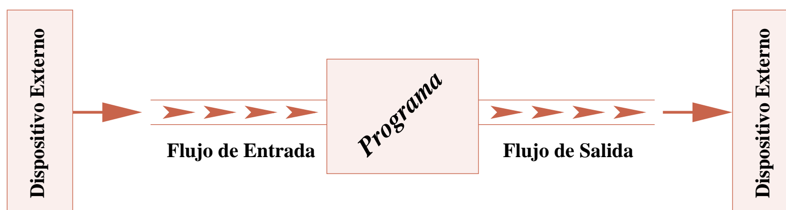


Figura 10.1
Flujos de E/S.

El lector ya conoce parte de las operaciones con flujos, ya que ha usado el flujo de entrada `cin` y el de salida `cout`. Estos dos flujos son objetos definidos en C++ para que cualquier programa disponga de las operaciones básicas con la entrada y salida estándar. Como consecuencia, dado que el uso de flujos es independiente del dispositivo usado, todo lo que hemos aprendido y aplicado en la E/S usando la consola también nos servirá para los ficheros.

10.2.1 Flujos y buffers

El esquema de la figura 10.1 muestra la idea general del concepto de un flujo. Sin embargo, es necesario mostrar un esquema más detallado para entender las operaciones y efectos de las distintas operaciones sobre flujos.

En la figura 10.2 se presenta el mismo gráfico, al que hemos incluidos dos elementos adicionales: dos *buffers*. La idea básica es que el dispositivo de entrada o salida no se comunica directamente con nuestro programa, sino que se utiliza una zona de memoria intermedia para almacenamiento temporal de los datos.

[1] Realmente, no es una secuencia de *bytes*, sino de caracteres. Sin embargo, lo indicamos así para enfatizar que la secuencia de datos no tiene por qué estar asociada a ningún tipo de dato concreto, aunque se maneje con el tipo `char`.

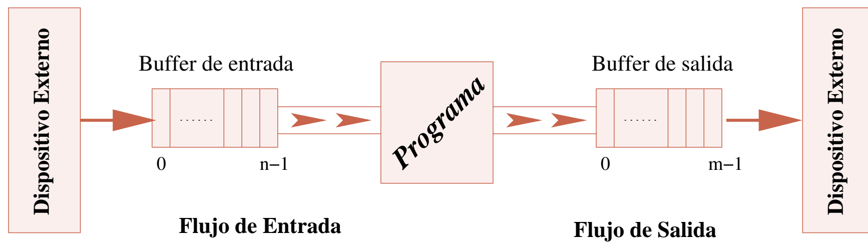


Figura 10.2
Flujos de E/S.

Cuando nuestro programa necesita leer un carácter, lo lee del buffer. Si el buffer está vacío, se deberá esperar a que se realice una operación de entrada desde el dispositivo hasta el buffer. Por otro lado, si el buffer está lleno, el dispositivo de entrada deberá esperar a que el programa lea caracteres del buffer para liberar espacio. En el caso de la salida, la idea es similar.

Como puede observar, un buffer no es más que una zona de memoria donde se van encolando los caracteres de forma temporal. Las principales ventajas de este nuevo esquema son:

1. Mejora la eficiencia en las operaciones de E/S. La entrada o salida de datos del programa sólo necesita trasladar una serie de caracteres entre estas zonas de memoria y nuestro programa. Podemos minimizar las operaciones de E/S con los dispositivos externos, especialmente si son más lentos. Por ejemplo, la comunicación con el dispositivo de salida se puede intentar retrasar hasta que el buffer esté lleno.
2. Dividimos el problema del diseño de los flujos en dos partes, una que se ocupa de realizar las operaciones de E/S con el exterior, y otra que se encarga de comunicar esos buffers con nuestro programa.

Estos comentarios sobre el esquema de funcionamiento son una visión simplificada de la solución que ofrece el lenguaje. Esta visión es suficiente para que entienda el comportamiento y pueda utilizar la mayoría de las operaciones sin ningún problema. Por ejemplo, es posible que haya intentado encontrar el punto de error de un programa añadiendo sentencias que indiquen el punto alcanzado en el programa[2]. Por ejemplo, un código similar al siguiente:

```
//... Código previo....
cout << "Hemos alcanzado este punto.";
//... Código posterior
//...     ---> Punto donde el programa TERMINA ANORMALMENTE <---
//... Código que no se ejecuta
```

que termina en el punto indicado. Ahora bien, la ejecución de este programa puede fallar en ese punto sin que se obtenga en la salida la línea que indicaba el punto alcanzado. Esto se debe a que la sentencia de salida se realizó correctamente, pero no llegó a obtenerse porque la cadena de caracteres aún estaba en el buffer.

Una forma de solucionarlo es añadir `endl` para que, además de escribir esa línea, se descargue el buffer. Precisamente, la diferencia entre escribir el carácter `'\n'` y `endl` es que este último realiza una descarga del buffer.

Por otro lado, si no queremos añadir el salto de línea, podemos usar la función `flush()` para realizar la descarga del buffer:

```
//... Código previo....
cout << "Hemos alcanzado este punto.";
cout.flush(); // Descarga del buffer
//... Código posterior
```

[2]En este caso, la mejor opción no es añadir líneas hasta encontrar un error, sino usar un programa depurador que nos ofrecerá muchas más posibilidades.

```
    //...   Punto donde el programa TERMINA ANORMALMENTE
    //... Código que no se ejecuta
```

Es decir, cuando escribimos **endl** no sólo mejoramos la legibilidad de lo que escribimos añadiendo el carácter `'\n'`, sino que además hacemos que el buffer se descargue.

10.2.2 Flujos globales predefinidos

La inclusión del archivo de cabecera **iostream** nos permite disponer de varios objetos para la E/S estándar. Concretamente, disponemos de:

- Flujo de entrada **cin**. Corresponde a la entrada estándar.
- Flujo de salida **cout**. Corresponde a la salida estándar.
- Flujo de salida **cerr**. Corresponde a la salida estándar de *error*.
- Flujo de salida **clog**. Corresponde a la salida estándar de *registro o diagnóstico* (*logging*).

Estos flujos se conectan automáticamente a los canales de E/S estándar. Al arrancar un programa, por defecto, hemos supuesto que la entrada se realiza desde el "teclado", y tanto la salida como los errores se conectan a la "pantalla". El flujo **clog** se conecta al mismo sitio que **cerr**, aunque difieren en que el flujo **cerr** no tiene *buffer*.

> **Ejercicio 10.1** Como podemos ver, pueden definirse flujos sin buffer. ¿Tiene sentido que **cerr** no tenga buffer?

Observe que hemos dicho explícitamente teclado y pantalla, cuando técnicamente no es correcto, ya que lo que se conecta es la consola, la cual tratamos informalmente como teclado/pantalla.

Redireccionamiento de E/S

Hemos presentado varias veces la idea de que los flujos predefinidos se conectan, por defecto, al teclado y la pantalla. Realmente, la "conexión" depende del programa que lance nuestro ejecutable. Así, por ejemplo, al ejecutar un determinado intérprete de órdenes[3], será éste el que se encargue de lanzar el programa y establecer los flujos de E/S. Por defecto, la entrada y la salida son las mismas que las del intérprete de órdenes.

Podemos indicar que se desean establecer otras "conexiones" para la E/S, si al lanzar nuestro programa así lo especificamos. Por ejemplo, en mi máquina, en la que uso el programa tcsh como intérprete (en *GNU/Linux*) puedo escribir la orden *ejemplo* —mi programa se llama así— seguida de una serie de parámetros con la siguiente línea (observe que el *prompt* lo representamos con un carácter >):

de forma que se lanza el programa *ejemplo*, pero estableciendo como entrada estándar el archivo *entrada.txt* y como salida estándar *salida.txt*. Al no establecer la salida de error, ésta se mantiene en el lugar por defecto. Si se desea realizar también esa asignación, es posible escribir:

[3] El intérprete de órdenes es el programa que se encarga de recibir nuestras órdenes con una determinada sintaxis, y lanzar las órdenes o programas correspondientes. Normalmente nos presenta un *prompt* para indicar que está esperando una orden; escribimos la orden y pulsamos salto de línea para lanzarla.

donde también se establece el archivo *error.txt* como la salida estándar de error.

El hecho de cambiar las fuentes y destinos de estos datos no debe confundirnos. Para el programa no se ha cambiado nada, ya que en todos los casos sólo conoce un flujo de entrada o salida, independiente del lugar de procedencia o destino.

Por ejemplo, considere un ejemplo diseñado para que el usuario introduzca desde el teclado tres valores enteros y los escriba en la pantalla. El programa se podría llamar *valores* y el listado podría ser el siguiente:

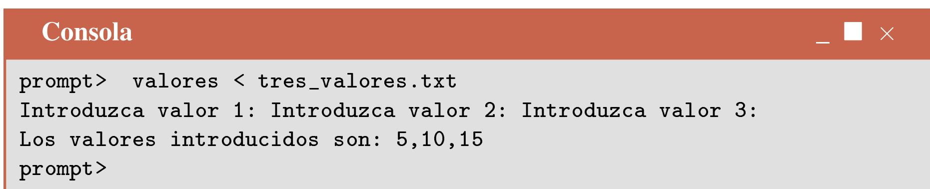
```
#include <iostream>
using namespace std;

int main()
{
   int a,b,c;

   cout << "Introduzca valor 1: ";
   cin >> a;
   cout << "Introduzca valor 2: ";
   cin >> b;
   cout << "Introduzca valor 3: ";
   cin >> c;
   cout << endl << "Los valores introducidos son: "
        << a << ',' << b << ',' << c << endl;
}
```

El usuario puede pensar que las operaciones de entrada están diseñadas para que el programa espere a que se escriba un número y lo finalice con el carácter de salto de línea. Sin embargo, para el programa, `cin` no es más que un flujo de caracteres, que en este programa debe interpretar como tres enteros.

Para que funcione correctamente, sólo es necesario que el flujo de entrada ofrezca tres valores enteros que se encuentren separados por *espacios blancos* (espacios, saltos de línea, etc.). El hecho de que el programa "espere" a que introduzcamos un dato se debe a que la operación de lectura no se puede realizar, ya que el buffer de entrada se encuentra vacío y no se introducirán nuevos valores hasta que el usuario no los escriba y pulse la tecla de confirmación *"Intro"*.

Podemos crear un archivo con los valores de entrada, por ejemplo, con nombre *tres_valores.txt* y el siguiente contenido:

y llamar al programa con el siguiente efecto:



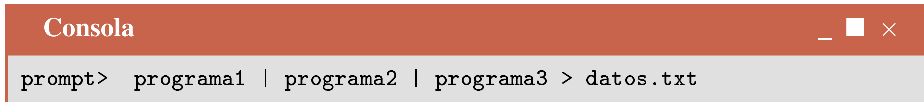
En la primera línea hemos indicado que se ejecute *valores* tomando la entrada estándar desde el archivo *tres_valores.txt*. Las dos siguientes corresponden a la salida estándar del programa —se presentan en pantalla porque no se ha seleccionado ningún destino especial al lanzar el programa—.

El programa no se detiene en ningún momento ya que cuando se ejecutan las instrucciones de entrada, existen datos suficientes en el buffer de entrada. Adicionalmente, puede observar cómo el

archivo de entrada poseía varios espacios en blanco y un salto de línea entre los datos. El efecto del operador >> en la lectura de un entero incluye la eliminación de los separadores antes del siguiente dato[4].

Observe lo cómodo que puede resultar aplicar pruebas a programas que esperan valores de entrada desde el teclado. Tal vez, en sus primeros programas haya reescrito varias veces los mismos datos de entrada para probar un programa que fallaba. Si los salvamos previamente en un archivo, se puede usar como entrada sin que necesitemos reescribirlos en cada ejecución.

Finalmente, es interesante destacar la posibilidad de usar *"tuberías"* —*pipes*— en la línea de órdenes. Podemos indicar que la salida estándar de un archivo se conecta con la entrada estándar del siguiente. Por ejemplo, con la siguiente línea:



indicamos que la salida del primer programa se dirija como entrada estándar del segundo, y la salida de éste, como entrada del tercero. Finalmente, la salida estándar del tercero se guarda como resultado en el archivo datos.txt.

En este caso, la entrada o salida se ha conectado, no con un dispositivo o archivo externo, sino con otro programa. Desde el punto de vista del programador, cada programa recibe o produce un flujo de caracteres, que es independiente de la fuente o destino de los datos.

10.3 Operaciones básicas con flujos

Conocemos algunas operaciones válidas con flujos ya que hemos usado intensivamente los objetos **`cin`** y **`cout`**. Sin embargo, en los temas anteriores hemos supuesto que las entradas de datos eran correctas y las únicas operaciones que se realizaban eran entradas y salidas de datos con los operadores >> y <<.

En esta sección añadiremos nuevos detalles sobre los flujos, incluyendo la detección de errores de E/S y otras funciones útiles para tener nuevas posibilidades.

10.3.1 Tamaño finito de los flujos

Hasta ahora, el tamaño de la entrada no tenía ningún límite (suponíamos que siempre podemos escribir nuevos datos en el teclado, hasta que el programa termine). Sin embargo, en las secciones anteriores hemos mostrado cómo un archivo puede ocupar el lugar de la entrada o salida estándar de un programa. ¿Qué ocurre si, al realizar una entrada desde el archivo, ya no quedan más datos?

Efectivamente, un flujo de datos es una secuencia de caracteres que tiene un fin. Desde este punto de vista, si leemos un archivo asociado a la entrada estándar de un programa, podemos suponer que la entrada está compuesta por una secuencia de n caracteres consecutivos (tamaño del archivo). Una vez leídos estos n caracteres, la lectura de un nuevo carácter provoca un error.

Una vez finalizada la entrada, la lectura de un nuevo carácter (véase más adelante la función `get`) devuelve como resultado la constante especial *`EOF`* (*end of file*) como resultado de la lectura. Este comportamiento permite imaginar el archivo de entrada como una secuencia de n caracteres, seguida por la constante *`EOF`*. En la figura 10.3 se presenta este esquema.

Esto puede llevar a confusión, ya que podemos pensar que los archivos tienen que almacenar un carácter *`EOF`* para indicar el final. Realmente, este comportamiento es independiente de los

[4] Éste es el comportamiento por defecto con el operador >>. Sin embargo, este comportamiento se puede cambiar si modificamos, en el flujo, la bandera *`skipws`*, que indica si se descartan los espacios blancos iniciales. Véase, por ejemplo, Stroustrup[31] o Josuttis[13] para más detalles.

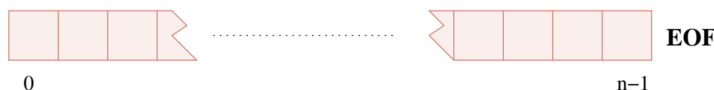


Figura 10.3
Archivo con una constante *EOF* final.

caracteres almacenados en el archivo. El único detalle que debemos conocer es que, si el archivo no contiene más datos, y se lee un nuevo carácter, se obtiene como resultado la constante `EOF`.

Cuando realizamos la entrada desde el teclado, parece que no tiene fin ya que si dejamos de escribir, el sistema simplemente se parará para que introduzcamos nuevos datos. Sin embargo, es posible mandar una señal `EOF` desde el teclado, pulsando las teclas *Ctrl-D*[5]. Lógicamente, si el programa recibe este *final-de-fichero*, entenderá que se ha terminado la entrada.

Por lo tanto, si antes pensaba que el sistema era muy amable al demostrar paciencia cuando escribía un dato y el programa no avanzaba mientras no pulsara la tecla "Intro", ahora puede entender que para el sistema usted no deja de ser un dispositivo realmente lento (tarda tanto que hay que esperar) y torpe (incluso tiene un sistema de confirmación de envío).

Para comprobar el funcionamiento, consideremos las funciones siguientes:

- La función de entrada `get` (`cin.get()`). Lee un carácter del flujo de entrada. Como hemos indicado, si no hay más caracteres, se devuelve la constante `EOF`[6].
- La función de salida `put` (`cout.put(c)`). Tiene un parámetro —el carácter `c`— que escribe en el flujo de salida.

Por ejemplo, el siguiente es un programa que simplemente se limita a repetir, en la salida estándar, lo mismo que se recibe por la entrada estándar. El programa termina cuando se termine la entrada.

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

int main()
{
   int c;
   while ((c=cin.get()) != EOF)
      cout.put(c);
}
```

Observe que el dato leído desde `cin` se guarda en un entero. El objetivo es que `c` permite almacenar cualquier `char` y además el valor especial `EOF` (que puede ser distinto a todos los `char`). Dado que la constante `EOF` no está fijada en el lenguaje, es posible que en su máquina lo ejecute declarando un `char` y parezca que funciona. En cualquier caso, el uso del tipo `int` garantiza el correcto funcionamiento en todos los sistemas que sigan el estándar.

Si ejecuta este programa con el teclado como entrada y la pantalla como salida, verá que cada línea que escribe se repite en la salida estándar. Tal vez piense que cada vez que pulsa una tecla debería escribirse en la salida. Sin embargo, recuerde que los datos introducidos por el teclado no llegan a la entrada hasta que no se confirman con la tecla *Intro*. Por supuesto, cuando desee terminar, mande `EOF` desde el teclado.

Otra forma de ejecutarlo es indicando que la entrada se obtiene desde un fichero. En este caso, obtendrá en la salida el contenido del fichero, puesto que leerá el fichero carácter a carácter, mandándolo a la salida en el mismo orden. Cuando se termine el fichero, la operación de lectura obtendrá `EOF`, que provocará el final del bucle.

[5] Estas teclas son válidas para el sistema *Unix*. Para el caso de *Windows*, se usan las teclas *Ctrl-Z*.

[6] El valor devuelto no es un `char` sino de tipo `int` para poder representar un rango de valores que incluya tanto los caracteres como la constante `EOF`.

Finalmente, es importante que observe la diferencia entre leer los caracteres con la función `get` o leerlos con el operador >>. Recuerde que la lectura con este operador considera que los datos se encuentran separados por *"espacios blancos"*. Se implementa de manera que, en primer lugar, nos saltamos todos los *"separadores"*, hasta llegar a un carácter significativo. Por ejemplo, si en la entrada aparecen de forma consecutiva los caracteres *"espacio, tabulador, salto de línea, espacio, A"* y hacemos una lectura como la siguiente:

```
char c;
cin >> c;
```

se leerá el carácter `'A'`, ya que los separadores previos son eliminados automáticamente.

Sin embargo, cuando usamos la lectura de caracteres con la función `get`, todos los caracteres son relevantes y, por tanto, una lectura como la siguiente:

```
char c;
c= cin.get();
```

obtiene el carácter `' '`(*espacio*). Es decir, para leer el carácter `'A'`, tendremos que hacer 4 lecturas previas con `get`. En nuestro ejemplo, esta es la forma adecuada de realizarlo, puesto que deseamos obtener en la salida estándar exactamente lo que aparece en la entrada.

10.3.2 Estado de los flujos

Cuando se realiza una operación de entrada o salida se pueden producir errores. Por ejemplo, si deseamos leer un entero y escribimos un carácter, la operación de lectura fallará. Para controlar si se ha producido algún error, cada flujo mantiene su estado como un conjunto de banderas (*flags*). En concreto, podemos hablar de 4 banderas:

- `eofbit`. Se activa cuando se encuentra el *final-de-fichero*. Si al realizar una lectura se recibe el carácter `EOF`, el flujo activará esta bandera para indicar que el flujo "llegó al final".
- `failbit`. Se activa cuando no se ha podido realizar una operación de E/S. Por ejemplo, si se intenta leer un entero y se encuentra en el flujo una letra, la lectura no ha tenido éxito y esta bandera se activará. Otro ejemplo, relacionado con la anterior, ocurre cuando se lee un carácter estando la entrada agotada. Se recibe el valor `EOF` y, dado que ha fallado la lectura, se activa también `failbit`. Note que en ambos casos el flujo sigue siendo válido, no se han perdido datos.
- `badbit`. Se activa cuando ha ocurrido un error fatal. Por ejemplo, cuando se solicita realizar una operación con parámetros incorrectos, de forma que el resultado es indefinido, se activa esta bandera para indicar que el flujo está corrupto.
- `goodbit`. Realmente no es una nueva bandera, en el sentido de que está relacionada con las 3 restantes. Esta bandera está "activada" cuando ninguna de las otras lo está.

Relacionadas con estas constantes hay una serie de funciones para comprobar el estado del flujo, así como para cambiarlo explícitamente. Las funciones que permiten consultar el estado son:

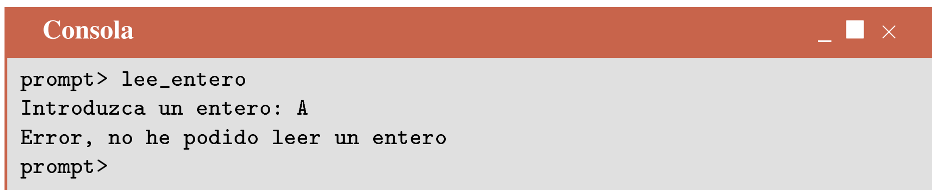
- `good()`. Devuelve si el flujo está bien, es decir, si ninguno de los tres bits anteriores está activado (`goodbit` está "activado".).
- `eof()`. Devuelve si `eofbit` está activado.
- `fail()`. Devuelve si `failbit` o `badbit` está activado.
- `bad()`. Devuelve si `badbit` está activado.

Observe que la función `fail` no está relacionada únicamente con `failbit`. Esta decisión de diseño facilita el desarrollo de código que comprueba el estado del flujo, ya que suele ser muy frecuente que se desee comprobar el estado de estos dos bits y, por tanto, se puede realizar con una sola llamada. Un ejemplo de uso de esta función es:

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

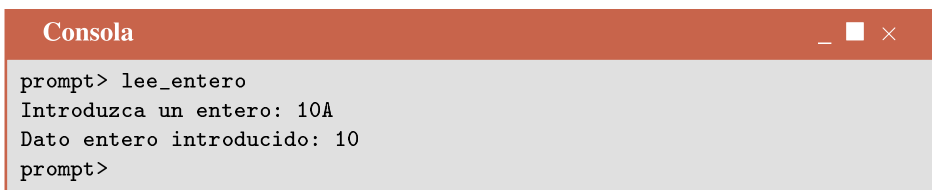
```
int main()
{
   int dato;
   cout << "Introduzca un entero: ";
   cin >> dato;
   if (!cin.fail())
      cout << "Dato entero introducido: " << dato << endl;
   else cout << "Error, no he podido leer un entero" << endl;
}
```

donde se comprueba el estado del flujo mediante la función `fail`. Para ver el efecto de este programa, ejecutamos el programa introduciendo una letra en lugar de un número. Por ejemplo,



```
prompt> lee_entero
Introduzca un entero: A
Error, no he podido leer un entero
prompt>
```

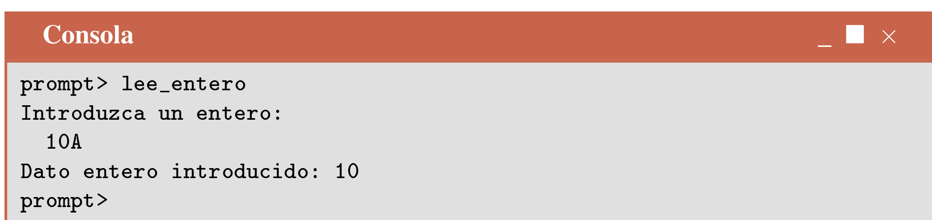
Otro ejemplo distinto de ejecución, en el que introducimos un entero seguido de una letra, es:



```
prompt> lee_entero
Introduzca un entero: 10A
Dato entero introducido: 10
prompt>
```

que nos indica que la lectura del entero se realizó con éxito. Al introducir la cadena *10A*, la entrada del entero comenzó con el dígito `'1'` hasta que termina el entero, es decir, hasta que se encuentra el carácter `'A'`. Por tanto, ha tenido éxito.

Por último, otro ejemplo ilustrativo es la introducción de una serie de espacios blancos seguidos por el entero. Por ejemplo,



```
prompt> lee_entero
Introduzca un entero:
  10A
Dato entero introducido: 10
prompt>
```

donde hemos introducido un salto de línea y dos espacios antes del dato de entrada. Como vemos, el efecto de una lectura con el operador >> implica la eliminación de los “separadores” previos al dato.

Flujos en expresiones booleanas

Para facilitar la comprobación de errores, el lenguaje permite usar directamente el flujo en una expresión booleana. El valor del flujo será **`true`** si no se ha producido un error, es decir, es equivalente a `!fail()`. Lógicamente, también podemos usar el operador `!` sobre el flujo, que devolverá lo contrario (equivalente a `fail()`).

Por ejemplo, podemos reescribir el ejemplo anterior para leer un entero y comprobar si ha habido algún error. El código de la función **`main`** sería:

```
int dato;
cout << "Introduzca un entero: ";
cin >> dato;
if (cin) // !cin.fail()
   cout << "Dato entero introducido: " << dato << endl;
else cout << "Error, no he podido leer un entero" << endl;
```

Además, la lectura de un dato a partir de un flujo devuelve como resultado ese mismo flujo. Por tanto, podemos incluir la lectura en la misma sentencia condicional:

```
int dato;
cout << "Introduzca un entero: ";
if (cin>>dato)
   cout << "Dato entero introducido: " << dato << endl;
else cout << "Error, no he podido leer un entero" << endl;
```

Por supuesto, también es posible usarlo en otras sentencias donde aparezcan expresiones booleanas. Por ejemplo, resulta muy cómodo para realizar lecturas en un bucle mientras que no se produzcan errores.

```
while (cin >> dato)
   cout << "Leído: " << dato << endl;
```

La función *eof* y el final del flujo

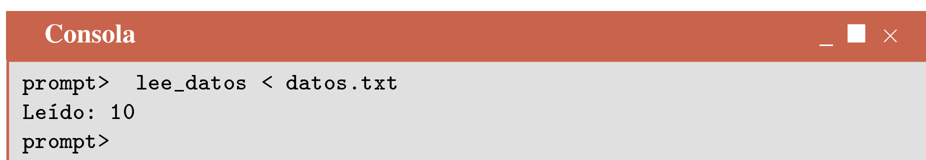
La función `eof` devuelve **`true`** cuando la bandera `eofbit` está activa, lo que ocurre cuando se alcanza el final del flujo.

Esta idea puede llevar a confusión, ya que podemos pensar que cuando leemos el último carácter del archivo se activa esta bandera. Realmente, se activará cuando intentemos leer más allá del último carácter. Recuerde que podemos imaginar los datos de un flujo como una secuencia de caracteres que termina en *EOF*. Cuando se accede a este valor es cuando realmente la función `eof` devolverá **`true`**.

Estos detalles son necesarios si queremos escribir código que tenga en cuenta el final con la función `eof`, ya que debemos ser conscientes de si el flujo alcanza esta marca o no. Por ejemplo, considere el siguiente programa:

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

int main()
{
   int dato;
   while (!cin.eof()) {
     cin >> dato;
     cout << "Leído: " << dato << endl;
   }
}
```

y un archivo de texto (*datos.txt*) que contiene 2 caracteres: `'1'` y `'0'`. La ejecución sería correcta:



Supongamos que el fichero *datos.txt* contiene 3 caracteres: `'1'`, `'0'` y `'\n'`. El resultado sería:

Consola

```
prompt>  lee_datos < datos.txt
Leído: 10
Leído: 10
prompt>
```

En la primera lectura se extraen los dos primeros caracteres del flujo, se escribe el dato leído y se comprueba la finalización con `eof`. Aún no hemos llegado al final, puesto que nos queda por leer el carácter `'\n'`.

Suponga ahora que leemos carácter a carácter con la función `get` que hemos visto anteriormente. Por ejemplo, con el siguiente programa:

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

int main()
{
   char c;
   while (!cin.eof()) {
     c= cin.get();
     cout << "Leído: " << c << endl;
   }
}
```

y ejecutamos de nuevo el programa con el archivo de 2 caracteres `'1'` y `'0'` que usamos anteriormente. La lectura puede sugerir que el bucle itera mientras no sea final de archivo y, por tanto, escribirá dos líneas *"Leído"*. Sin embargo, escribe tres, ya que tras la lectura del segundo carácter, aún no se ha leído `EOF`, aunque estemos al final de la entrada.

En la práctica, es más fácil trabajar con la función `fail` y los flujos en expresiones booleanas. Recordemos, que intentar hacer una lectura cuando no hay más datos en la entrada es un error, y provoca no sólo la activación de `eofbit`, sino también de `failbit`. En este sentido, la mayoría de las comprobaciones se pueden realizar en base a los fallos de lectura.

Por ejemplo, para reescribir el ejemplo anterior, podemos usar la función `get(char& c)`, que lee —en `c`— un carácter del flujo de entrada y devuelve como resultado este flujo.

El código puede usar el flujo devuelto como resultado en una comprobación para saber si la lectura ha tenido éxito o no. Por tanto, una forma sencilla de resolver el mismo problema es:

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

int main()
{
   char c;
   while (cin.get(c))
      cout << "Leído: " << c << endl;
}
```

Ejercicio 10.2 Considere el problema de escribir en la salida estándar la suma de todos los números enteros que se introducen en la entrada hasta el final de entrada. Escriba un programa que realice esa lectura y como resultado escriba la suma de los datos introducidos o un error en caso de que haya habido una entrada incorrecta.

Modificación del estado del flujo

Cuando una operación de E/S falla, el estado del flujo se modifica para indicar este error. En este caso, el código que usa el flujo es el responsable de comprobar si se ha producido ese error y de determinar la acción a realizar. Una de las posibles acciones es modificar el estado del flujo.

Un aspecto fundamental a tener en cuenta en el manejo de flujos es que si una operación falla, las siguientes operaciones de entrada no se realizarán hasta que no se modifique el estado del flujo. Por ejemplo, si escribimos el siguiente código:

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

int main()
{
   int dato1, dato2, dato3;

   cout << "Introduzca dato 1:";
   cin >> dato1;
   cout << "Introduzca dato 2:";
   cin >> dato2;
   cout << "Introduzca dato 3:";
   cin >> dato3;
}
```

y en el primer dato introducimos la letra `'A'`, veremos que las dos siguientes operaciones de entrada no tienen efecto. La ejecución de este programa podría ser la siguiente:

En la primera lectura se ha detectado un error, ya que se esperaba un entero, pero se ha encontrado el carácter `'A'` y, por tanto, se activa `failbit`. Cuando vuelve a las siguientes lecturas, el flujo tiene estado de fallo, por lo que no se realiza ninguna operación. Observe que el problema no es que se intente leer la letra en cada operación, sino que una vez que el flujo tiene un estado de fallo, las operaciones de E/S siguientes no tienen efecto (serán operaciones *no-op*).Por ejemplo, si escribimos las líneas:

```
int dato;
char c;
cin >> dato;
cin >> c;
```

e introducimos el carácter `'A'` en la primera lectura, provocará un fallo. La segunda línea de entrada —que lee un carácter— tampoco funcionará, a pesar de que en el buffer de entrada "está esperando" un carácter.

El problema es que el estado del flujo indica que ha habido un error y no realizará más operaciones hasta que el usuario modifique —explícitamente— ese estado. Podemos usar la función `clear()` para "limpiar" todas las banderas. Por ejemplo, si escribimos las siguientes líneas:

```
int dato;
char c;
cin >> dato;
if (!cin)
   cin.clear();
cin >> c;
```

e introducimos el carácter `'A'` en la primera lectura, provocará un fallo que hará que la condición de la sentencia condicional sea verdadera. En este caso, limpiamos las banderas para que la lectura del carácter se pueda realizar. Por tanto, al final, la variable `c` contendrá el carácter `'A'`.

Por otro lado, no sólo podemos desactivar la banderas, sino que también podemos activar cada una de las banderas. En concreto, la activación de la bandera `failbit` es especialmente interesante ya que nos permite señalizar errores de lectura cuando el formato no corresponda al que deseamos. Por ejemplo, imagine que estamos leyendo objetos de tipo `Racional` definidos como:

```
struct Racional {
   int numerador;
   int denominador; // Debe ser distinto de cero
};
```

compuestos de dos partes: el numerador y el denominador. Como condición para que el número sea correcto, la parte del denominador debe ser distinta de cero. Si realizamos una función de lectura podemos hacer lo siguiente:

```
void Leer (Racional& r)
{
   int n, d;

   cin >> n >> d;
   if (cin) {
      r.numerador= n;
      r.denominador= d;
   }
}
```

de forma que tras la lectura podemos comprobar si ha habido un fallo de lectura al cargar cada uno de los enteros. Por ejemplo:

```
int main ()
{
   Racional r;

   Leer(r);
   if (!cin)
      cerr << "Error: no he conseguido leer un racional" << endl;
   //...
}
```

Sin embargo, este código aceptaría sin problemas la lectura de dos enteros con el segundo siendo cero. Realmente es un error, ya que el valor leído no es un racional. Si queremos una mejor solución, podríamos señalizar el error de lectura cambiando el valor de `failbit` en el flujo. Se podría hacer como sigue:

```
void Leer (Racional& r)
{
   int n, d;

   cin >> n >> d;
   if (d==0)
      cin.setstate(ios::failbit);

   if (cin) {
      r.numerador= n;
      r.denominador= d;
   }
}
```

donde hemos añadido una llamada a la función `setstate` para activar la bandera `failbit`. Observe cómo hacemos referencia a una bandera añadiendo delante `ios::`.

Con este nuevo código ya podemos usar la función `Leer` sabiendo que si el flujo indica que la lectura ha tenido éxito, el objeto que hemos leído es un `Racional` correcto.

10.3.3 Otras funciones de entrada útiles

En esta sección presentamos algunas funciones interesantes que pueden facilitar la implementación de gran número de problemas de E/S. No se incluyen todas, ya que estamos en un tema introductorio, aunque sí algunas que probablemente conformen un buen grupo de utilidades como para resolver la mayoría de sus problemas. En concreto, consideramos las siguientes:

- Ignorar. La función `ignore` nos permite eliminar uno o varios caracteres de la entrada. Existen distintas posibilidades:
  1. Sin parámetros. La función `ignore()` descarta el siguiente carácter.

2. Con un parámetro de tipo entero. La función *ignore*(*n*) descarta *n* caracteres o hasta el final de la entrada.
3. Con dos parámetros: un entero y un delimitador. La función *ignore*(*n*,*delim*) descarta *n* caracteres o hasta que llegamos al delimitador *delim*.

- Consultar siguiente. La función *peek*() nos permite consultar el siguiente caracter de la entrada. No tiene parámetros, ya que es similar a la función *get*() sin parámetros. La diferencia con *get*() es que el carácter sigue estando en la entrada. Por tanto, un nuevo *peek*() devolvería lo mismo, mientras que un *get*() leería lo mismo eliminándolo de la entrada.
- Devolver. La función *unget* nos permite devolver el último carácter a la entrada. No tiene parámetros, pues lo único que hace es devolver el último carácter extraído al flujo de entrada. Por ejemplo, si hacemos un *unget* después de un *get* volveríamos a tener el mismo flujo de entrada. Es importante tener en cuenta que sólo se puede hacer una vez, es decir, dos *unget* consecutivos no funcionan: sólo se puede devolver el último carácter, nunca el penúltimo.

Ejemplo 10.3.1 Escriba un programa que lee desde la entrada estándar un número indeterminado de líneas que contienen los datos asociados a personas. En cada línea aparece —en primer lugar— un número de identificación (por ejemplo, el DNI) seguido por otros campos como el nombre y apellidos. El programa deberá leer toda la entrada y escribir en la salida estándar sólo los números de identificación.

Para resolver el problema usamos un objeto de tipo **string** que pueda almacenar el primer campo de la línea. Con ello, podemos leer una secuencia de caracteres alfanuméricos que estén separados de los demás campos por "espacios blancos".

Una vez leído el número de identificación debemos saltar todos los caracteres hasta la siguiente línea. Para conseguirlo podríamos usar la función *getline* que lee hasta el salto de línea, sin embargo no tiene sentido leer un objeto que no vamos a usar, ya que sólo queremos descartar los elementos hasta el siguiente salto de línea. Es decir, queremos descartar los caracteres hasta que se encuentre el delimitador '\n'. El programa podría ser el siguiente:

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

int main ()
{
   string str;
   while (cin >> str) {
      cout << str;
      cin.ignore(10000,'\n');
   }
}
```

donde usamos la función *ignore* para indicar que se ignoren hasta 10000 caracteres o hasta que encontremos el carácter de salto de línea (que también se descarta). Por supuesto, si una línea puede llegar a tener más de 10000 caracteres deberíamos aumentar este número para asegurarnos que nos saltamos toda la línea[7].

Observe que la función *ignore* descarta también el carácter delimitador. Si quisiéramos que el carácter quedara en el flujo para que siga disponible en la siguiente lectura podríamos devolverlo haciendo **cin**.*unget*().

Fin ejemplo 10.3.1 ■

[7] En la práctica se suele usar el número entero máximo.

Ejemplo 10.3.2 Se desea un programa que lee palabras, es decir, secuencias de una o más letras. Para ello, se decide implementar una función que se salta todos los caracteres hasta encontrar una nueva letra. Escriba una función que salta todos los caracteres de la entrada estándar hasta encontrar una nueva letra.

Básicamente, la idea es ir sacando carácter a carácter del flujo hasta encontrar una letra. Para determinar si es una letra usamos la función **isalpha** que está disponible si incluimos el fichero de cabecera **cctype**.

El problema de usar la función *get* para extraer un carácter del flujo a fin de comprobar si es letra es que al detectar la letra deberíamos devolverlo al flujo. Podríamos hacerlo usando una función que devuelve el último carácter leído pero en lugar de eso podemos usar la función *peek* que consulta el siguiente carácter. En caso de que no sea una letra podemos sacarlo y eliminarlo. Esta idea se puede escribir como sigue:

```
void SiguienteLetra()
{
   while (!isalpha(cin.peek()))
      cin.ignore(); // igual a cin.ignore(1)
}
```

aunque debemos tener en cuenta que tal vez no exista una letra y el flujo se termine. En ese caso, no podemos usar ese bucle pues se convertiría en un bucle infinito esperando a que apareciera una letra. Tenemos que añadir al bucle una condición de fin de flujo como la siguiente:

```
void SiguienteLetra()
{
   while (!isalpha(cin.peek()) && !cin.eof())
      cin.ignore(); // igual a cin.ignore(1)
}
```

Por último, es interesante indicar que algunos programadores usan la función *get*() para descartar un carácter. El código podría ser:

```
void SiguienteLetra()
{
   while (!isalpha(cin.peek()) && !cin.eof())
      cin.get();
}
```

donde vemos que extraemos una carácter del flujo pero descartamos su valor al no asignarlo a ninguna variable.

Ejercicio 10.3 Escriba una función que, utilizando la función anterior *SiguienteLetra*, extraiga una nueva palabra de la entrada estándar. Una palabra está compuesta por una o varias letras consecutivas. Para resolverlo, la función debe buscar la siguiente letra para crear un **string** con todos los caracteres consecutivos que la forman.

Fin ejemplo 10.3.2 ■

10.4 Flujos asociados a ficheros

Los flujos que se han usado hasta ahora son los predefinidos del sistema estándar de C++. Sin embargo, sabemos que es habitual que los programas puedan interaccionar con los ficheros de disco: leyendo el contenido de un archivo, salvando resultados en nuevo archivo, etc. En estas operaciones, el programa sólo necesita saber el nombre del archivo en disco para poder realizar las correspondientes operaciones de E/S. Para implementar estos algoritmos, podemos crear nuevos flujos asociados a ficheros en disco.

En esta sección se van a presentar las bases del uso de flujos asociados a fichero. Más concretamente, vamos a estudiar las herramientas básicas que nos permiten leer y escribir archivos de texto.

Para poder crear flujos asociados a ficheros será necesario incluir el archivo de cabecera `fstream`. Así dispondremos —entre otros— de los siguientes tipos:

- Tipo `ifstream` (**i**nput-**f**ile-**s**tream). Este tipo nos permite crear un *flujo de entrada* asociado a un archivo de disco. Como resultado disponemos de un flujo similar a `cin`. En este caso, cuando consumimos caracteres vamos avanzando en la lectura del fichero.
- Tipo `ofstream` (**o**utput-**f**ile-**s**tream). Este tipo nos permite crear un *flujo de salida* asociado a un archivo de disco. Como resultado disponemos de un flujo similar a `cout`. En este caso, cuando escribimos caracteres vamos añadiéndolos al archivo.

Con estos dos tipos y las operaciones más básicas podemos resolver la mayoría de las necesidades de E/S de problemas simples. Más adelante estudiará la jerarquía de tipos que nos ofrece el lenguaje, así como nuevas operaciones y formas más complejas de salvar información en dispositivos externos.

10.4.1 Apertura y cierre de archivos

La forma de interactuar con un archivo pasa por asociar un flujo de E/S con él. Los pasos que se deben seguir son los siguientes:

1. *Apertura*. El primer paso para poder realizar cualquier cosa es crear en enlace entre el flujo y el archivo en disco. Básicamente, consiste en indicar qué archivo en disco —usando el nombre— está asociado al flujo. Como consecuencia, se crea el objeto con todos los recursos necesarios para poder manejar el flujo e intercambiar información con el fichero.
2. *Uso*. Una vez que se han creado todos los recursos que mantienen el flujo ya podemos usarlo. En esta fase se realizan las operaciones de E/S que deseemos. Es además la que mejor conoce, ya que es la que ha estado aplicando desde que comenzó a programar usando los flujos estándar `cin` y `cout`.
3. *Cierre*. Cuando el flujo deja de usarse se eliminan todos los recursos que se han necesitado para gestionarlo. Además, se terminan de realizar todas las operaciones pendientes para que la información quede almacenada correctamente.

La parte más novedosa se refiere a la apertura y cierre de archivos. La segunda fase —uso del archivo— no se va a ampliar en este tema, ya que usaremos los conocimientos adquiridos en las secciones y temas anteriores. Simplemente tenga en cuenta que:

- Para usar un flujo de tipo `ifstream` puede aplicar todo lo que ha aprendido sobre el objeto `cin`. Cualquier objeto de tipo `ifstream` ofrece todas las operaciones que ofrece `cin` y, además, con idéntico resultado.
- Para usar un flujo de tipo `ofstream` puede aplicar todo lo que ha aprendido sobre el objeto `cout`. Cualquier objeto de tipo `ofstream` ofrece todas las operaciones que ofrece `cout` y, además, con idéntico resultado.

Como consecuencia, todo lo que hemos explicado sobre flujos, incluyendo control de errores, inclusión de buffers, operaciones tales como `get`, `put`, `getline`, etc. están presentes y disponibles ahora. Tenga en cuenta que las operaciones de entrada serán aplicables para objetos `ifstream` y las de salida para objetos `ofstream`.

Apertura

La apertura de un archivo se realiza con la función `open` del flujo. Como parámetro hay que introducir el nombre del archivo que deseamos abrir. Por ejemplo, podemos hacer:

```
#include <fstream> // para tener ifstream
//...
```

```
int main()
{
   ifstream f;

   f.open("datos.txt"); // Apertura
   // ...
}
```

para asociar el flujo `f` al archivo de disco con nombre *datos.txt*.

Lógicamente, la apertura no siempre tiene éxito. Por ejemplo, imagine que el archivo no existe, o tal vez existe pero no tenemos permiso de lectura. En estos casos la apertura habría fallado. La forma de comprobarlo es muy simple puesto que sólo tenemos que recordar cómo funciona el sistema de control de errores en los flujos. Por ejemplo:

```
ifstream f;

f.open("datos.txt"); // Apertura
if (f.fail())
   cerr << "Error, el archivo no se puede abrir para lectura" << endl;
else {
   // ... uso del archivo abierto
}
```

También usando directamente el flujo en la condición:

```
ifstream f;

f.open("datos.txt"); // Apertura
if (!f)
   cerr << "Error, el archivo no se puede abrir para lectura" << endl;
else {
   // ... uso del archivo abierto
}
```

Tenga en cuenta que el fallo en la apertura podría deberse a que el archivo no está en el directorio de trabajo. En los ejemplos anteriores el nombre no contiene información de la carpeta donde se encuentra y por tanto el programa lo buscará directamente en el directorio desde donde se lanzó el programa. Si desea abrir un fichero de otro directorio también puede incluir como parte del nombre la ruta o camino hasta el archivo.

Es probable que ya esté pensando que puede tener un programa que mantiene dos cadenas, una contiene el camino hasta el archivo y la otra el nombre de éste. Efectivamente es una situación habitual, aunque para usarlo tendrá que crear un único objeto con el nombre completo.

```
ifstream f;
string directorio;
string nombre;

cout << "Directorio de trabajo: ";
getline(cin,directorio);
cout << "Indique nombre: ";
getline(cin,nombre);

f.open(directorio+'/'+nombre); // Apertura
if (!f)
   cerr << "Error, el archivo " << nombre
        << " no se puede abrir para lectura" << endl;
else {
   // ... uso del archivo abierto
}
```

donde hemos usado el separador '/' para encadenar el directorio con el nombre con el objetivo de crear el objeto **`string`** que contiene el nombre completo.

Por otro lado, es fundamental conocer cómo funciona la apertura de un flujo de salida, es decir, la apertura de un objeto de tipo **`ofstream`**. La operación es la misma, con la misma sintaxis, aunque no tiene el mismo efecto ya que el archivo podría incluso no existir. El efecto de esta apertura sería la creación del archivo vacío de forma que la escritura en él implica añadir nueva

información al final. Además, si el archivo existe, se vacía. Por tanto, tenga mucho cuidado cuando escriba en un archivo que ya contiene datos, pues se eliminarían en cuanto se abriera.

Lógicamente, la apertura de un archivo de salida también puede fallar. Por ejemplo, tal vez no tengamos permisos en el sistema para escribir en la carpeta que va a contener el archivo, o incluso tal vez el archivo ya existe pero no tenemos permiso para escribir en él. En ambos casos podemos usar el sistema de error de E/S para comprobar si la operación ha tenido éxito.

Finalmente, es interesante saber que existen otras formas de apertura que ofrecen más posibilidades, como podría ser la apertura de un archivo para escritura sin destruir su contenido. Estos contenidos se dejan para más adelante, cuando estudie en profundidad el sistema de E/S.

Cierre

La última fase en el uso de un archivo es el cierre. Cuando ya no tenemos que realizar ninguna operación de E/S en el programa, es importante que se finalicen las operaciones correctamente y se liberen los recursos que se usan para gestionar el flujo. Para hacerlo sólo tenemos que llamar a la función `close` que no contiene parámetros. Por ejemplo, podemos hacer:

```
int main()
{
   ifstream f;

   f.open("datos.txt");
   if (f) {
      // uso...
      f.close(); // cierre: no vamos a usar más el archivo
   }
   else cerr << "Error, el archivo no se puede abrir para lectura" << endl;
}
```

Esta operación es fundamental para que funcionen correctamente los programas. Por ejemplo, imagine que está escribiendo datos en un archivo de disco. El flujo puede estar usando un buffer de forma que si manda un dato realmente no se escribe en disco, sino que se añada al buffer en espera de que sea descargado a disco. Si el programa termina sin cerrar el flujo, el dato no habrá llegado al disco a pesar de que ya lo hemos enviado desde el programa. La operación de cierre termina estas operaciones pendientes dejando el archivo en un estado correcto.

A pesar de su importancia, muchos programadores de C++ tienden a olvidar la operación de cierre. El motivo es que el tipo asociado al flujo se crea como una clase que contiene código para cerrar el flujo cuando deja de existir. Es decir, cuando la variable de tipo **`ifstream`** u **`ofstream`** deja de existir, el compilador se encarga de cerrar el flujo si no lo hemos hecho nosotros. En el ejemplo anterior, cuando acaba **`main`** deja de existir *`f`* y, por tanto, si el archivo está abierto el programa realiza el cierre.

Aunque el sistema ya nos resuelve el problema de cierre, es importante que tengamos en cuenta la necesidad de cerrarlos y, por consiguiente, incluiremos las llamadas a `close` de forma explícita aunque no hagan falta. Note que en algunos casos son necesarias o recomendables, en concreto:

- El objeto se va a volver a usar. Por ejemplo, hemos usado *`f`* para procesar un archivo y queremos volver a usarlo para procesar otro. La varible no deja de existir y por tanto el sistema no llama automáticamente a la función `close`.
- Si hemos terminado con un archivo, podemos cerrarlo para evitar que quede abierto mientras llega el fin de la variable que mantiene el flujo. Cuanto antes se cierre, antes lo dejamos en un estado consistente. Si lo dejamos abierto innecesariamente estamos ocupando recursos y corremos el riesgo de que un fallo inesperado nos haga perder información.
- Queremos cerrar el flujo y comprobar que el estado del flujo no contiene errores para certificar que todo se ha realizado con éxito. Aunque es algo muy poco probable y en muchos casos se ignora, lo cierto es que el cierre de un flujo también puede fallar.

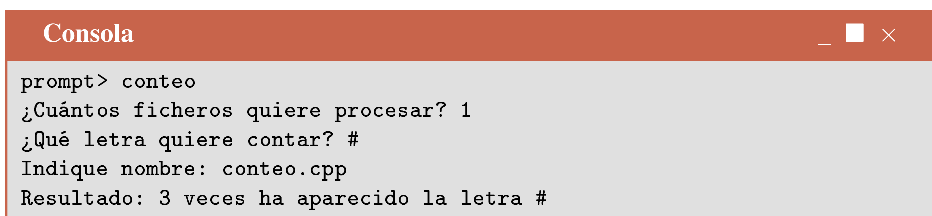
Ejemplo 10.4.1 Necesitamos resolver el problema de contar el número de veces que se repite una letra en una serie de archivos. Escriba un programa que pregunte la letra a buscar y los nombres de los archivos y escriba el resultado del conteo.

Para resolver este problema vamos a escribir un algoritmo sin funciones. Una solución que funciona correctamente es la siguiente:

Listado 8 — Programa *conteo.cpp*.

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

int main()
{
   int nfiles;

   cout << "¿Cuántos ficheros quiere procesar? ";
   cin >> nfiles;

   cout << "¿Qué letra quiere contar? ";
   char letra;
   cin >> letra;
   cin.get(); // Elimina el salto de línea

   int contador= 0;

   for (int i=0; i<nfiles; ++i) {
      ifstream f;
      string nombre;

      cout << "Indique nombre: ";
      getline(cin,nombre);

      f.open(nombre);
      if (!f)
         cerr << "Error, el archivo " << nombre
              << " no se puede abrir para lectura" << endl;
      else {
         int caracter;
         while ( (caracter=f.get()) != EOF )
             if (caracter==letra)
                contador++;
      }
   }

   cout << "Resultado: " << contador
        << " veces ha aparecido la letra " << letra << endl;
}
```

Una posible ejecución de este programa es:



donde hemos usado el mismo programa fuente como entrada de texto para contar el carácter.

En esta solución hemos incluido un bucle que abre múltiples archivos procesando su contenido para comprobar cada uno de los caracteres. Es interesante que observe que el objeto `f` está dentro

del bucle. Cada vez que termina el cuerpo del bucle, el objeto `f` se destruye, por lo que cuando volvemos a empezar vuelve a crearse desde cero. De forma similar, la variable `nombre` vuelve a crearse en cada iteración. Observe que no hay ninguna llamada a `close`, ya que cada vez que la variable `f` desaparece al final del bucle el sistema cierra el archivo automáticamente.

Si queremos reutilizar las mismas variables podemos cambiar el código del bucle, moviendo las dos declaraciones fuera del bucle. En este caso, las dos variables empiezan a existir antes de llegar al bucle y se reutilizan cada vez que iteramos.

```
//...
   ifstream f;
   string nombre;
   for (int i=0; i<nfiles; ++i) {
      cout << "Indique nombre: ";
      getline(cin,nombre);
//...
```

Sin embargo, esta solución no es válida si leemos más de un archivo, ya que la apertura de un archivo con un flujo que no se ha cerrado no es posible. Para volver a utilizar el mismo flujo tenemos que cerrar el anterior. La solución es añadir como última línea el cierre:

```
//...
      else {
         int caracter;
         while ( (caracter=f.get()) != EOF )
            if (caracter==letra)
               contador++;
         f.close(); // Cerramos el archivo para reabrir el siguiente
      }
   }
//...
}
```

Fin ejemplo 10.4.1 ■

Ejercicio 10.4 Escriba un programa que procesa un archivo que contiene un número indeterminado de líneas; cada una de ellas contiene un entero n seguido de n números reales. Por cada una de estas líneas, el programa escribe en la salida estándar una línea que contiene el entero y la sumatoria de los n números reales. Para resolverlo, haga que el programa lea dos nombres de archivo: el de entrada y el de salida.

Ejercicio 10.5 Simplifique el programa anterior para que sirva como filtro en la línea de órdenes. Para ello, debe evitar la lectura de los nombres de los archivos de entrada. En su lugar, use la entrada y salida estándar.

10.4.2 Paso de flujos a funciones

Hasta ahora hemos usado fundamentalmente dos flujos: **`cin`** y **`cout`**, de entrada y salida, respectivamente. Si queremos usarlos con funciones, es necesario conocer los tipos de datos que debemos usar y cómo podemos hacerlo.

Copia de flujos

Los tipos hasta ahora estudiados permitían crear objetos que se podían copiar. Por ejemplo, podíamos considerar la posibilidad de pasar por valor a una función para trabajar con una duplicado, o podíamos realizar una asignación para sustituir el valor de una variable con el de otra.

Una vez que disponemos de nuevos tipos —**`ifstream`** y **`ofstream`**— se nos plantea la posibilidad de realizar las operaciones habituales que hemos aplicado en otros casos. Sin embargo, los tipos asociados a flujos se deben tratar de distinta forma. Es fácil entender que si tenemos un

objeto que corresponde al flujo de información con un archivo, no tiene mucho sentido crear una copia, ya que el flujo es único.

Por otro lado, pensar que un flujo de entrada —donde se realizan sólo lecturas— es un flujo que no se modifica no es cierto. El hecho de leer un carácter de un flujo implica una modificación de éste. Por ejemplo, los buffers pueden cambiar, la posición en donde estamos situados en el flujo cambia, etc. Por tanto, las operaciones, tanto con flujos de entrada como de salida, son operaciones que generalmente modifican el flujo.

Como consecuencia, no está permitida la copia de los flujos. Cuando una función recibe un flujo para trabajar con él siempre será por referencia, sea cual sea la operación que deseemos realizar. Además, no es posible asignar un flujo a otro, ni devolver un flujo como resultado de una función. Incluso evitaremos el paso como referencia constante, pues las operaciones sobre el flujo lo pueden modificar.

Si intenta hacer alguna copia en su programa el compilador generará un error de compilación. Los tipos **ifstream** y **ofstream** se han creado como nuevos tipos de la biblioteca estándar, pero sin dotarlos de las operaciones de copia o asignación. Puede intentar compilar un trozo de código como el siguiente:

```
int ExtraeCaracter (ifstream f)
{
   return f.get();
}
```

para comprobar que el compilador genera un error al detectar el paso por valor de la variable *f*. La forma correcta de pasar *f* a la función es por referencia.

Tipos de *cin* y *cout*

En ningún momento hemos necesitado pasar **cin** o **cout** como parámetro a una función. Son objetos globales, por lo que se pueden usar en cualquier punto del programa. Sin embargo, el poder crear múltiples flujos de E/S y poder pasarlos a funciones como parámetros nos presenta la posibilidad de crear una función que sirva tanto para los flujos estándar como para los que creemos nosotros.

Por ejemplo, imagine que tenemos el tipo *Casilla* que almacena la localización de una casilla de ajedrez y para el que creamos una función que lee la casilla desde un flujo de entrada:

```
struct Casilla {
   int fila;
   char columna;
};

void Leer (ifstream& f, Casilla& c)
{
  f >> c.columna >> c.fila;
}
```

Con este código es fácil imaginar cómo podemos usar un flujo de entrada para leer la casilla desde un archivo *"casillas.txt"*:

```
// ...
int main()
{
   ifstream fichero;

   fichero.open ("casillas.txt");
   if (fichero) {
      Casilla posicion;
      Leer(fichero,posicion);
      if (fichero)
         cout << "Leída: " << posicion.columna << posicion.fila << endl;
   }
//...
```

Sin embargo, esta solución no se puede usar para los flujos estándar ya que no son del mismo tipo. El siguiente código no se puede compilar:

```
// ...
int main()
{
   Casilla posicion;
   Leer(cin,posicion);
//...
```

ya que `cin` no es del tipo `ifstream`.

Para poder crear funciones que reciben los flujos estándares es necesario conocer el tipo que les corresponde: los flujos de entrada son del tipo `istream` y los de salida `ostream`. Como puede ver, los nombres son parecidos, ya que los flujos asociados a ficheros tienen añadida como segunda letra una *f*. Realmente, el tipo `istream` es un tipo más simple que el `ifstream`, así como el tipo `ostream` es un tipo de dato más simple que el `ofstream`.

En principio, al ser de distinto tipo, la solución para leer un objeto `Casilla` desde la entrada estándar pasa por crear una segunda función. Sin embargo, el diseño creado para gestionar la E/S —basado en la herencia, propia de la programación dirigida a objetos— hace que los objetos `ifstream` también puedan pasar por `istream`. Por tanto, cualquier función que recibe un objeto de tipo `istream` también puede recibir un `ifstream`. En nuestro caso, podemos crear la siguiente función:

```
void Leer (istream& f, Casilla& c)
{
   f >> c.columna >> c.fila;
}
```

que sería válida tanto para pasar `cin` como para pasar `ifstream`. Por supuesto, de una forma similar, una función creada para el tipo `ostream` puede recibir el tipo `ofstream`.

> **Ejercicio 10.6** Considere las soluciones de los problemas 10.4 y 10.5. En estos problemas se ha creado un trozo de código muy similar para ficheros y para los flujos estándar. Escriba una función `SimplificarLineas` que tome como entrada dos flujos y que sea válida para los dos programas. Para mostrar su funcionamiento reescriba los programas de ambos ejercicios.

10.5 Flujos y C++98

Como se indicó al principio del tema, en este capítulo se han incorporado algunos detalles del estándar C++11. En concreto, hemos tenido en cuenta que:

- El estándar C++98 no admite la apertura del archivo con un objeto de tipo `string`. Aunque tiene algún sentido desde el punto de vista del diseño del lenguaje, resulta mucho más práctico considerar que es válido el tipo `string`. En el anterior estándar era obligatorio usar la función `c_str()` para obtener un objeto con el formato de *cadena-C*: el único admitido como nombre del archivo. En el estándar C++11 podemos hacerlo tanto con una *cadena-C* como con un objeto `string`.
- La función de apertura de un fichero limpia el estado del flujo. Es lógico que si queremos abrir un flujo y tenemos un objeto que ha dejado de estar asociado a otro archivo, sea posible abrirlo independientemente del estado final del flujo ya cerrado. En C++98 era necesario limpiar el estado para poder usar el flujo en nuevas operaciones. En el nuevo estándar la función de apertura se encarga de realizarlo. Así, la apertura falla si:
  - El flujo aún está asociado a otro archivo.
  - Realmente no es posible la apertura. Por ejemplo, si queremos leer un archivo que no existe.

10.5.1 Apertura con *cadena-C*

En C++98 es importante saber que el nombre del archivo se debe dar como una *cadena-C*. Lo más habitual es tener el nombre del archivo almacenado en un objeto y pasarlo como parámetro a `open` para que lo abra. Si el nombre está almacenado en un objeto de tipo **string**, no podemos pasarlo directamente. La solución a este problema es muy simple pues el tipo **string** ofrece una función para poder usarlo como *cadena-C*: `c_str()`. Por ejemplo:

```
ifstream f;
string nombre;

cout << "Indique nombre: ";
getline(cin,nombre);

f.open(nombre.c_str()); // Apertura
if (!f)
   cerr << "Error, el archivo " << nombre
        << " no se puede abrir para lectura" << endl;
else {
   // ... uso del archivo abierto
}
```

Observe que aunque pueda parecer una restricción importante en la práctica no es difícil resolverla. Para que un código que usa un objeto **string** sea válido en el estándar C++98 sólo tenemos que añadir `.c_str()`.

10.5.2 Reset del estado con *open*

Recordemos el ejemplo 10.4.1 (página 229). En C++98 tendríamos dos errores: por un lado estamos haciendo `open` a un flujo que no se ha cerrado y por otro la apertura de un flujo que tiene errores no es posible. Recordemos que cuando un flujo falla las operaciones de E/S que se realizan sobre el flujo no tienen ningún efecto. En C++98 no es posible hacer un `open` de un flujo que tiene activo `failbit`. Por tanto, para que el ejemplo sea válido, es necesario resetear el estado del flujo:

```
//...
   for (int i=0; i<nfiles; ++i) {
      cout << "Indique nombre: ";
      getline(cin,nombre);

      f.open(nombre.c_str()); // Necesitamos c_str
      if (!f)
         cerr << "Error, el archivo " << nombre
              << " no se puede abrir para lectura" << endl;
      else {
         int caracter;
         while ((caracter=f.get()) != EOF)
            if (caracter==letra)
               contador++;
         f.clear(); // Limpiamos estado para que el flujo vuelva a funcionar
         f.close();
      }
   }
//...
}
```

donde la función `clear()` limpia las banderas `eofbit` y `failbit` activadas al intentar leer más allá del final del fichero, y la función `close()` cierra el uso del archivo y lo deja preparado para un nuevo `open`.

En C++11 no es necesario, ya que la función `open` se encarga de limpiar el estado para intentar la nueva apertura.

10.6 Problemas

Problema 10.1 Considere el tipo `Racional` presentado en este tema:

```
struct Racional {
   int numerador;
   int denominador; // Debe ser distinto de cero
};
```

Escriba una función que lea y devuelva un `Racional` desde un flujo de entrada **`istream`** pasado por parámetro. Tenga en cuenta que el formato del racional debe ser un entero (el numerador) seguido por un carácter `'/'` y otro entero (el denominador). Si no existe ese carácter separador o el denominador es cero el flujo falla y debe activarse la bandera `failbit`.

Problema 10.2 Considere la función de lectura del problema 10.1. Escriba un programa que lea todos los números de tipo `Racional` hasta el final de la entrada estándar. Como resultado debe escribir en la salida estándar los mismos números en formato irreducible y con los denominadores como enteros positivos.

Problema 10.3 Considere el programa del problema 10.2. Escriba la solución aplicada sobre dos ficheros –de entrada y salida– con nombres que se solicitan por la entrada estándar.

Problema 10.4 Considere un problema en el que se mantiene un archivo de ventas realizadas por un conjunto de vendedores. Cada vendedor se identifica con un *ID* (podría ser, por ejemplo, un DNI). El fichero de ventas tiene un formato en el que cada línea corresponde a una venta y contiene el *ID* del vendedor y el número de unidades vendidas.

```
<IDvendedor1> <número items1>
<IDvendedor2> <número items2>
...
<IDvendedorn> <número itemsn>
```

Escriba un programa para simplificar el fichero. El programa lee un fichero del tipo indicado y genera otro fichero con una línea por vendedor. El formato de cada línea del archivo de salida será el siguiente:

```
<IDvendedor> <n> <items1> ... <itemsn>
```

Problema 10.5 Después de obtener la solución del problema 10.4, nos damos cuenta de que varios archivos de ventas se convierten a varios archivos simplificados. Para establecer otra estrategia de mezcla, escriba los siguientes programas:

1. Un programa que convierte un fichero de ventas —con parejas *ID* e *items*— a un fichero simplificado. La solución es convertir cada línea en una tripleta con los mismos valores y con un número 1 en el centro. De esa forma, tendrá formato de fichero simplificado.
2. Un programa para resumir un fichero simplificado. Como podemos tener un fichero que contiene el mismo *IDvendedor* en varias líneas, el programa leerá un archivo con ese formato y lo transformará para que no se repitan. Tenga en cuenta que eso implica que en una línea deben aparecer todas las ventas del mismo vendedor.

11 Compilación separada

Introducción 235
Compilación separada 236
El preprocesador: Múltiples inclusiones de archivos cabecera 240
Bibliotecas 246
Espacios de nombres 248
Objetos y múltiples ficheros 252
Problemas 255

11.1 Introducción

Los programas que se han desarrollado hasta ahora son simples y relativamente pequeños. Sin embargo, en la práctica, los programas pueden llegar a ser mucho más grandes y complejos, llegando fácilmente a miles de líneas de código. En estos casos, el modelo presentado hasta ahora —donde se crea un archivo fuente con todo el código— no es el más adecuado.

Cuando el programa es pequeño, se puede obtener una solución basada en un único componente —un único fichero *cpp*— en el que se pueden encontrar todos los detalles. Sin embargo, cuando el tamaño del problema aumenta, por ejemplo miles o cientos de miles de líneas de código, el ser humano es incapaz de manejar tal cantidad de detalles y es necesaria una descomposición en pequeñas partes más independientes —que denominamos *módulos*— y que unidos constituyen la solución buscada.

Por ejemplo, consideremos el caso de un programa de gestión bancaria. Resulta imposible pensar en una solución teniendo en cuenta, simultáneamente, todos los detalles que la componen. No podemos imaginarla si consideramos a la vez problemas como la forma de almacenamiento en disco, operaciones de comunicación con la base de datos central, el dibujo en pantalla de las ventanas de la interfaz, escribir en la impresora, etc.

La solución es dividir el problema en varios subproblemas más sencillos de resolver. Por ejemplo, podemos resolver una parte —o módulo— encargada de implementar funciones para hacer cálculos estadísticos, otro módulo para facilitar la presentación de resultados en pantalla, otro para facilitar los intercambios de información con disco, etc.

En este capítulo, estudiaremos la compilación separada y los espacios de nombres que nos ofrece C++ como herramientas que nos facilitan la descomposición del código en módulos.

11.2 Compilación separada

El lenguaje C++ permite la organización del código en distintos archivos. Un diseño en módulos independientes se traduce fácilmente en código situado en distintos archivos, facilitando así las tareas de desarrollo del software.

Por ejemplo, imagine que se desea desarrollar un programa para la lectura de una serie de datos numéricos y la presentación de valores estadísticos asociados. Para obtener la solución, podemos descomponer el problema en dos partes distintas:

1. Algoritmos que nos permitan realizar cálculos estadísticos.
2. Algoritmos para la lectura y presentación de los datos.

Aunque son dos partes que están relacionadas, son dos problemas en gran medida independientes. Esta independencia nos permite centrarnos en cada uno de ellos, obviando el resto del programa.

En primer lugar, para realizar cálculos estadísticos, podemos considerar una serie de funciones que implementan algoritmos como el cálculo de la media, la varianza, etc. Estas funciones las incluimos en un módulo que podemos denominar *estadistica.cpp*. Su contenido puede ser —en principio— algo como:

Listado 9 — Módulo estadistica.cpp.

```
1  #include <cmath> // fabs
2  #include <vector>
3  using namespace std;
4
5  double Media(const vector<double>& v)
6  {
7     double suma= 0;
8     for (int i=0; i<v.size(); ++i) suma+= v[i];
9     return suma/v.size();
10 }
11
12 double Varianza(const vector<double>& v)
13 {
14    double media= Media(v);
15    double varianza= 0;
16    for (int i=0; i<v.size(); ++i) varianza+= (v[i]-media) * (v[i]-media);
17    return varianza/v.size();
18 }
19
20 double MediaDeDesviaciones(const vector<double>& v)
21 {
22    double media= Media(v);
23    double res= 0;
24    for (int i=0; i<v.size(); ++i)
25       res+= fabs(v[i]-media);
26    return res/v.size();
27 }
```

Por otro lado, podemos centrarnos en el problema del programa que estamos resolviendo. Nuestro problema consiste en leer una serie de números y presentar los estadísticos; por ejemplo, media, varianza, y la media de las desviaciones. Podemos escribir un programa —que podemos denominar *principal.cpp*— que no incluye las definiciones del módulo anterior, aunque hace uso de esas funciones. Su contenido puede ser —en principio— algo como:

Listado 10 — Módulo principal.cpp.

```
1  #include <cmath> // sqrt
2  #include <iostream>
3  using namespace std;
4
```

```
5  double Media(const vector<double>& v);
6  double Varianza(const vector<double>& v);
7  double MediaDeDesviaciones(const vector<double>& v);
8
9  vector<double> LeerDatos()
10 {
11    int n;
12
13    do{
14       cout << "Introduzca número de datos (entero positivo): ";
15       cin >> n;
16    } while (n<=0);
17
18    vector<double> v(n);
19    for (int i=0; i<n; ++i) {
20       cout << "Introduzca dato " << i << ": ";
21       cin >> v[i];
22    }
23    return v;
24 }
25
26 int main()
27 {
28    vector<double> datos;
29
30    datos= LeerDatos();
31
32    cout << "Media: " << Media(datos) << endl;
33    cout << "Varianza: " << Varianza(datos) << endl;
34    cout << "Desviación Típica: "<< sqrt(Varianza(datos))<<endl;
35    cout << "Media de desv.: "<< MediaDeDesviaciones(datos)<<endl;
36 }
```

Estos dos módulos —*estadistica.cpp* y *principal.cpp*— son dos archivos independientes que unidos forman el programa solución. Note que ninguno de los archivos es un programa completo (el primero no tiene **main**, y el segundo no tiene todas las funciones). Para obtener el programa ejecutable, es necesario *"unirlos"* en la solución.

11.2.1 Compilación y enlazado

En una primera aproximación al proceso de obtención del archivo ejecutable que corresponde al programa, recordemos[1] que podemos dividirlo en dos fases:

1. *Compilación*. Corresponde a la traducción propiamente dicha. En nuestro ejemplo, hemos compilado los dos archivos fuente en dos archivos objeto denominados *estadistica.obj* y *principal.obj* (véase la figura 11.1).
   En esta etapa no se resuelven las "referencias" entre distintas partes del programa. Por ejemplo, en el archivo *principal.cpp* aparece la llamada `Media(datos)`, que se compila —traduce— pero sin resolver el lugar donde se encuentra la función definida (de hecho, si estamos compilando sólo este archivo, no se sabe dónde está).
   Note que hay dos operaciones de compilación que son independientes una de otra. Cuando compilamos *principal.cpp*, no se conoce nada acerca de qué otros módulos hay. Sólo podemos confirmar que las funciones usadas están en otro lugar.
2. *Enlazado*. Se resuelven las referencias entre distintas partes del programa. Los dos módulos compilados —ficheros objeto— se tienen que "unir" en un programa. Por ejemplo, la llamada a `Media(datos)` se "enlaza" con el lugar donde se encuentra la definición. Tenga en cuenta que para esta operación trabajamos con los dos módulos juntos, por tanto, podemos resolver las referencias entre ellos.
   Además, también se enlaza con otros recursos que no aparecen en nuestro programa. Por ejemplo, nuestro programa llama a la función **fabs** (desde el módulo *estadistica.cpp*), o a la

[1] Véase la página 17.

función **sqrt** (desde el módulo *principal.cpp*), o incluso realiza operaciones de E/S. Estas operaciones están disponibles en las bibliotecas del sistema, y deben "unirse" en nuestro programa final.

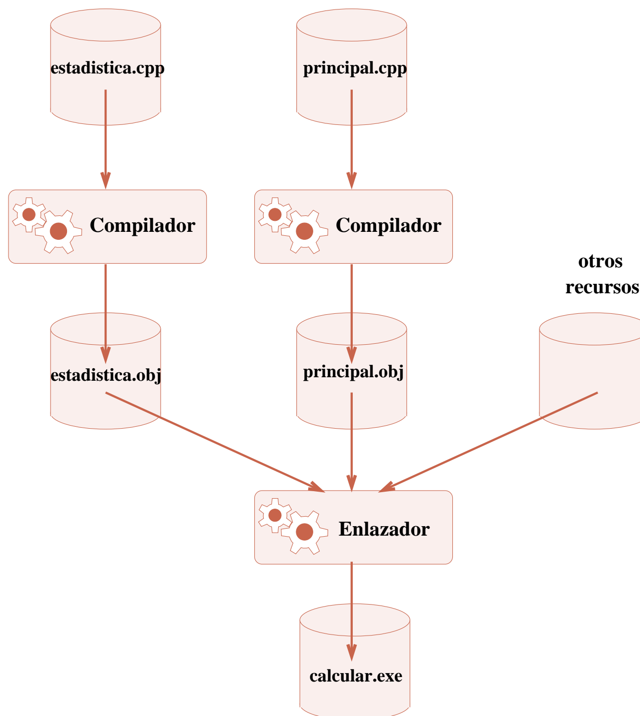


Figura 11.1
Compilación y enlazado.

Por consiguiente, cuando trabajamos en un módulo no es necesario compilar todos los fuentes. Así, podemos ocuparnos de la implementación y compilación de un módulo sin tener que volver a compilar todo el programa. Una vez tengamos todos los módulos compilados, será necesario enlazarlos para obtener el ejecutable.

Cuando hayamos terminado el programa, si necesitamos modificar un módulo, es probable que sólo sea necesario recompilar ese único módulo y volver a enlazarlo con el resto de compilados para obtener el nuevo ejecutable.

11.2.2 El preprocesador. Ficheros h y cpp

La etapa de compilación que hemos mostrado en la sección anterior (véase figura 11.1), necesaria para obtener los archivos objeto, realmente se divide en dos partes (véase 1.3.3, página 19):

1. *Preprocesamiento*.
2. *Compilación*.

En la primera parte —de preprocesamiento— se llevan a cabo algunas operaciones que preparan el código para la compilación propiamente dicha. Una de las tareas consiste en procesar las *directivas de preprocesamiento*. Éstas se distinguen porque comienzan con un carácter `#`. En los ejemplos anteriores se ha usado la directiva `#include`; por ejemplo con los archivos **cmath**, **iostream**.

Recordemos que esta directiva se usa para incluir un archivo externo. Cuando encontramos en el archivo fuente una línea con esta directiva, indicando que se incluya **cmath**, el preprocesador localiza un archivo denominado **cmath** —que tiene código C++— e inserta todo su contenido en el lugar donde se encuentra esa directiva.

A los archivos que se crean con la intención de ser incluidos con la directiva #include los denominamos *archivos cabecera*. En lenguaje C —y en las primeras versiones de C++— estos archivos tienen normalmente extensión *.h* (de *header*). En el actual C++, los archivos cabecera del estándar aparecen sin extensión.

Incluso a los que se "heredan" desde C, se les ha eliminado la extensión y añadido el carácter 'c' delante. Por ejemplo, el nombre del archivo **cmath** aparece desde el archivo math.h que corresponde a C.

Creando archivos cabecera

Para mostrar el objetivo de estos archivos, volvamos a nuestro ejemplo de los cálculos estadísticos. Recordemos que tenemos un módulo *estadistica.cpp* (listado 9, página 236) que contiene algunas funciones que se usan en el módulo *principal.cpp* (listado 10, página 236).

Una de las ventajas de esta separación es que podemos usar el módulo de estadística para otros programas. Por ejemplo, si necesitamos escribir un programa que lee las notas de una serie de alumnos, escribiendo la calificación final a partir de una media, podemos reutilizar el módulo *estadistica.cpp* para usar la función *Media*. El proceso parece muy sencillo, ya que incluso tenemos el módulo compilado. Los pasos serían:

1. Desarrollar el módulo *notas.cpp* que llama a la función *Media*.
2. Compilar este módulo, creando *notas.obj*.
3. Enlazar los módulos *estadistica.obj* y *notas.obj* para crear un ejecutable.

Sin embargo, para poder usar las funciones en el módulo *notas.cpp*, tendremos que volver a declarar las funciones que se utilicen (recuerde que en *principal.cpp* habíamos escrito las 3 cabeceras). Para resolver este problema, cuando creamos un módulo independiente que ofrece "utilidades" a otros módulos, crearemos dos ficheros:

1. Un fichero cabecera (*.h*). Contendrá todas las líneas de código que se necesitarán *incluir* en cualquier otro módulo para para poder usar los recursos que se ofrecen.
2. Un fichero de implementación (*.cpp*). Que contendrá el código que implementa esos recursos, y que no necesita ser incluido para poder usarse.

Por tanto, el fichero de cabecera puede ser considerado la *interfaz* para acceder a la funcionalidad del módulo que hemos creado.

En nuestro ejemplo, podemos crear un archivo *estadistica.h* que tenga el siguiente contenido:

```
double Media(const vector<double>& v);
double Varianza(const vector<double>& v);
double MediaDeDesviaciones(const vector<double>& v);
```

de forma que nuestro archivo *principal.cpp* se modifica como sigue:

```
#include <cmath> // sqrt
#include <iostream>
#include <estadistica.h>
using namespace std;

// Se han eliminado las líneas de prototipos

vector<double> LeerDatos()
{
   // ... igual....
}
int main()
{
   // ... igual....
}
```

En este programa, es el preprocesador el que se encarga de *incluir* las tres líneas de prototipos en el archivo anterior.

Note que ahora la reutilización del módulo *estadistica* es mucho más sencilla. Por ejemplo, en nuestro segundo programa de *notas*, no tenemos más que escribir una directiva `#include` para disponer de lo necesario para llamar a las funciones de ese módulo.

Finalmente, es interesante destacar que algunos programadores prefieren incluir este archivo usando una sintaxis con comillas. Es decir, en el código anterior las primeras líneas serían:

```
#include <cmath> // sqrt
#include <iostream>
#include "estadistica.h"
using namespace std;
```

La diferencia es que con esta sintaxis el procesador no sólo busca el archivo cabecera en los directorios donde sabe que hay archivos de este tipo, sino que también lo busca en el "directorio de trabajo", donde es probable que se encuentre el archivo *cpp* que lo incluye.

Por otro lado, esta sintaxis nos ayuda a identificar este archivo como parte del software que se está desarrollando, haciendo que sea más legible para el programador.

11.3 El preprocesador: Múltiples inclusiones de archivos cabecera

El procesador tiene múltiples posibilidades. Aunque no lo vamos a estudiar por completo, es interesante incluir nuevas directivas que nos permiten realizar la inclusión de archivos cabecera de una forma eficiente y con garantías de que funcionará sin ningún problema.

11.3.1 La directiva *define*

Esta directiva permite la definición de constantes simbólicas. El funcionamiento es muy simple, ya que se procesa por medio de una simple sustitución de texto. La forma de una definición es:

`#define` <identificador> <texto>

de forma que, a partir de esta definición, el preprocesador sustituye todas las ocurrencias de *identificador* por *texto*.

Observe que es una operación que no implica ninguna traducción, es decir, es un simple procesamiento del texto con una operación del tipo *buscar-sustituir*. El resultado es que al compilador le llega el código con el texto reemplazado. Una ejemplo de esta definición es:

```
#define PI 3.14159
```

Con ésta, el preprocesador modificará todo el código que aparece a partir de esa línea, sustituyendo las ocurrencias de `PI` por `3.14159`.

En este sentido, podemos considerarlo una alternativa para definir constantes. Note, sin embargo, que en este caso no hay más que una sustitución y por tanto no existe ninguna comprobación por parte del compilador. Por ejemplo, no tendrá información sobre ninguna constante `PI` de tipo **`double`**. De hecho, el compilador ni sabe que existe `PI`, ya que el preprocesador ha sustituido sus ocurrencias por la correspondiente definición. Además, el código que leemos cuando revisamos el programa es distinto al código que le llega al compilador, ya que éste recibe el resultado de las sustituciones. De esta forma, será más propenso a errores y dificultará su mantenimiento.

Por lo tanto, seguiremos recomendando el uso de constantes como se han visto hasta ahora: usando **`const`**. Obtendremos un código más sencillo de mantener, menos propenso a errores y que puede generar un resultado tan eficiente como el anterior.

Por otro lado, también se puede realizar una definición en la que el *texto* de reemplazo sea vacío. Un ejemplo es el siguiente:

```
#define ACTIVADO
```

El principal interés de este tipo de definiciones es servir como *indicador* o *bandera* de alguna condición.

Por ejemplo, podemos usar un identificador **NDEBUG** para indicar que no se está depurando. A partir de su definición podemos preguntarnos si está definida, en cuyo caso se podría insertar código de depuración (véase la sección 11.3.2).

De hecho, este identificador es conocido por el sistema, y podemos definirlo para eliminar las macros **assert** que se usan para depurar el programa. Por ejemplo, si queremos obtener la versión final de un programa, podemos definir **NDEBUG** para que el compilador descarte las líneas **assert**.

Macros con parámetros

Una forma más compleja de uso de la directiva `#define` es la definición de *macros con parámetros*. El funcionamiento es similar al de constantes, pero incluyendo parámetros en la definición que implican una sustitución un poco más compleja. Por ejemplo, podemos definir la macro:

```
#define SUMAR(x,y) x+y
```

que tiene los parámetros *x* e *y*. En este caso, cuando el precompilador localiza el identificador *SUMAR* en el código, extrae los parámetros asociados que se encuentran a continuación —determina qué es *x* e *y*— y "expande" el texto que indica la definición (escribe esos parámetros con un signo + en medio). Por ejemplo, si encuentra:

```
x= SUMAR(a,sqrt(2.0));
```

reemplaza la macro con:

```
x= a+sqrt(2.0);
```

Observe que la definición realizada no es muy adecuada. Por ejemplo, podemos escribir:

```
x= SUMAR(2,3)*7;
```

en la que deseamos obtener el valor 35. Sin embargo, la expansión resulta como:

```
x= 2+3*7;
```

que obtiene el valor 23. Para evitar este efecto, se incluyen paréntesis sobre todos los parámetros en la definición de la macro, incluso al principio y final de la macro, de la siguiente forma:

```
#define SUMAR(x,y) ((x)+(y))
```

de manera que el sentido de la expresión será correcto en cualquier caso.

Aunque disponer de macros puede resultar muy útil, es importante notar que el control que se tiene sobre el código es mínimo, ya que son sustituciones sin realizar ningún tipo de comprobación. Esto implica que el control de errores es más complejo. Incluso el depurado del programa es más difícil, ya que en el código leemos expresiones que no son realmente las que se están compilando. Por tanto, nosotros las evitaremos, usando funciones en su lugar.

Por otro lado, puede pensar que una clara ventaja de estas macros es que las sustituciones se realizan en tiempo de compilación —realmente, durante el preprocesado— generando un código más rápido. Sin embargo, podemos usar funciones **inline** para solicitar al compilador que intente evitar el coste de la llamada a la función. El resultado es que el compilador puede obtener un código tan rápido como en el caso de las macros pero teniendo mucha más información sobre lo que realmente se está realizando.

Finalmente, es interesante indicar que podemos usar la directiva `#undef` para eliminar la definición de cualquier macro creada con `#define`.

11.3.2 Compilación condicional

El programador puede controlar si un trozo de código se compila por medio de las directivas de compilación condicional. Básicamente, la idea consiste en incluir un trozo de código o no, dependiendo de una condición. Por ejemplo, en el siguiente código:

```
#ifdef DEPURAR
   cerr << "La variable n vale " << n << endl;
#endif
```

se tiene en cuenta la sentencia de salida sólo en el caso de que la macro `DEPURAR` esté definida. Note que para incluir líneas de depuración también podemos tener en cuenta la macro **NDEBUG**[2] ya incluida en el estándar. Así, podemos escribir:

```
#ifndef NDEBUG
   cerr << "La variable n vale " << n << endl;
#endif
```

donde incluimos la línea de salida sólo si *NO* está definida la macro (cuando esté definida no se desea depuración, y por tanto, la línea no debería incluirse en la compilación).

También se pueden considerar otras versiones más complejas de la compilación separada, incluyendo una parte **else**, encadenando distintas estructuras condicionales, o incluso creando expresiones condicionales más complejas. Sin embargo, no las usaremos aquí. El lector interesado puede consultar el preprocesador de C/C++ en *internet* para localizar una referencia.

11.3.3 Un ejemplo más complejo

Considere que desarrollamos un programa más complejo en el que es necesario manejar gráficos. Supongamos que se han creado 3 módulos:

1. Módulo *Punto*. Define un nuevo tipo de dato `Punto` y funciones asociadas. Se crean los archivos *punto.h* y *punto.cpp*.
2. Módulo *Círculo*. Define un nuevo tipo de dato `Circulo` y funciones asociadas. Hace uso del módulo *Punto*. Se crean los archivos *circulo.h* y *circulo.cpp*.
3. Módulo *Gráficos*. Contiene la función **main**; hace uso de puntos y círculos.

En primer lugar, veamos el esquema del módulo *Punto*. Los dos archivos asociados pueden contener el siguiente código:

Listado 11 — Archivo punto.h.

```
1 struct Punto {
2    double x, y;
3 };
4
5 double Distancia (Punto p1, Punto p2);
6 Punto PuntoMedio (Punto p1, Punto p2);
7 // ... otras declaraciones y definiciones
```

Listado 12 — Archivo punto.cpp.

```
1 #include "punto.h"
2
3 // ... declaraciones y definiciones ...
```

Es importante notar algunas novedades con respecto a las secciones anteriores:

[2]Recuerde que el sistema la considera para distinguir si estamos depurando, eliminado las líneas **assert** en caso de estar definida.

- En el archivo *punto.h* hemos incluido la estructura `Punto`. Esta definición es necesaria, ya que cualquier código que haga uso de este módulo —incluya este archivo— necesita conocer el tipo `Punto`. Note, por ejemplo, que los prototipos que vienen a continuación necesitan conocer esta estructura.
- El archivo *punto.cpp* incluye también el archivo *punto.h*. Una opción —que hubiera funcionado— es escribir en el módulo *punto.cpp* otra vez la definición de la estructura `Punto` y prescindir de este `#include`.
  Sin embargo, sería un diseño realmente malo, ya que tendríamos que estar atentos a los posibles cambios de una de las definiciones para cambiar la otra de igual forma. La solución obvia es definirla sólo en el archivo cabecera, e incluirlo con la directiva `#include`.
  En este sentido, es habitual —si no necesario— incluir en los archivos *cpp* sus correspondientes archivos cabecera.

En segundo lugar, veamos el módulo *Círculo*. Siguiendo el mismo esquema del módulo anterior, los dos archivos podrían contener el siguiente código:

Listado 13 — Archivo circulo.h.

```
1 struct Circulo {
2    Punto centro;
3    double radio;
4 };
5
6 double Area (Circulo c);
7 bool Interior (Punto p, Circulo c);
8 // ... otras declaraciones y definiciones
```

Listado 14 — Archivo circulo.cpp.

```
1 #include "circulo.h"
2
3 // ... declaraciones y definiciones ...
```

Sin embargo, el archivo *circulo.cpp* no sería compilable, ya que no conocería la estructura `Punto`. Una modificación que permitiría la compilación, aunque no es la más adecuada, sería incluir el archivo *punto.h* justo antes de incluir *circulo.h*.

Listado 15 — Archivo circulo.cpp.

```
1 #include "punto.h"
2 #include "circulo.h"
3
4 // ... declaraciones y definiciones ...
```

Este es un error típico de programadores sin experiencia. Dado que el error de compilación ocurre en el archivo *circulo.cpp* y se refiere a que la estructura `Punto` es desconocida, el programador tiende a solucionarlo incluyendo, en primera línea, el archivo cabecera que resuelve el problema.

Sin embargo, el problema realmente es que el archivo *circulo.h* depende de *punto.h*. Observe que en *circulo.h* ya se utiliza el tipo `Punto`. Esto queda claro si observa que:

- Si cambia el orden de inclusión de los dos archivos volveremos a obtener el mismo tipo de error. Un programa no debería ser sensible al orden en que se incluyen los archivos cabecera.
- Si cualquier módulo —de este u otro programa— desea usar el módulo *Círculo*, deberá incluir *circulo.h*, pero no será compilable si no antepone la inclusión de *punto.h* (aunque no necesite nada concreto sobre el tipo `Punto`).

La solución a este problema es clara: si es necesario que el código de *punto.h* esté antes que el de *circulo.h*, debemos incluir el primero en el segundo. El archivo quedaría:

Listado 16 — Archivo circulo.h.

```
1 #include "punto.h"
2 struct Circulo {
3    Punto centro;
4    double radio;
5 };
6
7 double Area (Circulo c);
8 bool Interior (Punto p, Circulo c);
9 // ... otras declaraciones y definiciones
```

donde es interesante ser consciente del comportamiento del preprocesador. Cuando un archivo *cpp* lo incluye, el preprocesador no sólo escribe todas estas líneas en el archivo *cpp*, sino que también vuelve a preprocesarlas. Así, al incluir *circulo.h*, se ha incluido también *punto.h* de forma indirecta.

Solución final: evitando la inclusión múltiple

Finalmente, consideremos el último módulo —*Gráficos*— que contiene la función `main` y hace uso de los otros dos. En este caso, como usamos tanto puntos como círculos, podemos hacer una inclusión de las dos cabeceras, de forma que el esquema del código podría ser el siguiente

Listado 17 — Archivo graficos.cpp.

```
1 #include "punto.h"
2 #include "circulo.h"
3
4 int main()
5 {
6    //...
7 }
```

Sin embargo, si pensamos en la salida del preprocesador, esta solución no parece la deseada. Al aparecer dos directivas `#include`, el preprocesador incluye el código de los ficheros asociados. Ahora bien, el archivo *circulo.h* incluye también *punto.h*. Por tanto, el resultado del preprocesador es la inclusión doble de este archivo. Esta duplicidad provoca que la compilación del archivo resultado contenga errores, ya que el compilador encuentra código repetido (por ejemplo, que la estructura `Punto` se está redefiniendo).

Para evitar este error, podemos hacer que se compile cada archivo una sola vez. La solución es usar las directivas `#define` y `#ifndef`−`#endif` para que un archivo sólo se tenga en cuenta la primera vez. Por ejemplo, si tenemos un archivo cabecera denominado *cabecera.h*, escribiremos un código como sigue:

```
/* Archivo: cabecera.h */

#ifndef _CABECERA_H
#define _CABECERA_H
   // Contenido del archivo
#endif
```

donde el identificador *_CABECERA_H* puede ser cualquier otro, siempre que tengamos cuidado de que sea poco probable que se repita en otro lugar. Así, un identificador válido y que raramente se repite es el mismo nombre del archivo en mayúsculas, con esos caracteres especiales, a fin de que resulta difícil de duplicar.

Observe el comportamiento de este código en caso de que se incluya múltiples veces. El resultado del preprocesador será algo como:

```
/* Archivo: cabecera.h */

#ifndef _CABECERA_H
#define _CABECERA_H
   // Contenido del archivo
#endif
/* Archivo: cabecera.h */

#ifndef _CABECERA_H
#define _CABECERA_H
   // Contenido del archivo
#endif
// incluso otras repeticiones...
```

Al llegar a la línea 3, si no se ha definido aún *_CABECERA_H*, tendrá en cuenta el código de la línea 4 y siguientes (hasta `#endif`). Al llegar a la línea 9, el identificador *_CABECERA_H* ya se ha definido (ya se incluyó una vez el código de *cabecera.h*), y por tanto, se ignoran las líneas 10 y siguientes (hasta `#endif`).

Si tenemos en cuenta esta forma de evitar la inclusión múltiple, podemos indicar una posible solución final sobre el ejemplo que estábamos viendo. Concretamente:

Listado 18 — Archivo punto.h.

```
#ifndef _PUNTO_H
#define _PUNTO_H
   struct Punto {
      double x, y;
   };

   double Distancia (Punto p1, Punto p2);
   Punto PuntoMedio (Punto p1, Punto p2);
  // ... otras declaraciones y definiciones
#endif
```

Listado 19 — Archivo punto.cpp.

```
#include "punto.h"

// ... declaraciones y definiciones ...
```

Listado 20 — Archivo circulo.h.

```
#ifndef _CIRCULO_H
#define _CIRCULO_H
   #include "punto.h"
   struct Circulo {
      Punto centro;
      double radio;
   };

   double Area (Circulo c);
   bool Interior (Punto p, Circulo c);
   // ... otras declaraciones y definiciones
#endif
```

Listado 21 — Archivo circulo.cpp.

```
#include "punto.h"
#include "circulo.h"

// ... declaraciones y definiciones ...
```

Listado 22 — Archivo graficos.cpp.

```
1 #include "punto.h"
2 #include "circulo.h"
3
4 // ... declaraciones y definiciones ...
5 int main()
6 {
7     // ... código ...
8 }
```

En el archivo *circulo.cpp* no es necesario incluir *punto.h*, ya que viene incluido de forma indirecta desde *circulo.h*. Sin embargo, no es erróneo, ya que las directivas del preprocesador evitarán la doble compilación, e incluso al ponerlo de forma directa, cualquier programador que lo lea será más consciente de la necesidad del módulo *Punto*.

Finalmente, piense en la eficacia de la solución final. Con este código podemos incluir los archivos cabecera tantas veces como queramos y en cualquier orden. El resultado es un programa válido. Es la mejor solución para un usuario de nuestro código. De hecho, es la solución que encontrará en los archivos de cabecera de su compilador.

11.3.4 Dependencias entre archivos

El código del ejemplo anterior se encuentra en los siguientes archivos: *punto.h*, *punto.cpp*, *circulo.h*, *circulo.cpp* y *graficos.cpp*. Estos archivos se usan para crear tres archivos objeto (*punto.obj*, *circulo.obj* y *graficos.obj*) que se unen para obtener el programa ejecutable deseado (*graficos.exe*).

Esta separación en distintos fuentes nos facilita el trabajo sobre una parte del código de todo el proyecto. Por ejemplo, podemos realizar modificaciones en *circulo.cpp* sin que estas modificaciones tengan que relacionarse con ningún código exterior. Además, un cambio en este archivo implica la necesidad de recompilar para obtener un nuevo *circulo.obj*, que a su vez implica que sea necesario enlazar los tres archivos objeto en un nuevo ejecutable. Note que no es necesario recompilar el resto de módulos.

En este conjunto de archivos podemos establecer una serie de dependencias, analizando los que se ven o no afectados tras la modificación de algún archivo fuente. Por ejemplo:

- El archivo *circulo.obj* depende de *punto.h*, *circulo.h*, *circulo.cpp*. Si alguno de estos cambia, tendremos que recompilar para obtener el nuevo fichero objeto.
- El archivo *graficos.exe* depende de los tres archivos objeto.

Cuando se desarrolla un programa con múltiples ficheros, se debe tener un control sobre cada uno de ellos, su estado, las modificaciones, y todo lo necesario para poder generar el archivo (o archivos) finales a partir de los fuentes.

Los sistemas de desarrollo modernos incluyen herramientas para facilitar esta labor. Este tipo de dependencias se generan y controlan automáticamente, de manera que si el desarrollador solicita la generación del programa ejecutable, el sistema se encarga de determinar las etapas necesarias y se ahorra las que no lo son. Tenga en cuenta que en la práctica los programas pueden llegar a ser muy grandes. La compilación de un software complejo puede requerir horas o incluso días en el caso de tener que compilar todo el código fuente.

11.4 Bibliotecas

Seguimos con el ejemplo anterior. Considere que surge la necesidad de desarrollar nuevos programas que necesitan hacer cálculos con gráficos. El conjunto de módulos que hemos desarrollado —*puntos* y *círculos*— resultan muy útiles para nuestra nueva aplicación.

Como vemos, el conjunto de herramientas desarrolladas para el manejo de gráficos es un módulo que se puede considerar independiente de las aplicaciones concretas, y que puede reutilizarse para otros programas. Así, podemos plantearnos desarrollar todo un conjunto de herramientas de gráficos para poder utilizarlas con las aplicaciones que lo necesiten. Por ejemplo, imagine que desarrollamos distintos módulos, como:

- *Punto*: archivos *punto.h*, *punto.cpp*.
- *Círculo*: archivos *circulo.h*, *circulo.cpp*.
- *Polígono*: archivos *poligono.h*, *poligono.cpp*.
- *Spline*: archivos *spline.h*, *spline.cpp*.

que podemos compilar para generar los ficheros objeto *punto.obj*, *circulo.obj*, *poligono.obj* y *spline.obj*.

Cuando desarrollamos un nuevo programa que usa alguna de las herramientas de los módulos anteriores, debemos determinar los módulos que son necesarios para enlazar los correspondientes archivos objeto en el ejecutable final. Esta operación puede resultar muy incómoda y difícil de gestionar, ya que necesitamos un control sobre los archivos y sus contenidos para poder seleccionar los necesarios para cada aplicación.

Una biblioteca es un archivo que permite almacenar de forma conjunta un grupo de módulos con el objetivo de gestionar sus contenidos y facilitar el acceso al código compilado que contiene.

En nuestro problema, podemos generar una biblioteca con las herramientas que se incluyen en los 4 módulos anteriores (véase figura 11.2). Para ello, usamos un programa especial que incluye todos los módulos compilados en un único archivo que llamamos, por ejemplo, *graficos.lib*.

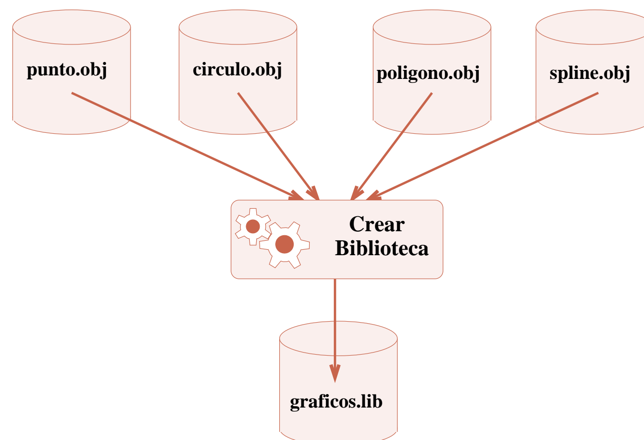


Figura 11.2
Creación de una biblioteca.

Una vez que disponemos de una biblioteca, podemos desarrollar una aplicación y compilarla indicando simplemente que use los recursos incluidos en ella, sin especificar los módulos concretos que se deben usar. Por ejemplo, imagine que creamos 3 archivos (*modulo1.cpp*, *modulo2.cpp*, *modulo3.cpp*) que corresponden a un programa que usa puntos y círculos (funciones de *punto.obj*, *circulo.obj*). Para crear un ejecutable, tendremos que:

- Compilar los 3 archivos en sus correspondientes objeto. Este paso lo realiza el compilador.
- Enlazar los ficheros objeto con la biblioteca y otros recursos. Para esto, sólo es necesario indicar al enlazador que enlace los tres archivos anteriores con la biblioteca que hemos generado.

En la figura 11.3 se presenta gráficamente esta idea. Como vemos, el enlazador recibe el archivo *graficos.lib* para resolver todas las referencias a los módulos que componen la biblioteca. El enlazador es el encargado de extraer los módulos que se usen desde la biblioteca, e incorporarlos como parte del ejecutable. Por supuesto, este archivo ejecutable no incluirá los módulos que no se referencian.

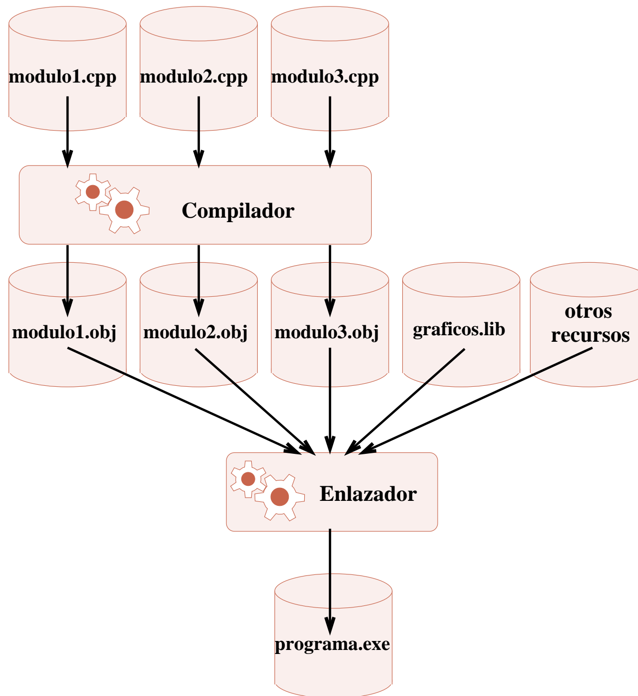


Figura 11.3
Uso de una biblioteca.

Observe que la biblioteca almacena el conjunto de archivos objeto, pero no incluye nada acerca de los archivos cabecera. Cuando se dispone de una biblioteca, normalmente tiene asociados un conjunto de archivos de este tipo para poder realizar la inclusión del código necesario para usarla.

En nuestro ejemplo, el programador que use la biblioteca dispone, no sólo del archivo *graficos.lib*, sino también de las cabeceras asociadas. Por ejemplo, de los archivos *punto.h*, *circulo.h*, *poligono.h* y *spline.h*, o incluso otros que el diseñador considere oportunos (por ejemplo, un archivo cabecera —que podemos llamar *graficos.h*— que incluye los 4 anteriores y que sería suficiente para usar cualquier parte de la biblioteca).

11.5 Espacios de nombres

Los espacios de nombres son una herramienta importante para conseguir la modularización eficaz del código en C++. El lector ya se ha encontrado con el espacio de nombres `std`, aunque sólo nos hemos limitado a escribir una línea `using namespace std`, sin dar más detalles.

En esta sección, sólo introduciremos el concepto y los usos más básicos. Algunos aspectos más avanzados o específicos se pueden consultar en la bibliografía (por ejemplo, en Stroustrup[31], donde se expone de manera bastante extensa), aunque nosotros no haremos uso de ellos.

11.5.1 Creación de espacios de nombres. Nombres calificados

Como acabamos de comprobar, es posible encontrarnos con definiciones de variables, funciones, tipos, etc. que corresponden a un grupo de herramientas estrechamente relacionadas entre sí. Por ejemplo, en las secciones anteriores nos encontramos con las estructuras `Punto` o `Circulo`, así como con un conjunto de funciones que las usan. Además, estas herramientas se pueden unir, en una o varias bibliotecas, para que puedan reutilizarse.

En una aplicación, es probable que usemos varias bibliotecas de forma que tengamos que incluir código desarrollado por distintos programadores. Por ejemplo, considere un programa para realizar dibujos —por ejemplo, en un plano— que usa dos bibliotecas:

1. La biblioteca *gráficos* que hemos visto en las secciones anteriores.
2. Otra biblioteca adicional (que podemos llamar *geometría*), para realizar otros cálculos matemáticos sobre los objetos del plano.

Si se han desarrollado de forma independiente, es posible que encontremos coincidencias en los identificadores. Por ejemplo, podemos encontrar la definición del tipo `Punto` en las dos bibliotecas, o incluso que `Punto` sea un tipo en la primera y una función en la segunda.

Esto provocaría un conflicto difícil de solventar si queremos usar ambas definiciones en el mismo trozo de código, pues es probable que el compilador no pueda distinguir entre los dos casos.

Para resolver este problema, el lenguaje C++ nos ofrece la posibilidad de definir un espacio de nombres, es decir, de agruparlos como partes de un entorno al que asignamos un nombre. La forma de realizarlo es con la palabra clave **`namespace`** y con el siguiente formato:

```
namespace <nombre de namespace> {
  <declaraciones y definiciones>
}
```

de forma que todas las declaraciones y definiciones que se realizan dentro de las llaves se consideran incluidas en ese espacio de nombres. Por ejemplo, podemos escribir:

```
namespace MisGraficos {
   const double PI = 3.14259;
   struct Punto {
      double x, y;
   };
   double Distancia(Punto p1, Punto p2);
}
```

donde los identificadores `PI`, `Punto` y `Distancia` están dentro del espacio de nombres `MisGraficos`
.

Un espacio de nombres crea un nuevo ámbito de forma que todos los identificadores son conocidos dentro de él. Sin embargo, fuera del espacio de nombres es necesario calificar el identificador, incluyendo el espacio de nombres al que pertenece. Para calificar un identificador, usamos el operador de resolución de ámbito "`::`" con la siguiente sintaxis:

```
<nombre de namespace>::<identificador>
```

Por ejemplo, si quiero escribir en la salida estándar la constante `PI` y estoy fuera del espacio de nombres, tendré que escribir:

```
cout << MisGraficos::PI << endl;
```

Es decir, para acceder a cualquiera de los identificadores que están en este espacio de nombres, tenemos que anteponer `MisGraficos::` delante de ellos.

Para ilustrar la existencia de identificadores idénticos, pero en espacios de nombres distintos, considere el siguiente ejemplo:

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

namespace ElPrimero {
   int v=1;
   void AsignarValor (int& n);
}
namespace ElSegundo {
   int v=2;
   void AsignarValor (int& n);
}

void ElPrimero::AsignarValor (int& n)
{ n= v; }
void ElSegundo::AsignarValor (int& n)
{ n= v; }

void EscribirValores()
{
   int numero;
   ElPrimero::AsignarValor(numero);
   cout << "Después del primero: " << numero << endl;
   ElSegundo::AsignarValor(numero);
   cout << "Después del segundo: " << numero << endl;
}

int main()
{
   int numero;
   EscribirValores();
   ElPrimero::v++;
   ElSegundo::v++;
   EscribirValores();
}
```

Observe que en los dos espacios de nombres aparecen los identificadores `v` y `AsignarValor`. Sin embargo, no existe ningún conflicto, ya que están en espacios de nombres distintos. De hecho, ninguno de esos dos identificadores está en el espacio de nombres global, es decir, si escribo en la función **`main`** lo siguiente:

```
cout << v << endl;
```

generará un error de compilación, ya que este identificador no existe en el espacio de nombres global.

Por otro lado, note que hemos declarado las funciones en los espacios de nombres, mientras las definiciones las hemos realizado fuera. Esto realmente no es necesario, ya que podríamos haber realizado la definición también dentro. Sin embargo, en este ejemplo queremos enfatizar cómo, al declarar esas funciones dentro de los espacios, cualquier referencia externa se debe calificar. Así, en la definición hemos tenido que usar los nombres correspondientes con el operador de resolución de ámbito.

Finalmente, note que el identificador `v` se puede usar dentro de las funciones sin calificar, ya que estas corresponden a uno de los espacios de nombres, y por tanto, `v` pertenece al mismo ámbito. El resultado de la ejecución del programa anterior será:

Consola

```
Después del primero: 1
Después del segundo: 2
Después del primero: 2
Después del segundo: 3
```

11.5.2 Eliminando la calificación: *using*

Si un nombre —o grupo de nombres— se utiliza con frecuencia fuera de su espacio de nombres, resulta incómodo tener que calificar todos y cada uno de los identificadores. Podemos hacer que los nombres estén disponibles directamente, con una *directiva de uso*, como sigue:

using namespace <espacio de nombres>;

de forma que todos los identificadores se pueden usar sin calificar. Un ejemplo es la línea que hemos usado en todos nuestros ejemplos:

```
using namespace std;
```

que hace que todos los identificadores del espacio **std** sean accesibles directamente. En este espacio de nombres se encuentran todos los identificadores que introduce el estándar de C++. Es decir, para la biblioteca del estándar se ha creado el espacio **std** de forma que los nombres no pueden "colisionar" con el resto del código.

Si eliminamos esta línea del código el compilador generará errores, ya que no conoce identificadores como **cout**, **endl**, **string**, etc. Sin embargo, no significa que sea recomendable, ya que precisamente al hacer que los identificadores estén disponibles directamente, podemos generar "colisiones" con otros identificadores.

Una solución menos peligrosa para poder usar un identificador sin calificar es hacer una *declaración de uso* para él. Por ejemplo, si queremos usar **cout**, podemos escribir:

```
using std::cout;
```

de forma que podemos usar **cout** sin calificar, aunque el resto seguirá necesitando indicar el espacio de nombres al que pertenece.

```
#include <iostream>
using std::cout;

int main()
{
   cout << "Hola "; // Correcto
   cout << "Mundo" << endl; // Error: endl no conocido
}
```

11.5.3 Espacio de nombres y distintos archivos

Los nombres que se incluyen en un espacio de nombres no tienen que aparecer en una única declaración. Es decir, podemos añadir nuevos nombres al espacio realizando nuevas declaraciones. Por ejemplo:

```
namespace MisFunciones {
   void Funcion1();
   void Funcion2();
}
// ...
namespace MisFunciones {
   void Funcion3();
   void Funcion4();
}
```

permite disponer del espacio `MisFunciones` con 4 nombres añadidos.

El hecho de que los espacios de nombres sean abiertos es especialmente interesante si tenemos en cuenta que en muchos casos queremos incluir, en un único espacio, nombres que se encuentran en distintos archivos. Podemos decir que mientras la división en archivos nos permite tener una separación física del código, los espacios de nombres nos permiten expresar una agrupación lógica.

Cuando separamos el código en archivos independientes, es recomendable que se haga con algún sentido lógico para definir módulos independientes. Cuando usemos espacios de nombres, es recomendable determinar el grupo de declaraciones que queremos agrupar de acuerdo a algún criterio, de forma que enfaticemos la lógica de esa agrupación.

11.5.4 Evitando *using* en archivos cabecera

Las consideraciones sobre espacios de nombres son independientes de la localización del código, de forma que lo que hemos presentado en las secciones anteriores puede aparecer tanto en archivos cabecera *.h* como en archivos de código *.cpp*.

Sin embargo, es importante no olvidar el objetivo de los archivos cabecera que usaremos con la directiva `#include`. Así, si escribimos algo en un archivo cabecera, sabemos que aparecerá en cualquier archivo que lo incluya.

Considere que tenemos que escribir una función que recibe como parámetro un dato de tipo **`string`**, que se encuentra en el espacio de nombres **`std`**. Si necesitamos incluir su prototipo en un archivo de cabecera, podríamos tener, por ejemplo, la siguiente situación:

```
/* Archivo: funciones.h */
string Transformar (string s);
```

donde declaramos la función `Transformar`, a la que se pasa un objeto de tipo **`string`** y devuelve otro del mismo tipo.

Este archivo no sería válido, ya que si lo incluimos en un fichero *.cpp* y lo intentamos compilar, podría provocar un error de compilación al no conocer el tipo **`string`**. Para solucionarlo, podemos añadir una directiva `#include` como sigue:

```
/* Archivo: funciones.h */
#include <string>
string Transformar (string s);
```

Sin embargo, todavía provocaría errores de compilación, ya que se sigue sin conocer el tipo **`string`** (recuerde que está en el espacio **`std`**). El lector puede estar tentado a resolverlo de la siguiente forma:

```
/* Archivo: funciones.h */
#include <string>
using namespace std;
string Transformar (string s);
```

que efectivamente eliminaría esos errores de compilación.

Ahora bien, recuerde que este archivo se incluirá en otros archivos donde el programador es posible que no desee hacer que los nombres de **`std`** estén disponibles de forma directa. Por tanto, para que el archivo sea lo más cómodo posible para el que lo use, es recomendable escribirlo calificando cada nombre como sigue:

```
/* Archivo: funciones.h */
#include <string>
std::string Transformar (std::string s);
```

de forma que el programador que lo usa podrá incluirlo sin necesidad de añadir una directiva de uso que tal vez no desea. Si el programador la necesita, siempre podrá añadirla en sus códigos. Por consiguiente, evitaremos las directivas de uso en todos nuestros archivos cabecera.

11.6 Objetos y múltiples ficheros

Si un programa se puede dividir en distintos archivos, una duda lógica que el lector podría plantearse se refiere a cómo usar variables globales. Cuando deseamos una variable de este tipo y

sólo tenemos un archivo *cpp*, es fácil declararla al comienzo para que todo el código tenga acceso a ella. Ahora bien, ¿cómo se puede hacer con varios archivos *cpp*?

La primera idea podría ser repetir la línea que define la variable en cada uno de los archivos, de la siguiente forma:

```
// archivo1.cpp
int a;
```

y la misma repetición en el segundo archivo:

```
// archivo2.cpp
int a;
```

Sin embargo, esta situación provoca una error de enlace, ya que recibe dos ficheros objeto que incluyen la definición de la misma variable.

Esta situación es similar al caso de las funciones; recuerde que las definimos sólo en un archivo *cpp* mientras que en los otros sólo las declaramos. Para resolver este error, en el caso de las variables aplicamos la misma solución, de la siguiente forma:

```
// archivo1.cpp
int a;
```

que queda indéntico, mientras en el segundo hacemos:

```
// archivo2.cpp
extern int a;
```

donde hemos incluido **extern**, indicando de esta forma que sólo estamos declarando la variable `a`, mientras su definición está en otro lugar (que podría ser también en el mismo archivo).

Cuando el enlazador recibe los dos archivos objeto a enlazar, la definición de la variable `a` sólo aparece en el primer archivo. De nuevo, no es lo mismo una declaración que una definición, como ocurría en el caso de las funciones

Observe que si tenemos un archivo con una definición de variable que queremos que se conozca en todos los demás, podemos hacer la declaración —con la palabra **extern**— en el archivo cabecera correspondiente, de forma que todos los módulos que lo incluyan conocerán esta variable.

11.6.1 Variables locales a un fichero

Por otro lado, también es posible que deseemos definir variables para que sean globales a un único archivo. En este caso, lo que deseamos es que no se usen desde ningún otro, ya que el objetivo de esa variable debe ser independiente del resto del código. Por ejemplo, si definimos la variable `local1` así:

```
// archivo1.cpp
int local1;
```

Sería fácil que otro archivo tuviera acceso a esa variable declarándola como **extern**, o incluso generar un error si, por casualidad, definimos una variable con el mismo nombre.

Por defecto, cuando definimos una variable de esa forma, el lenguaje asume que puede ser utilizada en distintos archivos[3]. Una solución para resolverlo es indicar que es local de la siguiente forma:

```
// archivo1.cpp
static int local1;
```

Con esta definición, ningún archivo puede acceder a `local1`, incluso aunque declare una variable **extern int** `local1`. Esta solución es la que ofrece el lenguaje C y que C++ también recoge. Sin embargo, en este estándar del 98 ya está despreciado y por lo tanto, adoptamos la solución que en éste se propone y que exponemos a continuación.

[3] Se dice que el nombre tiene *enlace externo*. Otro caso es el de las constantes, que tienen *enlace interno*.

Espacios de nombres anónimos

Para resolver el problema de variables locales a un archivo, podría proponerse una solución basada en los espacios de nombres. Por ejemplo:

```
// archivo1.cpp
namespace LocalArchivo1 {
   int local1;
}
```

que impide que cualquier otro archivo pueda acceder a ese nombre de forma directa. Sin embargo, sólo sería válido en caso de que no se accediera al espacio de nombres `LocalArchivo1`. Por ejemplo, si alguien escribe un módulo y quiere acceder a esta variable, puede usar el nombre `LocalArchivo1`. Incluso, podría ocurrir que por casualidad declarara el mismo espacio de nombres.

Para evitarlo, el lenguaje admite los *espacios de nombres anónimos*. Se pueden crear de la siguiente forma:

```
// archivo1.cpp
namespace {
   int local1;
}
```

de forma que la variable `local1` se puede usar en todo el archivo, pero no es posible acceder a ella desde otro. Para entenderlo, el compilador "sustituye" el código anterior por algo así:

```
// archivo1.cpp
namespace <nombre único> {
   int local1;
}
using namespace <nombre único>;
```

con la garantía de que desde ningún otro archivo podrá acceder a esa variable, ya que no es posible conocer el nombre del espacio de nombres que el compilador ha generado automáticamente.

11.6.2 Constantes globales

En las secciones anteriores hemos comentado cómo crear variables globales a varios archivos o a un sólo archivo. Aunque resulta de menor interés en un curso de fundamentos, completamos la discusión incluyendo detalles de cómo crear constantes globales, evitando que el lector intente imitar el comportamiento de las variables.

El caso de las constantes es distinto ya que por defecto tienen enlace interno. Esto implica que si declaramos una constante en un archivo no va a "colisionar" con otra del mismo nombre en otro. Observe que esto permite incluir una constante en un archivo cabecera que sea incluido desde distintos archivos *cpp*, sin que haya errores de enlazado.

En caso de que desee realmente que haya sólo un objeto para todo el programa, podemos indicarlo de forma explícita para que tenga enlace externo. Por ejemplo:

```
// archivo1.cpp
extern const int global= 5;
```

mientras que en el segundo archivo hacemos:

```
// archivo2.cpp
extern const int global;
```

Observe que los dos contienen **`extern`** para hacerlas de enlace externo, aunque la primera de ellas corresponde a una definición, puesto que una declaración con inicialización corresponde a una definición.

A pesar de ello, este caso no es habitual, ya que si deseamos una constante global a todo el programa la pondremos en un archivo cabecera; el enlace interno por defecto nos permitirá compilar el código sin problemas. Además, será más fácil de mantener, ya que sólo aparece en un sitio.

11.6.3 Funciones locales a un fichero

En las secciones anteriores hemos presentado distintos ejemplos de objetos locales y globales, dando lugar a la posibilidad de que se conozcan sólo en un archivo o en varios. Realmente, esta discusión puede extenderse al caso de las funciones. Cuando definimos una función en un archivo, sabemos que podemos usarla en cualquier otro archivo sin más que conocer la declaración.

Como puede ver, las variables globales y las funciones pueden usarse desde otros archivos *cpp*. Cuando un nombre se puede usar desde otros archivos distintos a los que se definió se dice de *enlace externo*. Si sólo se puede usar en el archivo *cpp* donde se definió, se dice de *enlace interno*.

Dado que las funciones son, por defecto, de enlace externo, también podemos decidir que sólo puedan usarse dentro de un archivo *cpp*. En este caso, tendremos que hacer algo similar a lo que hemos hecho con las variables locales a un archivo: añadir la palabra **`static`** antes de la cabecera o incluirlas dentro de un espacio de nombres anónimo. La primera solución es válida para el lenguaje C, mientras que la segunda sólo está disponible en C++.

Finalmente, es necesario indicar que la discusión sobre dónde se pueden colocar objetos o funciones se ha realizado haciendo referencia a si se ponen en un archivo *cpp* o en otro. Realmente, si queremos ser correctos, deberíamos haber usado el término *unidades de traducción*. Las unidades de traducción hacen referencia a los archivos que propiamente se compilan para ser pasados al enlazador, es decir, al resultado del preprocesador. Nosotros no lo hemos diferenciado porque realmente cada uno de los archivos *cpp* de nuestro proyecto ha dado lugar a una unidad de traducción.

11.7 Problemas

Problema 11.1 Considere los siguientes módulos

1. Módulo *Punto*. Se define el tipo `Punto` y un conjunto de funciones para manejarlo.

```
struct Punto {
  double x, y;
};

Punto LeerPunto();
void EscribirPunto (Punto p);
void InicializarPunto (Punto& p, double x, double y);
double CoordenadaX (Punto p);
double CoordenadaY (Punto p);
double Distancia (Punto p1, Punto p2);
Punto PuntoMedio (Punto p1, Punto p2);
```

donde:

a) `LeerPunto`. Devuelve un punto leído desde **`cin`**. La función se limita a leer dos números desde la entrada estándar y devolverlos como un punto.
b) `EscribirPunto`. Escribe en la salida estándar dos valores numéricos que corresponden a las coordenadas del punto.
c) `InicializarPunto`. Asigna a `p` el punto `(x,y)`.
d) `CoordenadaX`. Devuelve la coordenada `x` del punto.
e) `CoordenadaY`. Devuelve la coordenada `y` del punto.
f) `Distancia`. Devuelve la distancia entre los puntos.
g) `PuntoMedio`. Devuelve el punto medio de los dos puntos.

2. Módulo *Círculo*. Se define el tipo `Circulo` y un conjunto de funciones para manejarlo. Este módulo hace uso del anterior.

```
struct Circulo {
   Punto centro;
   double radio;
};

Circulo LeerCirculo();
```

```
void EscribirCirculo(Circulo c);
void InicializarCirculo (Circulo& c, Punto centro,
                                     double radio);
Punto Centro (Circulo c);
double Radio (Circulo c);
double Area (Circulo c);
bool Interior (Punto p, Circulo c);
```

donde:

a) `LeerCirculo`. Lee un círculo desde la entrada estándar. Note que podemos usar la función `LeerPunto` para leer el centro.

b) `EscribirCirculo`. Escribe los tres valores asociados a un círculo en la salida estándar. Note que podemos usar `EscribirPunto` para escribir los valores del centro.

c) `InicializarCirculo`. Asigna a `c` el centro y radio que pasan como parámetros.

d) `Centro`. Devuelve el centro del círculo.

e) `Radio`. Devuelve el radio del círculo.

f) `Area`. Devuelve el área del círculo.

g) `Interior`. Devuelve si el punto `p` pertenece al círculo `c`.

Se desea desarrollar un programa que lea dos círculos, calcule el círculo con centro el punto medio de los dos anteriores y radio tal que pase por dichos centros (véase figura 11.4). Como resultado, se escribirá en la salida estándar el círculo calculado.

Escriba este programa teniendo en cuenta que:

1. El módulo *Punto* y *Circulo* se desarrollarán en dos módulos independientes con un par de ficheros asociados cada uno (*punto.h*, *punto.cpp*, *circulo.h* y *circulo.cpp*).
2. El programa se escribirá en un archivo *graficos.cpp*, que incluye los dos ficheros cabecera *punto.h* y *circulo.h*.
3. Se deberá hacer uso de los tipos `Punto` y `Circulo` a través del conjunto de funciones relacionadas. En concreto, en el archivo *graficos.cpp* no es necesario, en ningún caso, usar el operador '.' de selección de miembro para estos tipos.
4. No se incluirá ningún espacio de nombres para resolverlo (véase problema 11.2).

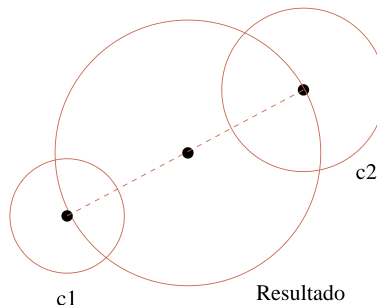


Figura 11.4
Problema a resolver.

Problema 11.2 Considere el problema 11.1. Realice las modificaciones necesarias para que los tipos y funciones de los módulos *Punto* y *Circulo* se incluyan dentro de un espacio de nombres `Graficos2D`. Además, modifique el archivo *graficos.cpp* de acuerdo a ese cambio proponiendo tres posibles soluciones:

1. Incluyendo una directiva de uso.
2. Incluyendo varias declaraciones de uso.
3. Calificando cada uno de los nombres.

12 C++11/14

Introducción ... 257
Inicialización de tipos simples ... 258
El tipo *vector* ... 261
El tipo *string* ... 266
Estructuras, pares y tuplas ... 270

12.1 Introducción

El libro desarrolla sus contenidos en base al estándar C++98, el primer gran estándar sobre este lenguaje, aunque deberíamos más bien referirnos al C++03, el estándar del 2003 con el que realmente trabajamos, aunque no referenciamos al considerarse en gran medida una revisión del anterior que no modificaba el núcleo del lenguaje.

En principio puede resultar algo anticuado, dado que han pasado ya del orden de 20 años, pero es totalmente actual. Realmente no se ha trabajado con un lenguaje obsoleto, sino con un subconjunto del estándar actual. El motivo por el que se realiza de esta forma es, fundamentalmente, evitar contenidos innecesarios para un alumno de primer curso.

Además, este curso no está diseñado únicamente para el estudio de C++, sino que se plantea como base para avanzar en otros temas. Por eso, se intenta ceñir a los fundamentos básicos y evitar detalles que, si bien son interesantes y pueden mejorar los programas, no son fundamentales para asimilar los conceptos que se presentan.

Sin embargo, es importante no limitarse a los contenidos del antiguo estándar, sino que deberíamos dar la oportunidad de añadirlos, especialmente si el alumno pretende continuar estudiando C++ y, sobre todo, si desea trabajar en el futuro con este lenguaje. En este punto del curso, los detalles adicionales que se presentan sobre el estándar actual ya no son tanto una dificultad más, sino una forma sencilla de afinar los conocimientos del estudiante.

Lógicamente, este capítulo no se ha diseñado para incluir todas las novedades del estándar actual, sino sólo los conocimientos fundamentales directamente relacionados con los contenidos de este libro. Esto permite usarlo como texto básico para un curso de C++11 sin más que incluir las distintas secciones de este capítulo a lo largo del temario.

Si opta por estudiar el nuevo estándar a la vez que avanza en los capítulos, un buen punto donde empezar a distinguir las nuevas características podría ser después del capítulo de funciones, aunque también se podrían empezar a incluir detalles desde las primeras declaraciones de objetos. En las siguientes secciones de este tema, se presentan los contenidos que podrían añadirse después de los capítulos de funciones, vectores, cadenas y estructuras, respectivamente.

Sin embargo,dado que el libro se ha creado para un alumno sin ningún conocimiento de programación, no es necesario introducirse en C++11 desde el principio, pues es importante que trabaje con los conceptos más básicos de forma que le sirva para conocer los fundamentos, le permita exportar esos conocimientos a otros lenguajes —como por ejemplo, C— y no se distraiga con una lista de alternativas, muchas de ellas redundantes o difícilmente distinguibles. Cuando el alumno haya asimilados los conceptos relacionados con tipos simples, funciones y tipos compuestos en C++98, será cuando realmente podamos dar una perspectiva más clara de las alternativas y las mejoras que incluye el nuevo estándar.

Es importante tener en cuenta que, a pesar de todo, será en cursos posteriores donde podrá sacarle todo el rendimiento a los contenidos que se introducen en este capítulo, ya que el lenguaje es muy amplio y muchas de las herramientas que se han añadido facilitan especialmente el desarrollo de programas más avanzados. Tenga en cuenta que para hacer un programa simple, el anterior estándar ya ofrece un lenguaje perfecto para su desarrollo.

12.2 Inicialización de tipos simples

A partir del estándar C++11 se amplía el lenguaje para aceptar nuevas formas de inicialización de objetos. El objetivo no es sustituir lo anterior, pues la sintaxis que se ha usado en los temas anteriores sigue siendo válida, sino proponer una forma de inicialización que sea más segura, homogénea e incluso ampliable a los tipos definidos por el usuario.

En primer lugar, recordemos que el formato que se ha presentado para la inicialización de una variable tiene la siguiente sintaxis:

```
<tipo de la variable> <nombre de variable> = <expresión>;
```

Esta forma se ha usado intensivamente en este libro como una forma simple, intuitiva y muy legible. Es conveniente indicar que no debería confundirse con una asignación, pues no estamos asignando un nuevo valor a una variable creada anteriormente, sino dándole el valor inicial de creación. Además, coincide con la forma en que podría hacerse en C.

En C++98, la declaración de objetos que corresponden a una clase definida introduce una nueva sintaxis. Cuando se crea un nuevo objeto se pueden pasar entre paréntesis una serie de parámetros que se usarán para poder inicializarlo. Por ejemplo, si consulta la forma en que se puede declarar un **`vector`** —capítulo 6— o un **`string`** —capítulo 7— podrá comprobar que se propone introducir los parámetros entre paréntesis. Recordemos algunos ejemplos de declaración:

```
vector<int> v;                   // Vector de enteros vacío
vector<double> datos(100);       // 100 valores de tipo double
vector<int> contadores(10,0);    // 10 contadores inicializados con cero
vector<char> ahorcado(8,'-');    // 8 caracteres inicializados con '-';
string s;                        // Cadena vacía
string mensaje("Hola mundo");    // Cadena inicializada la cadena "Hola mundo"
```

Esta sintaxis es muy natural pues —como verá cuando estudie clases en C++— la creación se realiza llamando a una función que crea el objeto. El resultado es una sintaxis distinta a la de los tipos básicos, donde no aparecían paréntesis.

Sin embargo, la idea de que distintos tipos de datos se traten de distinta forma no es muy recomendable, sobre todo si queremos abstraer los conceptos comunes creando clases que funcionan como tipos básicos y a la inversa, especialmente con la programación genérica, como estudiará más adelante. Por tanto, se propone extender la sintaxis de las clases a los tipos básicos, haciendo válida la inicialización con paréntesis:

```
<tipo de la variable> <nombre de variable>(<expresión>);
```

Con esta nueva propuesta, ya disponemos de una forma común de inicialización. Las siguientes líneas son válidas en C++98:

```
bool b(true);
int i(10);
double dato(2.0);
```

No sólo es válida la declaración, sino que podemos usar la sintaxis de las clases para especificar un objeto concreto de un determinado tipo indicando el nombre y los valores con los que se crea:

```
double dato(2.7);
int a= int(dato);   // Crea un entero a partir de dato y lo asigna
int tmp= 3*int();   // Crea un entero por defecto (sin parámetros): valor cero
```

Observe esta última línea, donde incluso especificamos que se cree un objeto de tipo entero sin parámetros, lo que da lugar al valor cero que multiplicado por 3 asigna el cero.

12.2.1 Llaves para tipos simples

En C++98 se pueden declarar objetos de los tipos simples del lenguaje —como **char**, **int**, etc.— que permiten la inicialización con el carácter '=' y con paréntesis, así como objetos de tipo compuesto (*vectores-C* —que no se han visto— y *estructuras-C*) que permiten la inicialización con llaves. Recuerde que en este libro, ya aparece la inicialización de estructuras usando llaves en el capítulo 8.

En C++11 se propone una nueva forma común de inicializar objetos: especificando los valores mediante llaves. El resultado es una forma de inicialización que puede usarse con todos los tipos, los básicos y los de la STL. Incluso se podrá usar para los tipos que defina el usuario.

La sintaxis es muy parecida a la de los paréntesis, pero en este caso se sustituyen por las llaves. La sintaxis es la siguiente:

```
<tipo de la variable> <nombre de variable> { <expresión> };
```

Además, también es posible incluir el carácter '=' después del nombre de la variable, dando lugar a la siguiente sintaxis:

```
<tipo de la variable> <nombre de variable> = { <expresión> };
```

En este capítulo, no vamos a entrar en los detalles que diferencian estas dos formas, pues no son exactamente lo mismo. Ahora mismo, ambas dan lugar al mismo resultado, aunque proponemos usar la primera de ellas[1]. Algunos ejemplos de inicialización son los siguientes:

```
bool b {true};
int i {10};
double dato {2.0};
```

Sin embargo, no es un simple cambio de sintaxis con los mismos efectos. La inicialización con llaves es una forma más segura de hacerlo, pues no permite realizarla si se pierde información, lo que en inglés podrá encontrar como *narrowing conversions*. Aunque el estándar especifica cada una de las situaciones que no se permiten en este tipo de inicialización —como no puede ser de otra forma— en la práctica la idea es muy simple de aplicar: no permite inicializar con un valor en el que perdemos o podríamos perder información. Por ejemplo, en la siguiente línea:

```
int i {2.5};
```

[1] Si quiere un motivo: para escribir menos.

se pierde información, pues el valor de *i* es 2 al perder los decimales. Es un caso de pérdida de información que detecta la inicialización con llaves.

Para reunir distintos ejemplos de inicialización válida en C++11, todas las siguientes líneas se pueden compilar:

```
1  int dato= 5;          // Sintaxis usada en el libro, válida en C
2  int entero= 5.3;      // Sintaxis habitual, pierde información
3  double flotante= 3;   // Sintaxis habitual, usa un int para un double
4
5  int valor(3);         // Sintaxis de constructor de clase
6  int truncado(3.75);   // Sintaxis de constructor, pierde información
7
8  int cero{};           // Sintaxis con {}, valor por defecto: 0
9  int indice{2};        // Sintaxis con {}, un literal int a un int
10 double real{2.75};    // Sintaxis con {}, un literal double a un double
11 int ascii{'a'};       // Sintaxis con {}, un literal char a un int, sin pérdida
12 char c{'a'};          // Sintaxis con {}, un literal char a un char
13 char A{65};           // Sintaxis con {}, un literal int a un char, sin pérdida
```

Observe que la última línea podría considerarse incorrecta, ya que hemos usado un literal de tipo **int** para inicializar un carácter. Efectivamente, podría haber pérdida si el entero fuera otro, pero en este caso no hay ningún problema.

Finalmente, es interesante fijarse en la sintaxis de la línea 8, donde hemos usado las llaves para inicializar en entero *cero* con un valor vacío, lo que corresponde al valor por defecto. Piense en realizar la misma inicialización con paréntesis, de la siguiente forma:

```
int cero();          // Ups!!! no es un entero
```

Esta línea no da lugar a la creación de un objeto de tipo **int**. El problema es que la sintaxis es exactamente la declaración de una función —de nombre *cero*— que no tiene parámetros y que devuelve un entero (véase sección 5.2.1 en página 84). Este error no se puede dar con las llaves, lo que es un motivo más para usarlas.

12.2.2 Deducción del tipo

El listado de posibles declaraciones de la sección anterior, donde tenemos que pensar en el tipo que declaramos y el tipo de literal con el que inicializamos nos permite motivar la introducción de la palabra reservada **auto** para la declaración con deducción de tipo[2].

La idea es sencilla: si estamos indicando exactamente el tipo de dato que es con el valor inicial, no es necesario indicar el tipo, es decir, que lo deduzca el compilador. La sintaxis para este tipo de declaración es la siguiente:

```
auto <nombre de variable> =  <expresión> ;
```

Donde puede ver que no especificamos ningún tipo. El compilador analiza el resultado de la expresión a la derecha del carácter `'='` y deduce el tipo de la variable. Algunos ejemplos son los siguientes:

```
auto cero= 0;                      // tipo int
auto indice= 2;                    // tipo int
auto real= 2.75;                   // tipo double
auto c= 'a';                       // tipo char
auto A= char(65);                  // tipo char

auto mitad_i= 7/2;                 // tipo int
auto mitad_d= 7/2.0;               // tipo double
auto ascii_b= 'a'+1;               // tipo int
auto siguiente_a= char('a'+1);     // tipo char
```

[2]Realmente **auto** ya existía en el lenguaje. Incluso en C, para indicar que una variable es automática. Sin embargo, se suponía implícita y no se usaba.

Observe que no hemos incluido ningún caso con llaves. Aunque se podrían usar, se recomienda evitarlas, al menos hasta que haya avanzado en el estudio de C++ y tenga en cuenta otras posibilidades. De hecho, podría incluirse algún cambio para el estándar C++17 que lógicamente no presentamos en este libro.

Por otro lado, observe cómo se ha modificado la sintaxis desviando los detalles sobre el tipo a la parte derecha. Incluso en casos donde podríamos suponer que es necesaria la declaración sin **auto** —como es la última línea— podemos escribirlo sin dificultad.

12.3 El tipo *vector*

En esta sección vamos a presentar algunos cambios que afectan al uso del tipo vector en el estándar de C++11. No es nuestra intención revisar todos los detalles, pero sí es interesante presentar las modificaciones y aportaciones más interesantes en el contexto de este curso de fundamentos.

12.3.1 Terminación con >>

Presentamos un pequeño detalle sintáctico que no sería extraño que esperara. En la sección 6.4 —página 137— puede ver cómo podemos declarar vectores de vectores. La sintaxis basada en los caracteres <> es muy simple y fácil de entender, aunque incómoda en el caso de anidar declaraciones. En C++98, cuando se declara un vector de un vector, tenemos que tener cuidado de añadir un espacio extra que permita al compilador separar los dos terminadores > >, evitando que se interprete como un único elemento. A partir de C++11 ya se pueden escribir sin separación. Las dos líneas siguientes son válidas en el nuevo estándar:

```
vector<vector<double> > matriz98;   // Válido C++98 y posteriores
vector<vector<double>>  matriz11;   // Válido C++11 y posteriores
```

12.3.2 Bucle para recorrer un contenedor

Realmente es un tema que habría que tratar de forma generalizada para cualquier contenedor, pero que incluimos aquí al ser el tipo **vector** el primer tipo de contenedor con el que hemos trabajado. Básicamente, C++11 ofrece un nuevo tipo de bucle **for**, denominado en inglés *range-based for* y que podríamos llamar *bucle basado en rango*. La razón de este nombre (rango), los detalles de su funcionamiento y cómo puede usarlo para sus nuevos tipos se dejan para más adelante[3].

En general, un contenedor está compuesto de una serie de objetos del mismo tipo. Normalmente, los contenedores dan facilidades para iterar sobre todos los elementos que lo componen. Cualquier contenedor que permite esta iteración, puede usarse en el nuevo bucle propuesto en el estándar C++11. Un ejemplo ideal es un vector de la STL que ya conoce.

La sintaxis del bucle basado en rango es la siguiente:

> **for** (*declaración* **:** *contenedor*)
> { *sentencias* }

Observe que es un bucle que también usa la palabra reservada **for**, pero que contiene dos partes en lugar de tres. Estas partes están claramente diferenciadas con el carácter `':'`, a diferencia del bucle habitual que usa el carácter `';'`:

- La primera parte corresponde a la declaración de la variable que recorrerá cada uno de los objetos del contenedor. La sintaxis de la declaración ya la conoce. Es similar a la declaración de un parámetro en una función.

[3] Más concretamente, cuando estudie el problema de la iteración sobre contenedores basada en tipos *iterador*.

- La segunda parte es el contenedor, es decir, el objeto que contiene una secuencia de elementos que se puede recorrer con un bucle basado en rango. Como comprobará más adelante, casi cualquier tipo de contenedor es válido, incluyendo incluso los *vectores-C* que no se han incluido en este libro.

Por ejemplo, podemos declarar un objeto de tipo **vector**<**double**> para almacenar elementos de tipo **double**, inicialmente con 10 objetos con `0.0` como sigue:

```
vector<double> v(10,0.0); // Contiene 10 veces el 0.0
```

Si queremos obtener el valor de cada objeto contenido usando este nuevo tipo de bucle, podríamos hacerlo de la siguiente forma:

```
cout << "Contenido: ";
for (double d : v) // d toma el valor de cada componente de v
   cout << d << ' ';
cout << endl;
```

Podríamos suponer que *d* se declara en cada iteración del bucle asignándole el valor de `v[0]`, `v[1]`,..., `v[9]`, es decir, recorriendo todos los objetos del contenedor. El resultado de su ejecución será, por tanto:

Este código nos ha permitido obtener cada uno de los objetos del contenedor. Si tenemos en cuenta que la declaración del bucle nos permite declarar también el objeto recorrido por referencia, podríamos usar este bucle para modificar el vector. Por ejemplo, como sigue:

```
for (double& d : v)
  d= 1;
```

que haría que el vector terminara con todos sus componentes valiendo `1`. Observe que en cada iteración, el objeto *d* ha recibido el valor de cada uno de los objetos del contenedor. Por ejemplo, en la primera iteración, *d* recibe por referencia `v[0]`, lo que permite cambiar el original —`v[0]`— cuando cambiamos *d*.

Es probable que lo esté pensando: sí, se puede también declarar la variable para usar el original pero sin modificarlo. Por ejemplo, podemos escribir el siguiente trozo de código:

```
for (double& d : v)        // Modificamos el contenedor
   d= 1;

cout << "Contenido: ";
for (const double& d : v) // Usamos originales sin modificarlos
   cout << d << ' ';
cout << endl;
```

que da lugar a la siguiente salida:

Observe que todo lo que ha aprendido relacionado con las funciones y los parámetros le permite entender cómo podemos declarar la variable que recorre el contenedor. Pero, recuerde que hemos presentado también la palabra **auto** para deducir tipos. El siguiente código:

```
for (auto& d: v)           // auto deduce el tipo double
   d= d*2;

for (const auto& d : v)   // auto deduce el tipo double
   cout << d << ' ';
cout << endl;
```

volvería a modificar el valor de los elementos del contenedor, duplicándolo y mostrándolos a continuación, en ambos casos dejando que el compilador deduzca qué tipo tiene *d*. Por tanto, obteniendo:

12.3.3 Listas de inicialización

C++98 hereda la forma de inicialización de los *vectores-C*, en los que se puede especificar mediante llaves todos los componentes del vector. Esta sintaxis no sólo se utiliza para los *vectores-C*, sino también para las estructuras (véase el capítulo 8).

Para el caso del nuevo tipo de dato **vector**<>, en C++98 no es posible inicializar un vector con una serie de valores concretos. Como sabe, podemos indicar el tamaño del vector y el valor inicial que contendrán todas y cada una de las posiciones. Sin embargo, si queremos inicializar el vector por ejemplo con los valores *2, 4, 6, 8*, no puede hacerse en la declaración, sino con código adicional. No parece un gran problema, pero ciertamente es un paso atrás.

En C++11 se amplía el lenguaje con las nuevas formas de inicialización basada en una lista delimitada por llaves. Ya ha visto en la sección 12.2.1 que podemos usar las llaves para los tipos simples. Realmente, el lenguaje propone usar las llaves para todos los tipos, los simples, los compuestos, los de la STL, e incluso para los que definamos nosotros.

En el caso del tipo **vector**<>, la sintaxis consiste en indicar los valores de inicialización entre llaves, separados por comas. El vector que se crea tendrá tantos elementos como valores haya en la lista de inicialización. Por ejemplo, si escribimos:

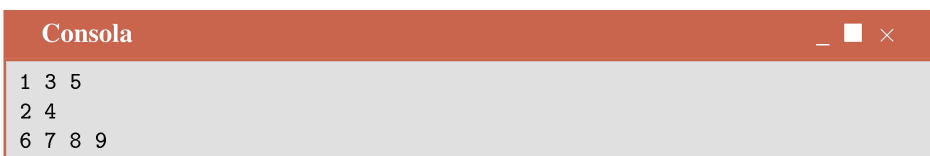
```
vector<double> vec {1, 2.5, 3, 5};
```

se creará un vector *vec* de tamaño 4 con los valores indicados exactamente en ese orden.

Además, la lista de inicialización no sólo puede definirse con una secuencia de valores simples, sino que es una sintaxis que puede anidarse sin problemas, siempre que la estructura coincida con el tipo que se inicializa. Por ejemplo, considere el siguiente trozo de código:

```
vector<vector<double>> bidim {{1, 3, 5},
                              {2, 4},
                              {6, 7, 8, 9}
                             };

for (const auto& fila: bidim) {
   for (auto valor: fila)
      cout << valor << ' ';
   cout << endl;
}
```

La primera línea contiene una declaración de una estructura bidimensional *bidim*, que se inicializa con tres vectores. Cada uno de estos vectores, a su vez, se inicializa con una lista de valores delimitada con llaves. El resultado de su ejecución es:



Observe que al ser una estructura bidimensional la hemos recorrido con dos bucles, para los que hemos aprovechado el bucle basado en rango. Hemos omitido cualquier referencia al tipo de

bidim, por lo que el mismo código podría servirnos para otros tipos; es un código más genérico. Por ejemplo, si cambia **double** por **int** el código es exactamente el mismo. Note, además, que en el primer bucle hemos hecho que la variable *fila* use el original, para evitar la copia.

Finalmente, es importante indicar que la inicialización de un vector con llaves implica la especificación de los valores que contendrá también en el caso de un único valor entero. Si tenemos en cuenta que la forma de crear un objeto permite ahora indicar los parámetros con llaves o con paréntesis, el caso del tipo vector de un único parámetro de tipo entero podría generar dudas. Por ejemplo, las siguientes dos declaraciones son muy distintas:

```
vector<int> v1 (10); // Vector de 10 enteros
vector<int> v2 {10}; // Vector de un entero de valor 10
```

Por lo tanto, aunque dispongamos de la llaves para poder declarar objetos e indicar con qué parámetros se inicializan, sigue siendo necesaria la sintaxis de creación mediante paréntesis, ya que si existen las dos posibilidades —como en el caso de **vector**— la lista de valores de inicialización tendrá prioridad. Dicho de otra forma, la nueva sintaxis no sustituye a la anterior, ya que ésta seguirá siendo necesaria en algunos casos.

Listas de inicialización fuera de la declaración

La versatilidad que se introduce en C++11 nos permite usar las listas de inicialización también fuera de las líneas de declaración. No es una herramienta para darle el primer valor a una variable que definimos, sino que podemos usarla para indicar un valor compuesto para crear un objeto o modificar el valor de un objeto. A continuación se presentan algunos ejemplos en distintas situaciones.

Podemos usarlo en una asignación:

```
vector<double> vec {1, 3, 5};

cout << vec.size() << endl; // Escribe 3
vec= { 1, 2, 3, 2, 1 };
cout << vec.size() << endl; // Escribe 5
```

También podemos usarla con **return**, para indicar el valor que se devuelve. Por ejemplo, en el siguiente trozo de código:

```
// ...
vector<int> Impares()
{
   return {1, 3, 5, 7, 9};
}

int main()
{
   vector<int> vi{2, 4, 6, 8};

   for (auto i: vi)
      cout << i << ' ';
   cout << endl;

   vi= Impares();

   for (auto i: vi)
      cout << i << ' ';
   cout << endl;
}
```

se asigna el valor devuelto por la función *Impares*, donde se ha usado directamente la lista de valores a devolver en la sentencia **return**.

También podríamos haberla pasado a una función. Por ejemplo, la que permite listar los elementos:

```
void Mostrar (const vector<int>& v)
{
   for (auto i: v)
```

```
        cout << i << ' ';
    cout << endl;
}

// ...
Mostrar({1,2,3,4,5,6,7,8,9});
```

Incluso directamente como la lista de valores que se recorren con un bucle **for** basado en rango:

```
int main()
{
    int valor;
    cout << "Valor: ";
    cin >> valor;
    for (auto i: { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
                   31, 37, 41, 43, 47, 53, 59, 61, 67,
                   71, 73, 79, 83, 89, 97})
        if (valor%i==0)
            cout << i << " divide a " << valor << endl;
}
```

12.3.4 Optimización de movimiento de objetos

No está claro si incluir esta sección o no. Realmente no es necesaria para alguien que está empezando; además, sólo se puede entender bien si se han estudiado algunos temas más avanzados del lenguaje. Sin embargo, haremos una pequeña introducción porque, inevitablemente, este nuevo añadido al lenguaje realmente hace necesario matizar algunas indicaciones que se han presentado en el libro.

Durante el estudio de vectores se ha insistido en un aspecto muy importante: la eficiencia en espacio y tiempo. Esta discusión es absolutamente necesaria, pues son objetos que podrían llegar a ser muy grandes. Como consecuencia, se recomienda el paso por referencia constante o la devolución de objetos evitando la sentencia **return**, optando por pasar una variable por referencia en su lugar.

En C++11 se ha ampliado el lenguaje de forma que estas operaciones pueden realizarse de una forma más eficiente: se ha incorporado el concepto de "mover objetos", lo que en inglés encontrará en la bibliografía como *"move semantics"*. La idea es, básicamente, que si tengo que copiar un objeto de un punto a otro del programa —dando lugar a dos objetos, el original y el copiado— pero el original no va a seguir usándose, se debería sustituir la construcción de la copia con mover el objeto original.

La forma en que se realiza está fuera de los objetivos de este libro. La forma en que se consigue es un tema que deberá estudiar, pues no sólo afecta a los tipos de la STL, sino que puede incluirlo para sus propios tipos. Sin embargo, es suficientemente complicado como para sacarlo de este curso.

Sin embargo, es importante tener en cuenta que existe la posibilidad de mover grandes objetos; es el caso de los vectores de la STL, es decir, del tipo **vector**<>. La inclusión de esta posibilidad hace incorrecta la aserción *"la devolución de un vector con* **return** *es una operación muy costosa"*. En C++11, en la mayoría de los casos la operación podría realizarse de forma muy eficiente, independiente incluso del tamaño del vector.

Como puede ver, esta sección no incluye nuevo código o sintaxis que tener en cuenta en sus programas, pero es importante porque:

- Devolver un objeto con **return** no es tan ineficiente como en el estándar anterior. Devolver un tipo **vector** o **string** son operaciones que pueden realizarse más eficientemente.
- Si tiene programas anteriores desarrollados en C++98, una nueva recompilación con el nuevo estándar puede dar lugar a código mucho más eficiente.

Finalmente, no piense que la mejora se limita a poder hacer **return** de un vector. Más adelante estudiará en profundidad el tema de *"move semantics"* y descubrirá que es mucho más amplio de lo que parece. Por ahora, tenga en cuenta que en C++11 los tipos **vector** y **string** son más eficientes de lo que inicialmente hemos presentado.

12.4 El tipo *string*

El nuevo estándar C++11 también incluye nuevas mejoras para manejar cadenas de caracteres. Tal vez, las primeras y más interesantes mejoras deberían centrarse en las nuevas codificaciones de caracteres y la forma en que se escriben los literales. Sin embargo, no vamos a incluir esos comentarios, pues no son especialmente necesarios en el contexto de este curso de fundamentos. En lugar de eso, vamos a comentar cómo afectan los cambios en el lenguaje al tipo **string**, así como alguna utilidad básica que puede aprovechar para sus primeros programas.

12.4.1 El tipo *string* como contenedor

En la sección anterior ya hemos incluido cambios en el lenguaje y cómo pueden afectar al uso de vectores de la STL. Recordemos que el tipo **string** está muy relacionado con el tipo **vector**, ya que conceptualmente es muy similar, aunque resulta una especialización muy importante al destinarse al procesamiento de cadenas de caracteres. Lógicamente, si el nuevo lenguaje nos facilita en gran medida el uso de vectores, también lo hará con el caso del tipo **string**.

Declaración con llaves

La sintaxis en la declaración con llaves es la misma que con vectores. Siendo un contenedor de elementos de tipo **char**, se pueden especificar entre llaves y separados por comas. Sin embargo, no es lo más habitual, pues al ser un cadena se admite la especificación entre comillas dobles. Por ejemplo:

```
string cad1{'u','f','f','f'};
string cad2{"Hola"};
```

son declaraciones de dos cadenas de tamaño 4. Lógicamente, la segunda forma es la que habitualmente se usa, pues es más fácil de escribir y de leer.

Deduciendo el tipo con *auto*

La palabra reservada **auto** también es válida para deducir el tipo de una cadena de caracteres. Sin embargo, debemos tener cuidado con el caso del tipo **string**, ya que un literal de cadena de caracteres como una secuencia delimitada por comillas dobles realmente no es un literal de tipo **string**. Para especificar un valor de este tipo usaremos la palabra **string** explícitamente para crear un objeto de la clase (véase sección 7.2.2 en página 146). Por ejemplo, las dos líneas siguientes se pueden compilar:

```
auto cadena1 = "Ups!";                // Cuidado, cadena1 no es de tipo string
auto cadena2 = string("Hola mundo"); // cadena2 es de tipo string
```

pero el primer caso no corresponde a un objeto de tipo **string**, ya que no se ha especificado como un objeto de esta clase.

El problema de esta forma de declaración es que el tipo que corresponde al primer caso es una cadena de caracteres del lenguaje C, que se maneja con punteros (que seguramente estudiará más adelante). Si queremos usar un objeto de la clase **string**, es necesario pedir explícitamente la creación de un objeto con ese nombre y con un parámetro *cadena-C*. Por tanto, para este curso de fundamentos, deberá seguir usando esta sintaxis para referirse a un objeto **string**.

Finalmente, no podemos terminar esta sección sin puntualizar que realmente se ha creado una forma de literal de tipo **string** en el último estándar C++14. Para ello, se puede añadir una

letra *'s'* como sufijo de una cadena delimitada por dobles comillas para indicar un literal de tipo **string**. Realmente, no es muy distinto a lo que proponemos aquí, pero sí establece formalmente que el literal es de tipo **string**, lo que permitiría deducir correctamente el tipo. A pesar de ello, se recomienda evitar todos estos detalles hasta un curso más avanzado, donde el alumno podrá entender en profundidad todas las alternativas disponibles.

Posición de búsqueda con *auto*

Uno de las primeras ventajas que se obtienen de la deducción de tipos con **auto** es la comodidad a la hora de evitar especificar el tipo que corresponde a una variable. Con una palabra tan corta como **auto**, dejamos al compilador el problema de determinarlo. Un ejemplo claro donde se puede usar es cuando usamos la operación de búsqueda de una cadena en otra.

Recuerde que la operación *find* de la clase **string** devuelve una posición que tiene tipo **string**::*size_type* (véase sección 7.2.7, página 154). Es un nombre incómodo de escribir, por lo que resulta ideal para usar la palabra **auto**. Por ejemplo, el siguiente código localiza la palabra *"mundo"* en una cadena:

```
string cad{"Hola mundo");
string::size_type pos = cad.find("de");

if (pos!=string::npos)
   cout << "Encontrada en: " << pos << endl;
else cout << "No he encontrado la palabra" << endl;
```

En el estándar C++11, la segunda línea se podría haber escrito más cómodamente, prescindiendo del nombre del tipo, como sigue:

```
string cad{"Hola mundo");
auto pos = cad.find("mundo"); // Deduce tipo de pos a partir de find
// ...
```

Cuando profundice en la STL y el uso de plantillas se dará cuenta que esta posibilidad pasa a ser no sólo cómoda sino casi imprescindible.

Bucle basado en rango

Como contenedor de caracteres, podemos usar el bucle **for** basado en rango para recorrer su contenido, desde el primero hasta el último de ellos. Como ejemplo, podemos usar un vector de cadenas —**vector**<**string**>— de forma que manejamos un contenedor de contenedores. En el siguiente trozo de código, declaramos un vector de palabras, las pasamos a mayúscula y las imprimimos:

```
string cadena{"Hola"};
vector<string> palabras { cadena , "mundo" };

for (auto& cad : palabras)
   for (auto& c : cad)
      c= toupper(c);

for (const auto& cad : palabras)
   cout << cad << ' ';
cout << endl;
```

Observe que hemos añadido un objeto **string** —de nombre *cadena*— al vector *palabras* , en lugar de un literal. Por otro lado, en el código aprovechamos la deducción con **auto** para simplificar las declaraciones de las variables que recorren el contenedor. Note que declaramos por referencia y por referencia constante.

Optimización de movimiento de objetos

El caso del tipo **string** también se ve beneficiado del nuevo estándar, donde la copia de objetos puede resultar más eficiente. Recordemos que la optimización permite realizar de una forma más eficiente la copia de un objeto en el caso de que el original deje de usarse.

Es interesante destacar que el tipo **string** es muy similar al tipo **vector**, pero es un caso muy concreto especializado en el tipo **char**, que resulta ser el más pequeño de los tipos disponibles en el lenguaje. Es muy probable que la implementación interna del tipo **string** esté especialmente optimizada para comportarse de forma eficiente. Por ejemplo, podría ser más rápida para objetos de tamaño relativamente pequeño. Sin embargo, no podemos olvidar que en la práctica podría llegar a almacenar un objeto de gran tamaño, lo que hace recomendable tener en cuenta que también podemos aprovechar el movimiento de objetos.

Como comentamos con el tipo **vector**, más adelante estudiará los detalles de cómo funciona este movimiento y entenderá en qué casos podemos aprovecharlo. Además, probablemente introducirá código que explícitamente provoque dichos movimientos.

12.4.2 Conversiones a/desde tipos numéricos

En C++11 se incluyen utilidades para convertir una cadena que contiene un número a un tipo numérico y al contrario. Aunque son varias las posibilidades, pues hay funciones para trabajar con los distintos tipos de datos numéricos, en este curso nos interesan especialmente las funciones relacionadas con el tipo **int** y **double**[4].

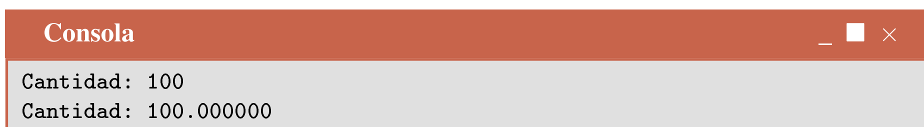
Las nuevas funciones realmente no resuelven algo nuevo, sino que ofrecen una solución muy simple a algo que podríamos hacer de otra forma más elaborada, con más posibilidades, pero más compleja. Por su simplicidad, pueden ser útiles para usarlas en los primeros programas de un estudiante de primer curso.

Conversión a cadena

En primer lugar, puede usar la función *to_string* para convertir un valor numérico a una cadena de caracteres. La sintaxis es la más simple: un parámetro con el número y devuelve un **string** que contiene los caracteres correspondientes. Por ejemplo, puede usarlo de la siguiente forma:

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

int main()
{
   double cantidad;
   cout << "Cantidad: ";
   cin >> cantidad;

   string cad1= to_string(cantidad);
   cout << "Cantidad: " << cad1 << endl;
}
```

El resultado de la ejecución es el siguiente:



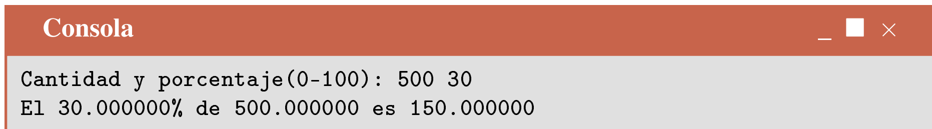
En este ejemplo parece poco útil el uso de *to_string*. En la práctica, lo lógico será usar el resultado para tratar la cadena de alguna forma, por ejemplo, para crear otra cadena.

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

[4]Para trabajar con otros tipos, seguro que le basta con consultar el manual de referencia para entender el resto de funciones que ofrece el lenguaje.

```
int main()
{
   auto cantidad= 0.0;
   auto porcentaje= 0.0;
   cout << "Cantidad y porcentaje(0-100): ";
   cin >> cantidad >> porcentaje;

   auto mensaje= "El " + to_string(porcentaje) + "% de "
                       + to_string(cantidad) + " es "
                       + to_string(cantidad*porcentaje/100);
   cout << mensaje << endl;
}
```

El resultado de la ejecución de este programa es el siguiente:



Observe que en este programa hemos usado la palabra **auto** para deducir los tipos. Es interesante que se fije en las dos primeras declaraciones, donde el uso de **auto** nos ha obligado a inicializar con cero un literal de tipo **double**.

Más interesante es observar el resultado de la ejecución, donde vemos que el formato de escritura del tipo ha producido un número de decimales determinado que no podemos controlar con *to_string*.

Conversión a tipo numérico

El lenguaje ofrece una serie de funciones con nombres del tipo *stoX*—con *X* cambiando según el tipo— para convertir un objeto de tipo **string** al correspondiente tipo entero. En nuestro curso, nos interesan especialmente las funciones *stoi* y *stod*, que permiten convertir a un tipo **int** y **double**, respectivamente.

La forma más simple de conversión consiste en pasarle la cadena con el número como único parámetro para que devuelva el tipo númerico correspondiente. La cadena debería comenzar con un número, tal vez precedido por espacios, para que funcione correctamente. Por ejemplo, podemos usarla como sigue:

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

int main()
{
   string numero{"   100K"};
   int dato= stoi(numero);
   cout << "Entero: " << dato << endl;
}
```

de forma que el número obtenido es el 100. Observe que no importa que la cadena comience por espacios, siempre que le siga un número. Los caracteres que siguen al número no afectan a la conversión.

Aunque se exponen estas funciones en este libro, ya que es posible que le sean útiles para los programas que se resuelven a nivel de un curso de fundamentos, son funciones más complejas de lo que parecen, ya que realmente ofrecen una interfaz más complicada:

- Se pueden añadir un segundo parámetro para poder conocer el primer carácter de la cadena que no se ha convertido.
- Para el caso de los enteros, se puede añadir un tercer parámetro para especificar otra base distinta a la base 10.

- En caso de que no se consiga realizar ninguna conversión, la función eleva una excepción que debería capturarse para procesar el error.

Por tanto, para este curso puede limitar el uso de estas funciones a programas donde sepa que pueden realizarse las conversiones con éxito. Más adelante podrá estudiar en detalle cómo se han diseñado estos parámetros y cómo se pueden manejar correctamente.

12.5 Estructuras, pares y tuplas

En este libro se han presentado tanto la definición de nuevos tipos compuestos heterogéneos con estructuras, como el caso de los tipos **pair** para manejar pares en general. Este contenido es suficiente para el curso, pero la discusión con respecto a C++11 nos permite completarlo con el caso de las tuplas, una generalización de los pares para un número indeterminado de componentes. Además, los nuevos contenidos expuestos en las secciones anteriores nos permiten mostrar nuevos y más completos ejemplos de código en el nuevo estándar.

12.5.1 Inicialización con llaves

La inicialización con llaves de estructuras ya era posible en el estándar del 98. No sólo eso, sino que también lo era en C, lo que indica que realmente estaba disponible desde el primer momento. En C++11 se homogeneiza con el uso de llaves, de forma que los tipos compuestos ya pueden usar las llaves para especificar los distintos componentes del objeto. Además, se pueden indicar los componentes sin necesidad de incluir el carácter `'='`. Por ejemplo, las siguientes líneas son válidas:

```
struct Pareja{
   char c;
   int valor;
};

int main()
{
   Pareja par1= {'X',1000};
   Pareja par2 {'Y',1000};
   // ...
}
```

No sólo es posible especificar una secuencia de elementos con llaves, sino anidarlas especificando tipos compuestos de objetos que a su vez son compuestos. Por ejemplo, podemos realizar la siguiente inicialización:

```
int main()
{
   vector<Pareja> v { {'a',5}, {'b',9}, {'c',15} };
   // ...
}
```

donde podemos ver que las llaves especifican tres elementos —el tamaño de *v* será tres— y que cada elemento se especifica con dos componentes entre llaves.

Por otro lado, la asignación de valores a los distintos componentes se puede realizar en líneas que no corresponden a la declaración de un nuevo objeto. Por ejemplo, podemos modificar el valor de una estructura como sigue:

```
struct Trio {
   char c;
   int valor;
   double dato;
};

int main()
{
   Trio t= {'a', 1, 2.0};
```

```
    t= {'b', 3, 4.0};
    // ...
}
```

donde podemos ver que se modifica el valor de `t` indicando entre llaves los nuevos valores de cada uno de los componentes de la estructura `Trio`. Un ejemplo más complejo, con valores anidados, puede ser el siguiente:

```
// ...
vector<Pareja> devolver()
{
    return { {'a',1}, {'b',2}, {'c',3} };
}
int main()
{
     for (auto d: devolver() )
        cout << d.c << ' ' << d.valor << endl;
}
```

donde el vector de parejas se ha usado como contenedor que recorrerá un bucle basado en rango.

12.5.2 Tuplas

En la sección anterior hemos presentado algunos ejemplos con los tipos `Pareja` y `Trio`, donde sólo queríamos empaquetar dos o tres objetos de distintos tipos. Lógicamente, en programas donde los tipos fueran suficientemente relevantes, pondríamos un nombre más significativo y probablemente crearíamos otras herramientas para hacerlos más útiles. Sin embargo, hay casos en los que no es necesario crear una entidad destacada concreta, sino simplemente manejar una agrupación de distintos objetos como uno solo.

Por ejemplo, si queremos calcular la frecuencia de aparición de letras en un texto, basta con usar un contenedor de pares con la letra y la frecuencia, sin necesidad de especificar nada especial sobre cada par. Como sabemos, en este caso podría bastarnos con usar el tipo `pair<char,int>` (véase sección 8.3 en página 175).

En C++11 se generaliza esta agrupación y se ofrece un tipo que permite tener un número cualquiera de componentes: el tipo `tuple`, disponible si incluye `#include <tuple>`. Este tipo permite agrupar varios objetos especificando simplemente el tipo de cada uno de los componentes. Es una idea sencilla, pues es similar a lo que se hizo con el tipo `pair`. La sintaxis para declarar una tupla es la siguiente:

tuple<*tipo_1, tipo_2, ... , tipo_n*> *nombre_objeto*;

que declara un nuevo objeto como una tupla que tiene *n* campos, de tipos *tipo_1* hasta *tipo_n*.

El problema aquí es que una tupla no tiene un número fijo de componentes como en el caso de `pair`, sino que podemos crear cualquier número, incluso disponer de distintas tuplas en un mismo programa, cada una con un número distinto. Por tanto, el acceso a cada uno de los campos no se realiza con nombres predeterminados —como `first` y `second` en `pair`— sino que se lleva a cabo con un índice. La sintaxis para acceder a cada campo es:

get<*Indice*>(*nombre_objeto*)

La función `get` nos permite seleccionar un campo con un índice especificado entre los caracteres <>, el valor cero corresponderá al primer campo y el valor *n-1* al último. Esta función se puede usar tanto para consultar el valor del campo como para asignarlo si lo ponemos a la izquierda de una asignación.

Por ejemplo, imagine que queremos almacenar una tripleta de valores relacionada con las calificaciones obtenidas por un alumno. Para gestionarlo, necesitamos el nombre del estudiante, la lista de calificaciones parciales y una calificación final. Estos tres objetos podrían almacenarse en una tupla como sigue:

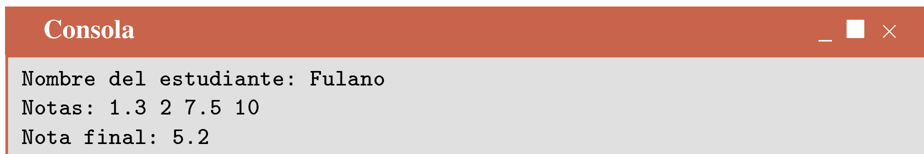
```
#include <tuple>
//...
tuple<string,vector<double>,double> datos {"Fulano", {1.3,2,7.5,10}, 5.2 };
```

donde además hemos usado la inicialización con llaves para indicar cada uno de los tres campos.

Para acceder a cada uno de los campos, podemos usar `get` con índices desde el cero hasta el 2. Por ejemplo, el siguiente trozo de código nos permite obtener en la salida estándar los valores de todos los campos:

```
cout << "Nombre del estudiante: " << get<0>(datos) << endl;

cout << "Notas: ";
for ( auto nota : get<1>(datos) )
   cout << nota << ' ';
cout << endl;

cout << "Nota final: " << get<2>(datos) << endl;
```

Con el siguiente resultado:



También podemos usar la función `get` para modificar el valor de uno de los campos. Por ejemplo, podemos modificarlos como sigue:

```
get<0>(datos) = "Mengano";
get<2>(datos) = Media (get<1>(datos));
```

donde la función `Media` podría implementarse como sigue:

```
double Media(const vector<double>& v)
{
   auto m= 0.0;
   for (auto dato: v)
      m+= dato;
   return m/v.size();
}
```

Finalmente, es interesante indicar que en el estándar C++14 también se puede acceder a cada uno de los campos si especificamos el tipo. Por ejemplo, en este ejemplo cada una de las tres posibilidades es un tipo distinto, por lo que podríamos indicar la selección sin conocer el orden de declaración. Por ejemplo, podríamos poner:

```
get<string>(datos) = "Mengano";
get<double>(datos) = Media (get<vector<double>>(datos));
```

lo que seguro le resulta mucho más fácil de escribir y de entender.

Creando literales con *make_tuple*

Para especificar un valor concreto de tipo `tuple` podemos usar la sintaxis que ya conoce en la que indicamos explícitamente el nombre del tipo con los parámetros que lo componen, ahora entre paréntesis o llaves. Por ejemplo, podemos asignar un valor a `trio` como sigue:

```
#include <tuple>
// ...
tuple<char,int,double> trio;

trio= tuple<char,int,double>{'a', 1, 2.0}; // Creamos objeto con tipo e inic.
```

Otra alternativa, mucho más sencilla, es usar la función **make_tuple**, muy similar a la función **make_pair** vista anteriormente. Esta función deduce el tipo de la tupla y devuelve el objeto creado a partir de los parámetros. Por ejemplo, podemos usarlo para asignar un nuevo valor a una tupla:

```
#include <tuple>
// ...
tuple<char,int,double> trio1{'a', 1, 2.0};   // Declaración con llaves
trio1= make_tuple('b', 3, 4.0);              // Asignamos un nuevo valor.
```

Finalmente, no podemos olvidarnos de que disponemos de la deducción de tipos, por lo que podríamos ahorrarnos especificar los tipos que el compilador puede deducir. Por ejemplo, sería válido hacer:

```
auto trio1= tuple<char,int,double>{'a', 1, 2.0}; // Se ve tipo a la derecha
auto trio2= make_tuple('c', 5, 6.0);   // make_tuple determina el tipo
```

12.5.3 Pares y tuplas

La similitud entre pares y tuplas es evidente. Los pares no son más que un caso particular de tuplas para los que se ofrecen los nombres `first` y `second` para acceder a sus campos. En el nuevo estándar, lógicamente, se homogeneizan las interfaces de forma que la interfaz de las tuplas también podrá usarse para los pares. Además, se hacen compatibles en el sentido de que un par se puede considerar una tupla de dos componentes, por lo que podrían mezclarse en el código: por ejemplo, si los dos campos son compatibles, se pueden asignar una a otra.

Mejora de la legibilidad con *tie*

Las utilidades de pares y tuplas constituyen una opción muy simple para trabajar con agrupaciones heterogéneas sin necesidad de crear un tipo específico para el problema. Si bien la simplicidad a la hora de declarar un nuevo tipo nos facilita la programación, la forma de acceder a los distintos componentes puede oscurecer el código. Es muy poco legible una línea que diga algo como *"el componente 3 de la tupla vale 3.0"*. Si definimos una estructura explícitamente para el problema, podríamos nombrar el componente 3 con un nombre significativo. Por ejemplo, en el código anterior donde guardábamos las calificaciones de un alumno, podemos crear la siguiente estructura:

```
struct Calificaciones {
   string nombre;
   vector<double> notas;
   double final;
};
```

Seguramente, en muchos casos esta solución le parezca mejor. Para los casos en los que decida usar un par o una tupla, el estándar ofrece la posibilidad de *"ligar"* los nombres de variables a cada uno de los campos, creando un objeto compuesto por esas variables. Por ejemplo, como primera solución, podemos implementar la siguiente función para mostrar las calificaciones:

```
void Mostrar (const tuple<string,vector<double>,double>& datos)
{
   cout << "Nombre del estudiante: " << get<0>(datos) << endl;
   cout << "Notas: ";
   for ( auto nota : get<1>(datos) )
      cout << nota << ' ';
   cout << endl;
   cout << "Nota final: " << get<2>(datos) << endl;
}
```

lo que hace un código menos legible, ya que está plagado de accesos `get<índice>` que no dejan claro a qué se refieren. Una alternativa es crear una tupla mediante la ligadura de variables independientes y a la que asignamos los datos a imprimir. Por ejemplo, de la siguiente forma:

```
void Mostrar (const tuple<string,vector<double>,double>& datos)
{
   string nombre;
   vector<double> notas;
   double final;
   tie(nombre,notas,final) = datos;

   cout << "Nombre del estudiante: " << nombre << endl;
   cout << "Notas: ";
   for ( auto nota : notas )
      cout << nota << ' ';
   cout << endl;
   cout << "Nota final: " << final << endl;
}
```

Además, para el caso en que no nos interese uno de los campos, el estándar nos permite usar el identificador `ignore` para que no ligue ninguna variable con ese campo. Por ejemplo, con el siguiente código:

```
vector<double> notas;
double final;
tie(ignore,notas,final) = datos;
```

podemos consultar los dos campos de notas e ignorar lo que contiene el primero.

A Solución a los ejercicios

Lenguajes de Programación 275
Introducción a C++ 275
La estructura de selección 278
La estructura de repetición 283
Funciones 290
Vectores de la STL 296
Cadenas de la STL 307
Estructuras y pares 311
Recursividad 316
Introducción a flujos de E/S 319

A.1 Lenguajes de Programación

Ejercicio 1.1 Calcular el valor del número hexadecimal *F3A.C9* en base decimal.

Solución al ejercicio 1.1 El valor del número se puede obtener con la siguiente expresión:

$$F \cdot 16^2 + 3 \cdot 16^1 + A \cdot 16^0 + C \cdot 16^{-1} + 9 \cdot 16^{-2}$$

Teniendo en cuenta que $F = 15$, $A = 10$, y $C = 12$, la respuesta al ejercicio es el resultado de:

$$15 \cdot 16^2 + 3 \cdot 16^1 + 10 \cdot 16^0 + 12 \cdot 16^{-1} + 9 \cdot 16^{-2}$$

Que resulta 3898.78515625.

Ejercicio 1.2 ¿Cuál es el rango de valores que puede tomar un entero que, con la representación anterior (1 bit de signo + resto de magnitud), usa dos bytes?

Solución al ejercicio 1.2 Si usamos dos bytes, disponemos de 15 bits para la magnitud. El mayor valor de un número binario de 15 bits corresponde a 111111111111111, o lo que es lo mismo, $2^{15} - 1$. Por tanto, el rango de valores será *[-32767,32767]*.

A.2 Introducción a C++

Ejercicio 2.1 Considere la siguiente lista de identificadores:

- *hola, primero, 1dato, datonúmero2, año, double, resultado, el_valor_5.*

Indique los identificadores que son válidos en C++.

Solución al ejercicio 2.1 Los identificadores son:

- *hola, primero*: Son válidos.
- *1dat*: No es válido, ya que comienza por un dígito.
- *datonúmero2*: No es válido, ya que contiene una vocal con tilde.
- *año*: No es válido, ya que contiene la letra *ñ*.
- *double*: No es válido, puesto que es una palabra reservada.
- *resultado, el_valor_5*: Son válidos.

Ejercicio 2.2 Identifique el tipo de los siguientes literales:
- *"X","Hola","",'Y',1345,30.0,true,15,"false",12.0,'\"'*

Solución al ejercicio 2.2 Los tipos de los literales son:
- *"X","Hola",""*: Cadenas de caracteres (de longitudes 1, 4 y 0 respectivamente).
- *'Y'*: Carácter.
- *1345*: Entero.
- *30.0*: Real. Aunque no tiene decimales, se ha especificado el punto decimal.
- *true*: Booleano.
- *15*: Entero.
- *"false"* : Cadena de caracteres.
- *12.0*: Real.
- '\"': Carácter.

Ejercicio 2.3 Supongamos las variables `a,b,c,d` de tipo **`double`**. Escriba en C++ las siguientes expresiones:

$$\frac{a+\frac{b}{cd}}{a+b}$$

$$a+\frac{b+c}{d}+\frac{b}{c}-b$$

Solución al ejercicio 2.3 Las expresiones son, respectivamente:
- `(a+b/(c*d))/(a+b)`
- `a+(b+c)/d+b/c-b`

Ejercicio 2.4 Suponga dos variables `x,y` que contienen dos valores desconocidos. Escriba las sentencias necesarias para hacer que los valores de esas variables se intercambien.

Solución al ejercicio 2.4 Para resolver el problema, es necesario declarar otra variable del mismo tipo (por ejemplo, de nombre `aux`). Con esta tercera variable, podemos realizar el intercambio de valores con las líneas:

```
aux= x;
x= y;
y= aux;
```

Observe que la variable auxiliar es necesaria para salvar el valor original de `x`, que se pierde al asignarle el valor de `y` (segunda línea).

Ejercicio 2.5 Indique el resultado de la ejecución del siguiente trozo de código:

```
double f;
int e;
```

```
f= 9;
e= 2;
cout << f/e      << ' ' << 9/2       << ' ' << 9/2.0 << ' '
     << f/e*1.0 << ' ' << 9/(e*1.0) << ' ' << 1.0*9/e <<
     << ' ' << 9/e*1.0;
```

Solución al ejercicio 2.5 Para resolver este problema, es importante tener en cuenta los tipos de los datos involucrados en la operación. Debemos prestar especial atención a la división de dos número enteros cuyo resultado es también de tipo entero. El resultado es:

Consola

```
4.5 4 4.5 4.5 4.5 4.5 4
```

Es importante que observe que la "mezcla" de un número real y otro entero implica una operación entre reales. Por otro lado, recordemos que las operaciones de multiplicación y división tienen la misma prioridad, resolviéndose de izquierda a derecha.

Ejercicio 2.6 Escribir un programa que lea —desde la entrada estándar— la base y altura de un rectángulo, y escriba —en la salida estándar— su área y perímetro.

Solución al ejercicio 2.6 El programa que resuelve este problema es:

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

int main()
{
   double base,altura;

   cout << "Introduzca la base y altura: ";
   cin >> base >> altura;
   cout << "Área: " << base*altura << endl
        << "Perímetro: " << 2*(base+altura) << endl;
}
```

Ejercicio 2.7 Escriba un programa que lea el valor de dos puntos en el plano, y escriba —en la salida estándar— la distancia entre ellos. Recuerde que ésta se calcula como:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Solución al ejercicio 2.7 Una posible solución al problema es la siguiente:

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

int main()
{
   double x1, y1, x2, y2;

   cout << "Introduzca dos valores (x,y) para el primer punto: ";
   cin >> x1 >> y2;
   cout << "Introduzca dos valores (x,y) para el segundo punto: ";
   cin >> x2 >> y2;
   cout << "La distancia entre ellos es: " <<
         sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) ) << endl;
}
```

Observe que no hemos usado la función **pow** para elevar al cuadrado las diferencias. Aunque también sería correcto, el problema es más fácil[1] de resolver con una simple multiplicación.

A.3 La estructura de selección

Ejercicio 3.1 Escribir un programa que lea dos valores reales y escriba en la salida estándar un mensaje indicando cuál es el mayor.

Solución al ejercicio 3.1 Una posible solución es la siguiente:

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

int main()
{
   double dato1, dato2;

   cout << "Introduzca dos números:" << endl;
   cin >> dato1 >> dato2;
   if (dato1>dato2)
      cout << "El primero es mayor" << endl;
   if (dato1<dato2)
      cout << "El segundo es mayor" << endl;
   if (dato1==dato2)
      cout << "Son iguales" << endl;
}
```

Ejercicio 3.2 Escribir un programa que lea un valor entero y escriba en la salida estándar un mensaje indicando si es par o impar.

Solución al ejercicio 3.2 Una posible solución es la siguiente:

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

int main()
{
   int n,resto;

   resto= n%2;
   if (resto==0)
      cout << "El número es par" << endl;
   if (resto!=0)
      cout << "El número es impar" << endl;
}
```

Ejercicio 3.3 Considere el ejemplo anterior, eliminando las dos llaves de la sentencia **if**. Si el usuario ejecuta el programa con el dato *0*, ¿qué salida obtendrá?

Solución al ejercicio 3.3 Al eliminar las dos llaves, el compilador entiende que la línea (11) corresponde a la sentencia que hay que ejecutar en caso de que la condición sea **true**. La línea 12 es, por tanto, la sentencia que sigue a la instrucción **if**.

Cuando se da el dato *0*, la condición del primer **if** es falsa, y por tanto no se ejecuta la sentencia asociada, pasando a la siguiente instrucción, es decir, a la línea 12.

La salida que se obtiene es:

[1] De hecho, es de esperar que una multiplicación sea más eficiente que la función **pow**. Recuerde que esta última está diseñada para cualquier número, incluso si no es entero.

Consola

```
Es un alumno muy obediente
Ups! que desobediente...
```

Ejercicio 3.4 Escriba un programa que lea un número real, correspondiente al radio de un círculo. Como resultado, escribirá el radio introducido, el área del círculo, y la longitud de su perímetro. Además, comprobará si el radio no es positivo, en cuyo caso se obtendrá un mensaje informando sobre ello.

Solución al ejercicio 3.4 Una posible solución es la siguiente:

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

int main()
{
   double radio;

   cout << "Introduzca radio del círculo: " << endl;
   cin >> radio;
   if (radio<=0)
      cout << "El radio no es positivo" << endl;
   if (radio>0) {
      cout << "Radio: " << radio << endl;
      cout << "Área: " << 3.14159*radio*radio << endl;
      cout << "Perímetro: " << 2*3.14159*radio << endl;
   }
}
```

Ejercicio 3.5 Escribir un programa que lea tres números reales que corresponden a una ecuación de segundo grado. En caso de que tenga solución real, la escribirá en la salida estándar. Compruebe para ello si el discriminante es no negativo.

Solución al ejercicio 3.5 Recuerde que la solución de la ecuación $ax^2+bx+c=0$ viene dada por la ecuación:

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

Por tanto, una posible solución es la siguiente:

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

int main()
{
   double a, b, c, discriminante;

   cout << "Introduzca (a,b,c) de ax^2+bx+c=0: " << endl;
   cin >> a >> b >> c;
   discriminante= b*b - 4*a*c;
   if (discriminante >=0 ) {
      cout << "Solución 1: "
           << (-b+sqrt(discriminante))/(2*a)<< endl;
      cout << "Solución 2: "
           << (-b-sqrt(discriminante))/(2*a)<< endl;
   }
}
```

Observe que también hubiera sido válido incluir la expresión dentro de la condición:

```
// ...
  if (b*b-4*a*c >= 0) {
// ...
```

Ejercicio 3.6 Escriba un programa que lea tres números reales correspondientes a una ecuación de segundo grado y escriba en la salida estándar las soluciones —si las hay— o un mensaje indicando que no las hay. Compruebe para ello si el discriminante es no negativo, usando una sentencia `if/else` para distinguir los dos casos.

Solución al ejercicio 3.6 La solución es similar al ejercicio anterior, añadiendo la parte `else` a la sentencia condicional. Concretamente:

```
// ...
   discriminante= b*b - 4*a*c;
   if (discriminante >= 0) {
      cout << "Solución 1: " << (-b+sqrt(discriminante))/(2*a)<< endl;
      cout << "Solución 2: " << (-b-sqrt(discriminante))/(2*a)<< endl;
   }
   else cout << "La ecuación no tiene solución real" << endl;
}
```

Ejercicio 3.7 Escriba un programa que lea tres números reales, calcule el máximo, y finalmente lo escriba en la salida estándar.

Solución al ejercicio 3.7 Una posible solución es la siguiente:

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

int main()
{
   double a, b, c, maximo;

   cout << "Introduzca tres números reales: " << endl;
   cin >> a >> b >> c;

   if (a>b)
      if (a>c)
         maximo= a;
      else maximo= c;
   else if (b>c)
           maximo= b;
        else maximo= c;

   cout << "El máximo es: " << maximo << endl;
}
```

Este código, aunque ilustrativo para el anidamiento, se puede simplificar de la siguiente forma:

```
// ...
maximo= a;
if (b>maximo)
  maximo= b;
if (c>maximo)
  maximo= c;
```

Ejercicio 3.8 Escriba un programa que lea tres números reales correspondientes a una ecuación de segundo grado y escriba en la salida estándar las soluciones. Para ello, debe comprobar que el valor del coeficiente de grado dos es distinto de cero y que el discriminante es no negativo. En estos casos, en lugar de las soluciones, escribirá el mensaje correspondiente en la salida estándar.

Solución al ejercicio 3.8 Una posible solución es la siguiente:

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

int main()
{
   double a, b, c, discriminante;

   cout << "Introduzca (a,b,c) de ax^2+bx+c=0: " << endl;
   cin >> a >> b >> c;
   discriminante= b*b - 4*a*c;
   if (a==0)
      cout << "La ecuación no es de segundo grado" << endl;
   else if (discriminante>=0) {
           cout << "Solución 1: "
                << (-b+sqrt(discriminante))/(2*a) << endl;
           cout <<"Solución 2: "
                << (-b-sqrt(discriminante))/(2*a) << endl;
        }
        else cout << "La ecuación no tiene solución real" << endl;
}
```

Ejercicio 3.9 Escriba un programa que lea un número real correspondiente a una nota, y escriba en la salida estándar el mensaje "Suspenso" (menor que 5), "Aprobado" (de 5 a 7), "Notable" (de 7 a 9) o "Sobresaliente" (9 o más) dependiendo del valor de la nota.

Solución al ejercicio 3.9 La solución consiste en leer un valor real, e ir comprobando en qué intervalo se encuentra para escribir el mensaje correspondiente. No comprobaremos si la nota está en el intervalo *[0,10]* (véanse las siguientes secciones del tema). De esta forma, una posible solución es la siguiente:

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

int main()
{
   double nota;

   cout << "Introduzca la nota: " << endl;
   cin >> nota;
   if (nota<5)
      cout << "Suspenso" << endl;
   else if (nota<7)
           cout << "Aprobado" << endl;
        else if (nota<9)
                cout << "Notable" << endl;
             else cout << "Sobresaliente" << endl;
}
```

Ejercicio 3.10 Escribir las expresiones booleanas que corresponden a las siguientes condiciones:

1. El valor del entero A está en $\{1,2,3,4,5,6\}$
2. El valor del real Aestá en $[-\infty,-5]\cup[5,\infty]$
3. El valor del real A no está en $[-5,3]$
4. El valor del real A está en $\{1,2,3\}\cup[10,20[$
5. El valor del entero A es par o un valor impar entre cero y diez.
6. El valor del real A está en $[0,10]\cup[20,30]$
7. El valor del real A no está en $[0,10[\cup]20,30]$

Solución al ejercicio 3.10 Puede haber varias formas para expresar una misma condición. Aquí presentamos sólo algunas de ellas. Las expresiones pueden ser:

1. `A>=1 && A<=5`. También: `A>0 && A<6`
2. `A<=-5 || A>=5`
3. `A<-5 || A>3`
4. `A==1 || A==2 || A==3 || (A>=10 && A<20)`
5. `A%2==0 || (A%2!=0 && A>0 && A<10)`
6. `(A>=0 && A<=10)|| (A>=20 && A<=30)`
7. `!((A>=0 && A<10)|| (A>20 && A<=30))`
   También: `(A<0 || (A>=10 && A<=20)|| A>30)`

Ejercicio 3.11 Reescriba el formato general de la sentencia **switch** con la misma notación, pero haciendo uso de sentencias **if/else**.

Solución al ejercicio 3.11 Para resolverlo, tendremos que incluir una sentencia **if** para comprobar cada uno de los casos de la sentencia **switch**. En el último **if**, si no se ha localizado ningún caso, añadimos una parte **else** con las sentencias que corresponden a **default**. La solución es:

if (*expresión==valor1*)
    sentencias1;
else if (*expresión==valor2*)
    sentencias1;
    ...
    else *sentencias;*

Ejercicio 3.12 Escriba un programa que, después de pedir dos números, presente al usuario un conjunto de 4 opciones (suma, resta, producto, división). Una vez que el usuario seleccione la deseada, el programa presentará el resultado de la operación correspondiente.

Solución al ejercicio 3.12 El programa comienza solicitando dos valores y presentando un menú de 4 opciones. Esta parte puede resolverse como sigue:

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

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

   cout << "Introduzca dos valores: ";
   cin >> dato1 >> dato2;
   cout << "1.- Suma" << endl;
   cout << "2.- Resta" << endl;
   cout << "3.- Producto" << endl;
   cout << "4.- División" << endl;
   cout << "Introduzca opción: ";
   cin >> opcion;

   // ...
}
```

Una vez que tenemos los datos, usamos **switch** para realizar la operación correcta. El resto del programa puede ser como sigue:

```
// ...
   switch (opcion) {
      case '1':
         cout << "Suma: " << dato1+dato2 << endl;
         break;
```

```
            case '2':
               cout << "Resta: " << dato1-dato2 << endl;
               break;
            case '3':
               cout << "Producto: " << dato1*dato2 << endl;
               break;
            case '4':
               cout << "División: " << dato1/dato2 << endl;
               break;
            default:
               cout << "No se ha indicado ninguna opción" << endl;
         }
   }
```

Observe que los casos son literales de tipo `char` (entre comillas simples), ya que la opción se ha declarado de este tipo. También podríamos haber usado el tipo `int`, en cuyo caso, deberíamos escribir los valores enteros en lugar de los caracteres.

A.4 La estructura de repetición

Ejercicio 4.1 Para calcular la parte entera del logaritmo en base 2 de un número entero positivo, se ha decidido implementar un algoritmo iterativo. La idea es que podemos ir dividiendo el número de forma sucesiva por 2 hasta que lleguemos a 1. El número de divisiones realizadas es el valor buscado. Escriba el algoritmo en C++.

Solución al ejercicio 4.1 Para resolver el problema, usamos un bucle `while` que, en cada iteración, divide el número entre 2 y aumenta en 1 un contador. La solución puede ser:

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

int main()
{
   int n, i;

   cout << "Introduzca un número entero positivo: ";
   cin >> n;
   i= 0;
   while (n>1) {
      n= n/2;
      i= i+1;
   }
   cout << "Resultado: " << i << endl;
}
```

Observe que n se divide entre 2 despreciando la parte decimal, ya que se realiza una división entera.

Ejercicio 4.2 Escriba un programa para determinar si un número entero es un cuadrado perfecto. Recuerde que estos números son de la forma n^2, es decir, los número 1,4,9,16,etc. Para ello, no podrá hacer uso de la función `sqrt`, sino que tendrá que ir calculando los cuadrados de los números desde el 1 en adelante.

Solución al ejercicio 4.2 La solución que proponemos es usar un bucle `while` para incrementar el valor de un contador mientras que su cuadrado sea menor que el número. La solución puede ser:

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

int main()
{
   int n, i;
```

```
    cout << "Introduzca un entero positivo: ";
    cin >> n;
    i= 0;
    while (i*i<n)
       i= i+1;

    if (i*i==n)
       cout << "Es el cuadrado de "<< i << endl;
    else cout << "No es cuadrado perfecto" << endl;
}
```

Por supuesto, resulta una solución muy ineficiente ya que, incluso si deseamos una solución basada en la búsqueda del número `i` que mejor aproxima la raíz cuadrada, es posible aplicar algoritmos más eficientes.

Ejercicio 4.3 Escriba un programa que lea 100 valores e imprima el máximo y el mínimo en la salida estándar.

Solución al ejercicio 4.3 La solución del problema consiste en un bucle `for` en el que se pide un valor desde la entrada estándar y, si es necesario, se actualizan dos variables auxiliares que almacenan el máximo y mínimo, respectivamente. La solución podría ser:

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    double maximo, minimo;

    cin >> valor;
    maximo= valor;
    minimo= valor;
    for (i=1; i<=99; i=i+1) {
       cin >> valor;
       if (valor>maximo) maximo= valor;
       if (valor<minimo) minimo= valor;
    }
    cout << "La suma total es:" << suma << endl;
}
```

Observe que hemos leído un primer valor fuera del bucle para inicializar los valores de máximo y mínimo antes del bucle. Otra solución podría haber sido inicializar el máximo a un valor muy pequeño y el mínimo a un valor muy grande para garantizar que se actualizan con el primer valor introducido.

Ejercicio 4.4 Indique el resultado de ejecutar el siguiente trozo de código:

```
1 int a= 2, b= 3, c= 5;
2 cout << ++a + ++b + ++c << endl;
3 a--; --b; --c;
4 cout << a++ + --b * c++ << endl;
5 cout << a << ' '<< b << ' '<< c << endl;
```

Solución al ejercicio 4.4 Recuerde que el preincremento y predecremento se realizan antes de "usar" la variable. Así, en la línea 2, los tres incrementos son previos, y por tanto, primero se incrementan en uno y luego se suman. El resultado es *13*.

En la línea 3, los decrementos se realizan sobre la variable, aunque no se usa el valor de ésta. Por tanto, el efecto es simplemente que las variables decrementan su valor en uno.

En la línea 4, se decrementa el valor de `b` antes de usarse, y se incrementan `a,c` después de usarse. Por tanto, el resultado es la expresión *2+2*5*, es decir, *12*.

Finalmente, la última línea escribe los valores finales de `a,b,c`, que son, respectivamente *3,2* y *6*.

Ejercicio 4.5 Indique el comportamiento del programa cuando introducimos el número 1 como dato a analizar.

Solución al ejercicio 4.5 Cuando introducimos el numero *1*, la línea 12 obtiene como raíz también el valor *1*. Al llegar al bucle —línea 14— se inicializa la variable `i` con 2 y se pasa a evaluar la condición del bucle, antes de la primera iteración. Dado que la condición es falsa, el bucle no itera ninguna vez, y se pasa directamente a la línea 18. Finalmente, se escribe que es un número primo, ya que no se ha cambiado el valor de la variable `esprimo` desde que se inicializó en la línea 13.

Ejercicio 4.6 Cuál es el efecto si modificamos el trozo final por:

```
if (esprimo)
   cout << "El número es primo" << endl;
else cout << "El número no es primo (divisor:"
          << i << ")" << endl;
```

Solución al ejercicio 4.6 La modificación consiste en escribir el último valor de `i` para indicar el divisor que "rompe" la primalidad. En principio, puede parecer que es correcto, pero recordemos que el incremento se realiza al final del cuerpo, y antes de evaluar la condición. De este modo, cuando localizamos un divisor `i` (cambiamos `esprimo`), incrementamos el valor de `i` antes de evaluar la condición (que será falsa, ya que hemos localizado un divisor). Por tanto, el valor que se escribe en la última línea es el siguiente al divisor, ya que antes de que la condición terminara el bucle, se acumuló el último incremento.

Ejercicio 4.7 El programa anterior aún se puede mejorar si tenemos en cuenta un par de condiciones más:

1. El único entero primo y par es el dos.
2. Un entero n mayor que 3, sólo puede ser primo si verifica $n^2 \mod 24 = 1$.

Modifique el programa anterior para incorporar estas propiedades.

Solución al ejercicio 4.7 La primera propiedad se puede usar para el bucle, ya que no tenemos que comprobar todos los valores a partir de 2, sino únicamente los impares.

La segunda propiedad indica, que si no se cumple $n^2 \mod 24 = 1$, el número no puede ser primo.

Podemos escribir el siguiente programa:

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  int main()
5  {
6     int numero, raiz, i;
7     bool esprimo;
8
9     cout << "Introduzca el número" << endl;
10    cin >> numero;
11
12    esprimo= true;
13    if (numero>3)
14       raiz= sqrt(numero);
15       esprimo= numero*numero %24 == 1;
16       for (i=3; i<=raiz && esprimo; i+=2)
17          if (numero%i==0) // Es divisible
18             esprimo= false;
19    }
20
21    if (esprimo)
22       cout << "El número es primo" << endl;
```

```
23     else cout << "El número no es primo" << endl;
24  }
```

Observe que en la línea 15 hemos inicializado el valor de `esprimo` con la expresión booleana $n^2 \mod 24 = 1$. Si se cumple esta condición, el valor será **`true`**, y el bucle tendrá que comprobar si es primo o no. Si no se cumple sabemos que el número no puede ser primo, el valor de `esprimo` se hace **`false`**, y por tanto el bucle no iterará ninguna vez.

Ejercicio 4.8 Escriba un programa para resolver una ecuación de primer grado ($ax+b=0$). Tenga en cuenta que se deberá repetir la lectura de a hasta que sea distinto de cero.

Solución al ejercicio 4.8 Una posible solución es la siguiente:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double a, b;

   cout << "Introduzca a:";
   cin >> a;
   while (a==0) {
      cout << "Debe ser distinto de cero:"
      cin >> a;
   }
   cout << "Introduzca b:";
   cin >> b;
   cout << "Solución: " << −b/a << endl;
}
```

Ejercicio 4.9 Escriba un programa que lea una secuencia de números reales e imprima la media. Realice dos versiones:

1. Un programa para leer 100 números reales.
2. Un programa que lee 100 números o hasta que se introduce un valor negativo. El valor negativo no se deberá incluir en la suma.

Solución al ejercicio 4.9 El primer problema es muy simple, ya que es parecido a alguno de los ejemplos que hemos estudiado. El programa podría ser:

```
#include <iostream>
using namespace std;
int main()
{
   int i;
   double x, suma;

   for (i=1, suma=0; i<=100; ++i) {
      cout << "Introduzca valor:";
      cin >> x;
      suma+= x;
   }

   cout << "Media: " << suma/100 << endl;
}
```

El segundo programa es similar al caso anterior, pero no tiene que iterar 100 veces, sino que puede parar antes en caso de que se introduzca un valor negativo.

Observe que si introducimos un valor negativo, no puede ser incluido en la suma. Por lo tanto, podemos cambiar el bucle para que se haga una lectura adelantada. La idea sería:

```
cin >> x;
while (x>=0){
   // ...
```

```
    cin >> x;
}
```

Sin embargo, la condición de parada es más compleja, ya que parará cuando termine con el valor 100 o cuando haya introducido un dato negativo. Por lo tanto, necesitamos una condición compuesta para incluir también un contador hasta 100 valores. La idea sería:

```
cin >> x;
for (i=1; i<=100 && x>=0; ++i)
    // ...
    cin >> x;
}
```

El problema de esta solución es que lee 101 elementos. Mezclar las dos soluciones, con una lectura adelantada, no parece una buena idea. La solución podría ser modificar el comportamiento del bucle **for** con una variable booleana:

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    double x, suma= 0;
    bool fin= false;

    for (i=1; i<=100 && !fin; ++i)
        cout <<"Introduzca valor:";
        cin >> x;
        if (x>=0)
            suma+= x;
        else fin= true;
    }
    cout << "Media: " << suma/(i-1) << endl;
}
```

donde puede ver que la media se realiza dividiendo por `i-1`, ya que el valor final de `i` es uno más que el número de valores acumulados. Sin embargo, esto sólo es válido si hemos leído los 100 elementos, ya que en la última iteración con un valor negativo no se acumula nada.

Tal vez se le ocurra que la solución pasa por eliminar el último valor acumulado de `i`. El código sería:

```
    for (i=1; i<=100 && !fin; ++i)
        cout <<"Introduzca valor:";
        cin >> x;
        if (x>=0)
            suma+= x;
        else {
            fin= true;
            --i;
        }
    }
    if (i>1) // Debe haber al menos un sumando
      cout << "Media: " << suma/(i-1) << endl;

}
```

donde también hemos añadido una condición por si el primer elemento es negativo.

Este es un buen momento para reflexionar sobre este código. Observe la cantidad de líneas que contienen la variable `i` y la estrecha relación que hay entre ellas. Parece muy fácil equivocarse y se complica la tarea de incorporar modificaciones. Puede dejar este código a un amigo para que lo interprete, seguro que necesitará un buen rato para confirmar el porqué de algunas líneas.

En general, no es una buena idea modificar la variable contadora dentro del bucle o usarla después de haber terminado.

Para resolver este problema, vamos a cambiar la implementación para evitar este mal uso. Incorporamos una variable `n` para contar el número de sumandos:

```
#include <iostream>
using namespace std;
int main()
{
   int i, n= 0;
   double x, suma= 0;
   bool fin= false;

   for (i=1; i<=100 && !fin; ++i)
      cout <<"Introduzca valor:";
      cin >> x;
      if (x>=0) {
         suma+= x;
         n= n+1;
      }
      else fin= true;
   }
   if (n>0) // Debe haber al menos un dato
      cout << "Media: " << suma/n << endl;
}
```

Observe que el uso de `i` se limita a controlar el contador del bucle. Seguro que esta versión le parece mucho más simple y más fácil de manejar.

Ejercicio 4.10 Escriba un programa que obtenga en pantalla las 10 tablas de multiplicar (del 1 al 10), con 10 entradas cada una (del 1 al 10).

Solución al ejercicio 4.10 Para resolverlo, anidamos dos bucles `for`. El primero controla la tabla a escribir y el segundo cada entrada de la tabla. Una posible solución es la siguiente:

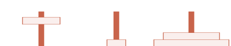
```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   int t, i;

   for (t=1; t<=10; ++t) { // Cada tabla
      cout << "Tabla del " << t << endl;
      for (i=1; i<=10; ++i)
         cout << t << 'x' << i << '=' << t*i << endl;
   }
}
```

Observe que el cuerpo del primer bucle está entre llaves, ya que tiene más de una sentencia.

Ejercicio 4.11 Escriba un programa que lea un número de la entrada estándar y escriba su tabla de multiplicar. Al finalizar, preguntará si se desea escribir una nueva tabla, si se responde *'S'* repetirá el proceso y si se responde *'N'* finalizará.

Solución al ejercicio 4.11 El programa presenta al menos una tabla y repite esta operación mientras que la opción sea *'S'*. Para resolverlo, podemos usar un bucle `do`/`while`, ya que la operación se realiza al menos una vez. Una solución puede ser:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   int t, i;
   char opcion;

   do {
      cout << "Introduzca número: ";
      cin >> t;
      for (i=1; i<=10; ++i)
         cout << t << 'x' << i << '=' << t*i << endl;
```



```
        cout << "¿Desea otra?";
        cin >> opcion;
    } while (opcion=='S');
}
```

Ejercicio 4.12 Modifique el ejemplo anterior para obtener una solución más eficiente.

Solución al ejercicio 4.12 El problema es muy sencillo de resolver, ya que sólo es necesario calcular una aproximación con 100 sumandos del valor de π, escribiendo en cada uno de los pasos de acumulación la solución hasta ese momento. El código podría ser el siguiente:

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

int main()
{
   double valor;
   double acumulado;
   int n;

   acumulado= 0;
   for (n=0; n<100; n++) {
      valor= 1.0/( (2*n+1) * (2*n+1) * (2*n+1) );
      acumulado+= (n%2==0) ? valor : -valor;
      cout << "Valor con " << n+1 << " sumandos: "
           << pow(acumulado*32,1.0/3) << endl;
   }
}
```

Ejercicio 4.13 Se quiere comprobar si la expresión

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

obtiene con pocos sumandos una aproximación aceptable (por ejemplo, 4 decimales) del valor de e. Escriba un programa que pida el número de sumandos y escriba el resultado del cálculo.

Solución al ejercicio 4.13 Para resolverlo, usamos un bucle para calcular cada sumando. Para calcular un sumando, necesitamos otro bucle, ya que tenemos que obtener un factorial. La solución puede ser[2]:

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  int main()
5  {
6     int sumandos, n, i, factorial;
7     double e;
8
9     cout << "Indique número de sumandos: ";
10    cin >> sumandos;
11
12    e= 0.0; // Importante para acumular una suma
13    for (n=0; n<sumandos; ++n) {
14       factorial= 1;
15       for (i=2; i<=n; ++i)
16          factorial*= i;
17       e+= 1.0/factorial;
18    }
19
```

[2]El compilador tiene una precisión por defecto en la escritura, que tal vez no sea suficiente. Aquí no vamos a entrar en problemas de formato de salida de un número real, aunque se podría indicar un número determinado de decimales (por ejemplo, con `setprecision`).

```
20     cout << "Aproximación: " << e << endl;
21  }
```

Observe que la división de la línea 17 se realiza con el literal `1.0`, ya que debe ser de números reales. Otra solución hubiera sido declarar *factorial* de tipo **double**.

Finalmente, es importante indicar que la solución propuesta muestra el uso de dos bucles anidados. Sin embargo, se podría resolver sin el bucle interno, ya que el factorial para un determinado *n* se puede obtener a partir del factorial del anterior sumando.

A.5 Funciones

Ejercicio 5.1 Implemente una función *EsVocal* que devuelva si un carácter es una vocal o no.

Solución al ejercicio 5.1 La función recibe como entrada un dato de tipo **char**, y devuelve como salida un dato de tipo **bool**. La cabecera de la función puede ser, por tanto, la siguiente:

```
bool EsVocal (char c)
```

donde hemos identificado el dato de entrada —parámetro— con el nombre *c* para poder referenciarlo en el cuerpo de la función. La función completa puede implementarse como sigue:

```
bool EsVocal (char c)
{
   return (c=='a' || c=='e' || c=='i' || c=='o' || c=='u');
```

La expresión que se devuelve es de tipo booleano, al igual que la función. Note que no se han incluido otras posibilidades, como las vocales con tilde. Si queremos que éstas sean también identificadas como vocales, tendremos que añadirlas al listado de las 5 anteriores.

Ejercicio 5.2 Considere el programa de la página 62, donde se calcula el máximo común divisor de dos enteros. Reescriba el programa para incluir una función *MCD* que tenga dos parámetros y devuelva el máximo común divisor de ellos.

Solución al ejercicio 5.2 El programa puede ser el siguiente:

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

int MCD (int a, int b)
{
   int resto;

   resto= a%b;
   while (resto!=0) {
      a= b;
      b= resto;
      resto= a%b;
   }
   return b;
}

int main()
{
   int a, b;

   cout << "Introduzca dos enteros positivos: ";
   cin >> a >> b;
   cout << "El m.c.d. es: " << MCD(a,b) << endl;
}
```

Ejercicio 5.3 Escriba un programa que lea un valor entero desde la entrada estándar y escriba el factorial en la salida estándar. Realice la modularización que considere más adecuada.

Solución al ejercicio 5.3 Para resolver el problema podemos incluir dos funciones:

1. `LeerIntNoNegativo`. Para leer un entero mayor o igual a cero. La solución puede ser:

```
int LeerIntNoNegativo()
{
   int valor;
   cout << "Introduzca entero positivo:";
   cin >> valor;
   while (valor<0) {
      cout << "El valor es negativo. Introduzca otro:";
      cin >> valor;
   }
   return valor;
}
```

2. `Factorial`. Para calcular el factorial de un entero. La solución puede ser:

```
int Factorial (int n)
{
   int resultado= 1;
   for (int i=n; i>1; --i)
      resultado*= i;
   return resultado;
}
```

Una vez disponemos de estas dos funciones, podemos escribir el programa de la siguiente forma:

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

int LeerIntNoNegativo()
{
   // ...
}
int Factorial (int n)
{
   // ...
}
int main()
{
   int n;
   n= LeerIntNoNegativo();
   cout << "Factorial: " << Factorial(n) << endl;
}
```

donde los puntos suspensivos deben sustituirse por el cuerpo de las funciones que acabamos de ver.

Otra posible versión de la función **main** algo menos legible es:

```
// ...
int main()
{
   cout << "Factorial: "
        << Factorial( LeerIntNoNegativo() )
        << endl;
}
```

Ejercicio 5.4 Modifique el programa anterior para que resuelva la ecuación $5x^3 + 3x - 1 = 0$.

Solución al ejercicio 5.4 El hecho de encapsular el cálculo de la función en un módulo permite modificar el programa fácilmente, ya que sólo tendremos que cambiar esa parte para obtener la nueva versión. Concretamente, podemos modificar la siguiente función:

```
double FuncionF(double x)
{
   return 5*x*x*x + 3*x - 1; // 5x^3+3x-1
}
```

Ejercicio 5.5 Analice la condición de parada del bucle en la función *Biseccion*. Realmente, damos una iteración de más, puesto que escoger el valor central garantiza que la precisión es la mitad del tamaño del intervalo. Modifique el código para ahorrarnos esta iteración.

Solución al ejercicio 5.5 La solución más simple es duplicar el valor de la precisión antes de entrar al bucle:

```
double Biseccion (double izq, double der, double prec)
{
   prec= prec*2;

   while (der-izq>prec) {
      double centro= (der+izq)/2;
      if (FuncionF(izq)*FuncionF(centro)<=0)
         der= centro;
      else izq= centro;
   }
   return (der+izq)/2;
}
```

Este ejemplo resulta interesante, primero porque demuestra cómo una modificación se puede realizar sin pensar en los detalles del resto de módulos y segundo porque podemos ver que el parámetro *prec* se puede modificar sin problemas, pues no afecta al argumento de llamada.

Ejercicio 5.6 Modifique la función anterior para que también garantice que el valor de la función en los extremos del intervalo tiene distinto signo.

Solución al ejercicio 5.6 Para resolverlo, sólo tendremos que modificar la condición del bucle de la siguiente forma:

```
void LeerIntervalo (double& izq, double& der)
{
   cout << "Introduzca los dos valores del intervalo: ";
   cin >> izq >> der;
   while (izq > der || FuncionF(izq)*FuncionF(der)>0) {
      cout << "No es correcto. Nuevo Intervalo: ";
      cin >> izq >> der;
   }
}
```

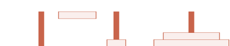
Ejercicio 5.7 Desarrolle una función que tenga como entrada dos enteros correspondientes a una fracción —numerador y denominador— y que como salida modifique sus valores para hacerla irreducible. Nota: Recuerde que para simplificar la fracción, se dividen ambos por el máximo común divisor (una función que ya tenemos resuelta).

Solución al ejercicio 5.7 La solución puede ser la siguiente:

```
void Irreducible (int& a, int& b)
{
   int mcd= MCD(abs(a), abs(b));
   a/= mcd;
   b/= mcd;
}
```

Observe que no hemos comprobado nada acerca del valor inicial de los parámetros. La función supone que *b* no es cero.

Cualquier programa que use esta función deberá incluir también *MCD*, así como la cabecera **cstdlib** para usar la función *abs* del estándar (obtiene el valor absoluto de un entero).



Ejercicio 5.8 ¿Cual sería el efecto de hacer la llamada como `Ecuacion2Grado(a,b,c, a,b)`?

Solución al ejercicio 5.8 En esta llamada hemos pasado `a,b` como parámetros donde almacenar el resultado. Por tanto, cuando modifiquemos uno de estos parámetros en la función (`x1` o `x2`), estaremos modificando `a,b`.

En principio, esta modificación puede hacer pensar que la función no va a obtener los valores correctos, ya que tiene que usar `a,b` como los dos primeros parámetros. Sin embargo, los valores de los tres primeros parámetros de la función no se verán modificados, ya que el paso se realiza por valor.

Por tanto, en el momento de la llamada se crean tres variables `a,b,c` que son copia de los tres primeros argumentos. Una vez dentro, la modificación de los dos últimos parámetros no afectará a los primeros, y el resultado final se obtendrá, sin ningún problema, en las variables `a,b`.

Por supuesto, este tipo de llamadas no son recomendables, ya que genera demasiado confusión y hace el programa difícil de leer.

Ejercicio 5.9 En los algoritmos de cálculo numérico resulta "delicada" la comparación de números reales con un valor concreto (véase problema 2.6). En el ejemplo anterior, realizamos algunas operaciones dependiendo de si un valor real vale cero. Sin embargo, en la práctica los cálculos de número reales son aproximados ya que no podemos representar todos los valores[a]. Modifique la función `Ecuacion2Grado` para incorporar esta posibilidad. Considere que un valor es igual a otro si su diferencia es menor que un cierto épsilon.

[a] Este problema resulta fundamental en muchos algoritmos. Especialmente, cuando un valor se obtiene tras muchas operaciones en las que se van acumulando los errores de aproximación.

Solución al ejercicio 5.9 Resolvemos el problema creando una nueva función `Iguales` que devuelva si dos valores reales son iguales. Puede ser la siguiente:

```
bool Iguales(double x1, double x2)
{
   const double EPSILON= 10e-20;
   return fabs(x1-x2)<EPSILON;
}
```

Para usarla en nuestro programa sólo debemos modificar las comparaciones, sustituyéndolas por las llamadas correspondientes:

```
//...
bool Ecuacion2Grado (double a, double b, double c,
                     double& x1, double& x2)
{
   bool exito;

   if (Iguales(a,0))
      if (Iguales(b,0))
         exito= false; // No hay x

   // ...
```

Es interesante destacar que el problema de la representación aproximada de los números reales es muy importante en aplicaciones de cálculo numérico. La solución puede ser más compleja, dependiendo del tipo de aplicación que se esté implementando.

El hecho de incluir este ejemplo tiene como objetivo recordar al lector que el cálculo con números reales no tiene precisión infinita, aunque en la mayoría de nuestros problemas podemos suponer que el ordenador tiene precisión de sobra.

Ejercicio 5.10 Considere las ecuaciones:

$$x_0 = \sqrt{2} \qquad y_0 = 0 \qquad \pi_0 = 2+\sqrt{2} \tag{A.1}$$
$$x_{n+1} = \frac{1}{2}\left(\sqrt{x_n}+\frac{1}{\sqrt{x_n}}\right) \quad y_{n+1} = \frac{(1+y_n)\sqrt{x_n}}{x_n+y_n} \quad \pi_n = \pi_{n-1}\cdot y_n \cdot \frac{x_n+1}{y_n+1}$$

presentadas anteriormente (véase el problema 4.13), como un método iterativo para el cálculo aproximado del número π. Escriba un programa para leer un valor n, y escribir el correspondiente π_n. Para ello, tenga en cuenta módulos para leer el valor de entrada, inicializar las tres variables involucradas en el método iterativo, realizar una iteración y calcular π_n.

Solución al ejercicio 5.10 Con los módulos que hemos descrito, el algoritmo puede ser el siguiente:

1. Leer n, número de iteraciones.
2. Inicializar los valores de x_n, y_n, π_n.
3. Iterar tantas veces como indique n
4. Escribir el resultado.

La inicialización consiste en asignar los valores de índice cero. Para eso creamos una función que devuelve tres valores en tres parámetros pasados por referencia:

```
void Inicializar (double& x0, double& y0, double& pi0)
{
   x0 = sqrt(2.0);
   y0 = 0;
   pi0= 2 + x0;
}
```

El módulo para realizar una iteración recibe tres valores —x_n, y_n y π_n— y recalcula los tres nuevos valores. Para ello, podemos usar tres variables que pasan por referencia y que sirven tanto para entrada como para salida: tienen como entrada los datos iniciales y en la salida los nuevos datos. La función puede ser:

```
void Recalcular (double& xn, double& yn, double& pin)
{
   yn= (1+yn)*sqrt(xn)/(xn+yn);
   xn= (sqrt(xn) + 1/sqrt(xn)) / 2.0;
   pin= pin * yn * (xn+1) / (yn+1);
}
```

El programa completo podría ser como sigue:

```
1  #include <iostream>
2  #include <iomanip> // setprecision
3  #include <cmath> // sqrt
4  using namespace std;
5
6  int LeerIntPositivo()
7  { ... }
8
9  void Inicializar (double& x1, double& y1, double& pi1)
10 { ... }
11
12 void Recalcular (double& xn, double& yn, double& pin)
13 { ... }
14
15 int main()
16 {
17    int n;
18    double xn, yn, pin;
19
20    n= LeerIntPositivo();
21    Inicializar(xn, yn, pin);// crea x0, y0, pi0
22    for (int i=0; i<n; ++i) //  n vueltas
23       Recalcular(xn, yn, pin);
24    cout << "Pi= " << setprecision(20) << pin << endl;
```

```
25 }
```

Observe que:

- Los puntos suspensivos indican código de funciones que ya hemos visto anteriormente.
- Hemos incluido **iomanip** —línea 2— para poder usar *setprecision*. Con ello, hacemos que el valor real que se escribe en la línea 24 tenga hasta 20 dígitos de precisión.

Si quiere comprobar la velocidad de convergencia, aquí tiene algunos decimales del número π:

3.14159265358979323846 26

Más detalles sobre estas ecuaciones puede encontrarlos si busca información sobre el algoritmo de Borwein. Además, encontrarás otras alternativas más interesantes con las que podría practicar.

Ejercicio 5.11 Desarrolle un programa que lea tres números reales, los ordene y los escriba en pantalla.

Solución al ejercicio 5.11 Para resolver el problema es necesario ordenar tres valores reales. Para ello, considere el siguiente algoritmo:

1. Se ordenan los 2 primeros.
2. Se ordenan los 2 últimos.
3. Se ordenan los 2 primeros.

Después del primer paso, el mayor valor está en segundo o tercer lugar. Después del segundo, el mayor valor de los tres estará en último lugar. Un paso más para ordenar los dos primeros deja los tres valores ordenados.

Para resolver este problema tenemos que comparar dos valores y, si están desordenados, intercambiarlos. Este subproblema se puede resolver, como hemos visto, con la siguiente función:

```
void Intercambiar(double& a, double& b)
{
   double aux= a;
   a= b;
   b= aux;
}
```

Con esta función, el problema de ordenar tres valores se puede escribir como sigue:

```
void Ordenar(double& a, double& b, double& c)
{
   if (a>b) Intercambiar(a,b);
   if (b>c) Intercambiar(b,c);
   if (a>b) Intercambiar(a,b);
}
```

Observe que pasamos tres variables por referencia. Como resultado, obtendremos en la primera el menor valor, en la segunda el siguiente y en la última el mayor. El programa principal puede ser el siguiente:

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

// ... Funciones Ordenar e Intercambiar

int main()
{
   double a, b, c;

   cout << "Introduzca 3 valores: ";
   cin >> a >> b >> c;
   Ordenar(a, b, c);
   cout << "Ordenados: " << a << ' '<< b << ' '<< c << endl;
}
```

Lógicamente incluyendo las dos funciones que hemos comentado antes de **main**. Observe que no hemos incluido funciones para la entrada de datos o la salida, ya que el código es muy simple y es innecesario complicar la modularización.

A.6 Vectores de la STL

Ejercicio 6.1 Modifique el ejemplo del listado 5 de manera que no se lean 5 valores, sino 10.

Solución al ejercicio 6.1 Al haber usado el número 5 a lo largo de la función **main** tendremos que cambiar varias líneas. Es interesante no sólo que lo cambie de forma automática, sino que piense por qué tendrá que usar ese valor, especialmente en las últimas líneas, donde se cambia el valor a 9.

```
1  #include <iostream> // cout, cin
2  #include <vector>   // vector
3  using namespace std;
4
5  int main()
6  {
7     vector<double> notas(10);
8
9     for (int i=0; i<10; ++i)
10       cin >> notas.at(i);
11
12    double media= 0;
13    for (int i=0; i<10; ++i)
14       media= media + notas.at(i);
15    media= media/10;
16
17    cout << "Resultado: " << media << " es la media de ";
18    for (int i=0; i<9; ++i)
19       cout << notas.at(i) << ',';
20    cout << notas.at(9) << '.' << endl;
21 }
```

Como puede ver, hemos cambiado las líneas 7, 9, 13, 15, 18 y 20. Reflexione un momento sobre este cambio. Un cambio tan simple como éste ha provocado una revisión demasiado “costosa”. Además, proponsa a errores ya que si el código es más complejo, será sencillo que nos equivoquemos en algún valor. Seguro que ya intuye que la solución deberá cambiar.

Ejercicio 6.2 Modifique el ejemplo anterior de manera que el número de datos a procesar sólo aparezca en la declaración del vector.

Solución al ejercicio 6.2 Los valores que aparecen a lo largo de la función tendrán que referirse al tamaño usando la función `size()` del tipo vector. El codigo podría ser el siguiente:

```
#include <iostream> // Para cout, cin
#include <vector>   // Para vector
using namespace std;

int main()
{
   vector<double> notas(10);

   for (int i=0; i<notas.size(); ++i)
      cin >> notas.at(i);

   double media= 0;
   for (int i=0; i<notas.size(); ++i)
      media= media + notas.at(i);
   media= media/notas.size();

   cout << "Resultado: " << media << " es la media de ";
   for (int i=0; i<notas.size()-1; ++i)
```

```
        cout << notas.at(i) << ',';
    cout << notas.at(notas.size()-1) << '.' << endl;
}
```

Observe que `size` devuelve el tamaño del vector. En la última parte tenemos que hacer referencia al último elemento, la última posición, que está en `size()-1`, ya que los índices comienzan en cero.

Como puede deducir si comparamos con los programas anteriores, un cambio de tamaño en esta solución resulta mucho más fácil de realizar: sólo tendremos que cambiar la primera línea de **`main`**.

Ejercicio 6.3 Modifique el ejemplo anterior para garantizar que el tamaño es un entero positivo y las notas están en el rango $[0,10]$.

Solución al ejercicio 6.3 El problema del número de datos es conocido y está resuelto en temas anteriores. Podemos usar una lectura adelantada para facilitar un mensaje de error en caso de que no sea válido. La solución puede ser:

```
// ...
int main()
{
    int n;

    cout << "Introduzca el número de calificaciones: ";
    cin >> n;
    while (n<=0) {
        cout << "Error: el número debe ser positivo: ";
        cin >> n;
    }
```

Si el tamaño es correcto, la declaración del objeto `notas` resulta segura. Una vez declarado, podemos preguntar por cada una de las calificaciones. El código puede ser:

```
    vector<double> notas(n);

    for (int i=0; i<notas.size(); ++i) {
        cout << "Introduzca calificación " << i+1 << ": ";
        cin >> notas.at(i);
        while (notas.at(i)<0 || notas.at(i)>10) {
            cout << "El valor debe estar en [0,10]: ";
            cin >> notas.at(i);
        }
    }
    //...
}
```

En este código hemos planteado de nuevo una lectura adelantada. Observe que puede resultar algo más complicado que los ejemplos de los temas anteriores, pero no es más que el uso de un nombre algo más complejo al ser una referencia a un objeto de tipo **`double`** que está en una posición de un vector.

Finalmente, recordemos que no controlamos todos los tipos de error. Suponemos que el usuario está introduciendo números y que la lectura se realiza con éxito, aunque el número esté fuera de rango. Más adelante estudiaremos todas estas posibilidades.

Ejercicio 6.4 Modifique el ejemplo anterior añadiendo al final un trozo de código que primero calcule la media de las notas introducidas, y segundo calcule el número de datos que se encuentren por encima y por debajo de la media. En este trozo de código, use los corchetes como método de acceso a cada posición.

Solución al ejercicio 6.4 Dado que es un trozo que se añade al final, nos ahorramos volver a escribir las líneas que no cambian. A continuación se presenta únicamente el código final de la función `main`:

```
   // El mismo código ...

   int mayores= 0;
   int menores= 0;
   for (int i=0; i<notas.size(); ++i) {
      if (notas[i]>media) mayores++;
      if (notas[i]<media) menores++;
   }
   cout << "Media: " << media << endl;
   cout << "Por debajo: " << menores << endl;
   cout << "Por encima: " << mayores << endl;
   cout << "Iguales: " << notas.size()-(mayores+menores) << endl;
}
```

Observe que no se incluyen los que son idénticos a la media: la suma de menores y mayores podría ser menor que el número de datos.

Ejercicio 6.5 En la función anterior siempre obtendremos la posición de la primera ocurrencia del elemento en el vector. Proponga un cambio en la función con el objetivo de que se use para localizar otras posiciones donde también se encuentra el elemento.

Solución al ejercicio 6.5 La solución más simple es añadir un parámetro que indique la posición desde la que tendremos que buscar el elemento. La función podría ser la siguiente:

Con esta función podemos encontrar la primera ocurrencia

```
int Secuencial (const vector<double>& v, double elemento, int desde)
{
   int resultado= -1;
   for (int i=desde; i<v.size() && resultado==-1; ++i)
      if (v[i]==elemento)
         resultado= i;

   return resultado;
}
```

Con esta función podemos encontrar la primera ocurrencia pasando el valor cero como último parámetro de la función. Si desea la siguiente ocurrencia, se puede obtener pasando como último parámetro la posición del elemento encontrado más uno.

Ejercicio 6.6 En la función anterior escogemos entre dos mitades por cada iteración del bucle. La primera mitad empieza en `izq` e incluye la posición `centro`. Considere la posibilidad de dejar esa posición en la segunda mitad. ¿El código sería correcto?

Solución al ejercicio 6.6 Para dejar la posición del centro en la mitad derecha tenemos que cambiar la actualización de los índices. Concretamente podríamos escribir:

```
int BusquedaBinaria(const vector<int>& v, int elem)
{
   int izq= 0, der= v.size()-1, centro;

   while (izq<der) {
      centro= (izq+der)/2;
      if (v.at(centro)<elem)
         izq= centro;
      else der= centro-1;
   }

   return v.at(izq)==elem ? izq : -1;
}
```

de forma que si el elemento es mayor que el central, movemos el índice izquierdo al centro. La nueva iteración tendrá que buscar en la mitad derecha, incluida esa posición.

Esta función no sería correcta porque en caso de que el elemento esté justo en el centro no se localiza. Si es igual, la condición será falsa y se ejecutará la parte **else**. En esta parte nos quedamos con la mitad izquierda que no incluye el centro. Por tanto, ya no localizaremos el elemento.

Para resolverlo deberíamos tener en cuenta también si el elemento está o no en esa mitad. La lógica del algoritmo de búsqueda binaria nos obliga a escoger la mitad donde se encuentra el elemento. El cambio no era tan trivial. Podemos modificarlo cambiando la condición de la sentencia **if**:

```
int BusquedaBinaria(const vector<int>& v, int elem)
{
   int izq= 0, der= v.size()-1, centro;

   while (izq<der) {
      centro= (izq+der)/2;
      if (v.at(centro)<=elem)
         izq= centro;
      else der= centro-1;
   }

   return v.at(izq)==elem ? izq : -1;
}
```

En este caso sí que escogemos la mitad donde se encuentra el elemento. Responde al algoritmo, aunque de nuevo tenemos una función que puede fallar.

El problema está en que el bucle podría iterar infinitamente. La idea es que en cada iteración se descartan los elementos de una mitad, y cada vez tenemos un subvector más pequeño, hasta que los índices sean iguales. En esta versión no siempre ocurre.

Piense en un subvector de dos elementos, es decir, con los índices tal que `izq+1==der`. El valor `centro` será igual que `izq`, ya que la media de los índices es un valor entero. Si determinamos que el elemento está a la derecha del centro —incluido el centro— se intentará actualizar el valor de `izq`. Sin embargo, este valor sería idéntico y la nueva iteración tendría exactamente los mismos valores.

Ejercicio 6.7 Reorganice el código anterior de forma que el algoritmo esté incluido en su totalidad en una única función `OrdenarSeleccion`. Intente reescribirlo teniendo en cuenta la lógica del algoritmo en lugar de consultar los detalles del código anterior.

Solución al ejercicio 6.7 La solución puede ser la siguiente:

```
void OrdenarSeleccion(vector<int>& v) {
   for (int pos=0; pos<v.size()-1; pos++) {
      int pmin= pos;
      for (int i=pos+1; i<v.size(); i++)
         if (v.at(i) < v.at(pmin))
            pmin= i;
      int aux= v.at(pos);
      v.at(pos)= v.at(pmin);
      v.at(pmin)= aux;
   }
}
```

donde hemos eliminado las funciones auxiliares, tanto la búsqueda del mínimo como el intercambio de los dos valores.

Ejercicio 6.8 Considere la condición del bucle anterior. Es una condición compuesta que usa el operador lógico *AND*. ¿Qué ocurriría si cambiamos el orden de los operandos?

Solución al ejercicio 6.8 Cambiar el orden de los operandos da lugar al siguiente código:

```
void DesplazarHaciaAtras(vector<int>& v, int pos)
{
   int aux= v.at(pos);
   int i;
   for (i=pos; aux<v.at(i-1) && i>0; i--)
      v.at(i)= v.at(i-1);
   v.at(i)= aux;
}
```

La operación *Y* lógica es conmutativa, por lo que en principio es tentador pensar que el resultado no cambia. Sin embargo, podríamos decir que en C++ la operación no es conmutativa. La razón para ello es la evaluación en corto (sección 3.4.1, en la página 52).

Una condición compuesta se evalúa de izquierda a derecha de forma que si un operando determina el resultado, no se evalúa el resto de la expresión. En el ejemplo anterior, si la primera condición es **false**, se descarta la evaluación de la segunda, pues el resultado es con seguridad **false**.

Esta forma de evaluación podría considerarse una forma de optimizar el código. Si ponemos en primer lugar la condición más simple y además determina el resultado en la mayoría de los casos, nos ahorra la evaluación de la segunda parte. Sin embargo, lo más relevante de esta forma de evaluación es que puede hacer que el código funcione o no. En este ejemplo, es necesario que la condición tenga el orden adecuado.

En este código, el problema de evaluación se da cuando el elemento que se desplaza hacia la izquierda es el más pequeño de todos y, por tanto, llega hasta la posición cero.

La condición de parada no sólo busca la posición según el orden de elementos, sino que añade la comprobación de que `i` sea mayor que cero para parar el desplazamiento cuando lleguemos a la posición cero. Si llegamos al valor `i==0`, la evaluación de la condición no se puede realizar. La primera parte intentará acceder a la posición `-1`, que al estar fuera del vector provoca el fallo del programa.

Ejercicio 6.9 Reorganice el código anterior de forma que el algoritmo esté incluido en su totalidad en una única función `OrdenarInsercion`. Intente reescribirlo teniendo en cuenta la lógica del algoritmo en lugar de consultar los detalles del código anterior.

Solución al ejercicio 6.9 La solución puede ser la siguiente:

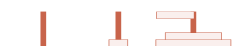
```
void OrdenarInsercion(vector<int>& v)
{
   int i, aux;
   for (int pos=1; pos<v.size(); pos++) {
      aux= v.at(pos);
      for (i=pos; i>0 && aux<v.at(i-1); i--)
         v.at(i)= v.at(i-1);
      v.at(i)= aux;
   }
}
```

donde las variables locales `i` y `aux` se han sacado fuera del bucle principal para que puedan reutilizarse en cada iteración.

Ejercicio 6.10 Reorganice el código anterior de forma que el algoritmo esté incluido en su totalidad en una única función `OrdenarBurbuja`. Intente reescribirlo teniendo en cuenta la lógica del algoritmo en lugar de consultar los detalles del código anterior.

Solución al ejercicio 6.10 La solución puede ser la siguiente:

```
void OrdenarBurbuja(vector<int>& v)
{
   for (int hasta=v.size()-1; hasta>0; hasta--)
      for (int i=0; i<hasta; i++)
```



```
            if (v.at(i) > v.at(i+1)) {
               int aux= v.at(i);
               v.at(i)= v.at(i+1);
               v.at(i+1)= aux;
            }
   }
```

donde hemos eliminado las funciones auxiliares, tanto la de subir el elemento como la de intercambio de los dos valores.

> **Ejercicio 6.11** Considere el código que se ha presentado para el algoritmo de ordenación por el método de la burbuja. Proponga algún cambio que mejore el caso de una secuencia de datos que ya esté ordenada.

Solución al ejercicio 6.11 La solución consiste en controlar si al subir un elemento ha habido algún cambio o no. Si el vector está ordenado, cuando intentamos subir un elemento no se realiza ningún intercambio. Para incorporarlo al código usamos una variable booleana que controla si se ha realizado. El código puede ser el siguiente:

```
bool SubirElemento (vector<int>& v, int hasta)
{
   bool cambio= false;
   for (int i=0; i<hasta; i++)
      if (v.at(i) > v.at(i+1)) {
         Intercambiar(v.at(i),v.at(i+1));
         cambio= true;
      }
   return cambio;
}
```

En esta función devolvemos como resultado de la operación si ha habido algún cambio o no. Con que haya un intercambio, el valor que se devolverá será `true`. Puede ser interesante puntualizar un error típico de estudiantes que comienzan a programar:

```
bool SubirElemento (vector<int>& v, int hasta)
{
   bool cambio= false;
   for (int i=0; i<hasta; i++)
      if (v.at(i) > v.at(i+1)) {
         Intercambiar(v.at(i),v.at(i+1));
         cambio= true;
      }
      else cambio= false;
   return cambio;
}
```

Esta versión es incorrecta, ya que puede devolver un valor `false` cuando realmente se han realizado intercambios en el vector.

El ejercicio se debe completar con la modificación de la función principal. En este caso usamos el valor booleando de la anterior. En concreto, tenemos que parar el algoritmo cuando se de un caso en que no se haya hecho ningún cambio, o equivalentemente, el algoritmo sigue mientras la función anterior devuelve `true`. La función puede ser:

```
void OrdenarBurbuja(vector<int>& v)
{
   bool seguir= true;
   for (int hasta=v.size()-1; hasta>0 && seguir; hasta--)
      seguir= SubirElemento(v,hasta);
}
```

Si un vector está ordenado, la primera llamada a *SubirElemento* devolverá false y el bucle terminará. El resultado es que sólo hemos tenido que realizar una iteración a lo largo del vector para confirmar que nada cambia.

Ejercicio 6.12 Considere el código que se ha presentado para el algoritmo de ordenación indirecta de datos. Escriba una función que recibe el vector de índices y datos y escribe en la salida estándar la secuencia de elementos ordenados.

Solución al ejercicio 6.12 El algoritmo es un simple bucle `for` como sigue:

```
void EscribirOrdenados(const vector<double>& original,
                       const vector<int>& orden)
{
   for (int i=0; i<orden.size(); ++i)
      cout << original[orden[i]] << endl;
}
```

donde los parámetros se han pasado como referencias constantes para usar los originales —por eficiencia— sin modificarlos.

Ejercicio 6.13 ¿Por qué el código anterior no es válido para leer un vector de n calificaciones?

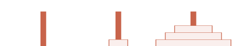
Solución al ejercicio 6.13 El problema está en que la declaración del contenedor `notas` se ha hecho indicando que el vector debe inicializarse con n elementos. Es decir, antes del bucle el vector ya contiene n datos almacenados —su `size()` ya es n— por lo que cualquier operación `push_back` implica un aumento de tamaño por encima de n. El resultado es que el vector acaba teniendo $2*n$ elementos donde la segunda mitad es la que realmente corresponde a los datos leídos.

Ejercicio 6.14 Considere el código de mezcla de vectores de la sección 6.2.5 (página 131). En él se declara un vector resultado con el tamaño necesario para almacenar la solución. Reescriba la función declarando un vector vacío que va creciendo conforme se añaden elementos al final.

Solución al ejercicio 6.14 En el algoritmo de mezcla se van realizando inserciones de nuevos elementos al final del vector. En la versión que se ha presentado necesitamos un índice —de nombre `res`— que nos controla la posición del vector que toca modificar. Empieza valiendo cero y va creciendo uno a uno conforme se insertan los elementos.

Si declaramos el vector vacío, podemos sustituir esas instrucciones con una operación `push_back` sin necesidad de usar ningún índice. El nuevo elemento simplemente se añade después de todos los anteriores. La función sería:

```
vector<double> Mezclar(const vector<double>& orig1,
                       const vector<double>& orig2 )
{
   vector<double> mezcla; // Vacío
   int i1= 0, i2= 0;

   // Mientras haya elementos en ambos vectores
   while (i1<orig1.size() && i2<orig2.size())
      if (orig1[i1] < orig2[i2]) {
         mezcla.push_back(orig1[i1]);
         i1++;
      }
      else {
         mezcla.push_back(orig2[i2]);
         i2++;
      }
   }

   for ( ;i1<orig1.size(); i1++)
      mezcla.push_back(orig1[i1]);

   for ( ;i2<orig2.size(); i2++)
      mezcla.push_back(orig2[i2]);

   return mezcla;
}
```



En esta solución no sólo hemos modificado las líneas directamente relacionadas con el cambio propuesto, sino que hemos hecho que los dos últimos bucles sean del tipo **for**. No es un cambio necesario, ni importante, pero probablemente hayamos mejorado la legibilidad del código.

Finalmente, es probable que considere que este nuevo código simplificado no es conveniente porque es más lento. Efectivamente, aunque la eficiencia de la operación *push_back* es muy buena y en la mayoría de los casos no es necesario mejorarlo, la versión anterior es más rápida. Los creadores del estándar C++ son conscientes, y por ello han facilitado una nueva función *reserve* para que el programador pueda optimizar el código si sabe la capacidad que el vector va a requerir.

Para implementar esta mejora añadimos una línea a la función. En concreto la siguiente:

```
vector<double> Mezclar(const vector<double>& orig1,
                       const vector<double>& orig2 )
{
   vector<double> mezcla; // Vacío
   mezcla.reserve(orig1.size()+orig2.size())

   int i1= 0, i2= 0;
   // ...
```

donde indicamos al vector que debería reservar espacio para al menos ese número de elementos.

Es importante que entienda que el resto del código es idéntico. No es lo mismo esta operación que indicar un tamaño inicial en la declaración. Si declaramos con un tamaño el vector ya contiene ese número de elementos. Si ejecutamos *reserve* simplemente indicamos al vector que prepare espacio para todo ese tamaño, simplemente para optimizar las operaciones. Por lo tanto, la operación *reserve* no cambia el tamaño del vector. En el ejemplo anterior, el vector *mezcla* seguirá teniendo un *size* de cero.

A este nivel del curso, la operación *reserve* no debería ser especialmente relevante, ya que es un aspecto simplemente para optimizar el código, haciendo que los algoritmos sean exactamente los mismos con los mismos resultados. Más adelante, cuando estudie de forma más detallada cómo podría estar implementado el tipo **vector**, entenderá mucho mejor por qué esta operación puede ser muy conveniente.

Ejercicio 6.15 La solución anterior refleja un mal diseño, ya que hemos incluido todo el código —que realiza varias tareas independientes— en la función **main**. Proponga una mejor solución. Para ello tenga en cuenta que:

1. Se puede crear una solución para la entrada de datos y otra para la salida.
2. Se puede crear una solución para calcular las medias. Para ello, tenga en cuenta que podemos escribir una función para calcular la media de un vector de **double** y usarla repetidamente.

Solución al ejercicio 6.15 Siguiendo las indicaciones del ejercicio descomponemos la solución en entrada, cálculos y salidas. El siguiente listado podría ser una solución:

```
1  #include <iostream> // Para cout, cin, endl
2  #include <vector>   // Para vector
3  using namespace std;
4
5  void EntradaDeDatos (vector<vector<double> >& notas)
6  {
7     for (int i=0; i<notas.size(); ++i) {
8        for (int j=0; j<notas[i].size(); ++j) {
9           cout << "Introduzca nota " << j+1 << " del alumno " << i+1 << ": ";
10          cin >> notas[i][j];
11       }
12    }
13 }
14
15 double Media(const vector<double>& v)
16 {
17    double media= 0;
```

```
18    for (int i=0; i<v.size(); ++i)
19       media+= v[i];
20    if (v.size()>0) // Si no datos, cero.
21       media/= v.size();
22
23    return media;
24 }
25
26 vector<double> Medias (const vector<vector<double> >& notas)
27 {
28    vector<double> medias;
29
30    for (int i=0; i<notas.size(); ++i)
31       medias.push_back(Media(notas[i]));
32
33    return medias;
34 }
35
36 void SalidaInformes (const vector<vector<double> >& notas,
37                      const vector<double>& medias, double global)
38 {
39    for (int i=0; i<notas.size(); ++i) {
40       cout << "Alumno " << i+1 << ":";
41       for (int j=0; j<notas[i].size(); ++j)
42          cout << ' ' << notas[i][j];
43       cout << '.' << " Media: " << medias[i] << endl;
44    }
45    cout << "Media global: " << global << endl;
46 }
47
48 int main()
49 {
50    int alumnos, ndatos;
51
52    cout << "Introduzca el número de alumnos: ";
53    cin >> alumnos;
54    cout << "Introduzca el número de notas para cada uno: ";
55    cin >> ndatos;
56
57    if (alumnos>0 && ndatos>0) {
58       vector<vector<double> > notas(alumnos,vector<double>(ndatos));
59       EntradaDeDatos(notas);
60
61       vector<double> medias = Medias(notas);
62       double media_global= Media(medias);
63
64       SalidaInformes(notas,medias,media_global);
65    }
66 }
```

En esta solución es interesante que observe que:

- La entrada de datos no se ha volcado completamente en una función. En la función `main` hemos dejado la parte del número de datos. Esta solución permite mantener dos funciones —`main` y `EntradaDeDatos`— bien definidas: con un objetivo concreto y fácilmente legibles. Además, la segunda se ha modificado de forma que sería válida también para matrices no rectangulares.
- Los cálculos no se han resuelto con una sola función. Se ha creado un módulo para calcular la función de un vector de objetos `double`, fácil de reutilizar y útil para calcular tanto la media de cada vector como la global. Observe cómo en la línea 31 se pasa un vector a este módulo, que a su vez devuelve una valor resultante que se añada al vector de medias. Además, dicho vector es un componente de un vector de vectores, todo pasado de forma eficiente al realizar todos los cálculos sobre el original gracias al paso por referencia. En línea 62 vuelve a reutilizarse este módulo para el cálculo de la media global.
- La salida se centra únicamente en mostrar los resultados. Tal vez puede haber pensado en la posibilidad de simplificar la interfaz de la función, pasando sólo el parámetro `notas` y hacer los cálculos —son dos líneas— dentro de la función. El resultado sería una función sin una intención concreta al realizar dos operaciones independientes.

Ejercicio 6.16 Escriba una función que recibe un vector de valores reales que indican las calificaciones de una serie de estudiantes, y devuelve una matriz de 10 filas. La primera fila contiene los índices de las calificaciones c tal que $c < 1$, la segunda los índices de las calificaciones que cumplen $1 \leq c < 2$, y así hasta la décima, que contiene los índices de las calificaciones que cumplen $9 \leq c \leq 10$.

Solución al ejercicio 6.16 Una solución puede ser la siguiente:

```
vector<vector<double> > Clasificar (const vector<double>& notas)
{
   vector<vector<double> > resultado(10); // Contiene 10 filas vacías

   for (int i=0; i<notas.size(); ++i) {
      int pos= notas[i];
      if (pos>9) pos= 9;
      resultado[pos].push_back(notas[i]);
   }

   return resultado;
}
```

Es interesante observar que:

1. La declaración del objeto `resultado` se hace con un tamaño de `10` vectores. Cada uno de ellos vacío, preparado para que se añadan elementos al final.
2. Dentro del bucle nos aseguramos de que la posición de la nota no sea mayor que `9`. Esta comprobación es necesaria para resolver el valor *10.0* que debe dar lugar al índice `9`. Otra solución perfectamente válida sería poner:

   ```
   if (pos==10) pos= 9;
   ```

   aunque la primera sería más robusta frente a errores en las notas en el caso de tener valores mayores que 10. En cualquier caso, este tipo de error no debería darse.

La devolución del resultado se hace mediante **`return`**. Si queremos resolver la función evitando que un objeto que puede ser muy grande se devuelva así[3], podemos modificarla para que el resultado se guarde en un parámetro por referencia. La nueva función podría ser:

```
void Clasificar (const vector<double>& notas, vector<vector<double> >&
     resultado)
{
   resultado.resize(10); // Fijamos 10 filas
   for (int i=0; i<10; ++i)
      resultado[i].clear(); // Vaciamos fila

   for (int i=0; i<notas.size(); ++i) {
      int pos= notas[i];
      if (pos>9) pos= 9;
      resultado[pos].push_back(notas[i]);
   }
}
```

En este código hemos realizado algo similar, pero anteponiendo varias líneas que garantizan que la matriz resultante tiene exactamente 10 líneas y cada una de ellas vacía:

- Tenemos que hacer `resize` para tener 10 líneas. El parámetro `resultado` podría estar vacío (se aumenta el tamaño), tener más de 10 líneas (se recorta el número de filas) o incluso tener exactamente 10 (no cambia nada).
- Si el parámetro `resultado` no estaba vacío, la operación de `resize` dejaría los primeros vectores con el contenido previo. Es necesario que estén vacíos para que las operaciones de adición de elementos funcionen correctamente.

[3]Comprobará que en el último estándar de C++, esta devolución es posible que se pueda realizar de forma muy eficiente.

Finalmente, podríamos haber resuelto la devolución a través de un parámetro de la siguiente forma[4]:

```
void Clasificar (const vector<double>& notas, vector<vector<double> >&
    resultado)
{
   vector<vector<double> > res(10);

   for (int i=0; i<notas.size(); ++i) {
      int pos= notas[i];
      if (pos>9) pos= 9;
      res[pos].push_back(notas[i]);
   }

   resultado= res;
}
```

donde creamos la solución en una variable local que al final sobreescribe el valor previo del parámetro.

Ejercicio 6.17 Modifique la función anterior para hacerla más genérica. Para ello, tenga en cuenta los números almacenados pueden estar en cualquier rango y que la función también recibe como parámetro el número de casilleros que queremos usar.

Solución al ejercicio 6.17 Una solución más genérica puede ser la siguiente:

```
1  void BucketSort (vector<double>& v, int ncasillas)
2  {
3     double minimo, maximo;   // Calculamos rango de valores
4     minimo= maximo= v[0];    // Al menos hay un elemento
5     for (int i=1; i<v.size(); ++i)
6        if (v[i] < minimo)
7           minimo= v[i];
8        else if (v[i] > maximo)
9              maximo= v[i];
10
11    if (maximo > minimo) { // Tal vez no haya distintos
12       vector<vector<double> > buckets(ncasillas);
13
14       double factor= ncasillas/(maximo-minimo);
15       for (int i=0; i<v.size(); ++i) {
16          int casilla= (v[i]-minimo)*factor;
17          if (casilla==ncasillas)
18             casilla--;
19          buckets[casilla].push_back(v[i]);
20       }
21
22       for (int b=0; b<buckets.size(); ++b)
23          OrdenarInsercion(buckets[b]);
24
25       // Volcamos de nuevo en v
26       int dest= 0; // posición a sobreescribir
27       for (int b=0; b<buckets.size(); ++b)
28          for (int i=0; i<buckets[b].size(); ++i) {
29             v[dest]= buckets[b][i];
30             dest++;
31          }
32    }
33 }
```

en la que el código modificado corresponde fundamentalmente a la primera parte, hasta la línea 20. Lo que se ha cambiado se refiere a la distribución de los elementos en casillas. En concreto:

- Tenemos que calcular el rango de valores en el que se encuentran los elementos a ordenar (líneas de 3 a 9). Hemos supuesto que hay al menos un elemento a ordenar, es decir, que si llamamos a la función es porque el vector `v` no está vacío. Podríamos mejorar la solución si

[4]Probablemente, sería mejor solución realizar un intercambio de los contenidos de las variables con la función `swap` del estándar C++.

queremos evitar esta precondición, facilitando su uso, por ejemplo añadiendo una condición para comprobar si el vector está vacío.

- El cálculo de la casilla correspondiente (línea 16) se realiza mediante una transformación lineal sencilla. La expresión usada es:

$$\frac{v_i - minimo}{maximo - minimo} \cdot ncasillas$$

donde vemos que la primera parte da lugar a un número en el rango $[0,1]$ que al multiplicarlo por el número de casillas se convierte en un valor en $[0, ncasillas]$.
Observe que los valores `v[i]` que sean iguales a `maximo` darán como resultado el valor `ncasillas`, que no es válido. En este caso lo corregimos —línea 17— asignando la última casilla (`ncasillas−1`). Otra forma de corregirlo podría ser modificar `factor` para que fuera ligeramente inferior y que la conversión diera lugar al un valor en $[0, ncasillas)$, es decir, menor estricto que `ncasillas`.

A.7 Cadenas de la STL

Ejercicio 7.1 Considere el siguiente trozo de código con dos bucles sobre un objeto `cadena` de tipo `string`. ¿Cuáles son los errores que encuentra? Indique brevemente el efecto que podrá tener su ejecución.

```
for (int i=1; i<=cadena.size(); i++)
   cout << cadena.at(i);
for (int i=1; i<=cadena.size(); i++)
   cout << cadena[i];
```

Solución al ejercicio 7.1 El error de este código es considerar que el rango de índices válidos para una cadena de caracteres empieza en 1. Los bucles se deberían haber escrito considerando el rango desde el cero al tamaño menos uno.

El código genera un error en tiempo de ejecución en caso de que la cadena no esté vacía. El tipo de error será distinto para cada bucle:

- En el primer caso usamos el acceso con `at`. Esta operación comprueba si el índice es correcto. El programa genera un error en cuanto accedemos a la posición `cadena.size()`.
- En el segundo caso el efecto es indeterminado. Es un error, pues estamos accediendo a una posición incorrecta, pero el programa no hace ninguna comprobación. El intento de acceso a la posición incorrecta puede desde detener el programa hasta parecer que no tiene errores.

Para finalizar la solución, recordemos que el codigo correcto sería:

```
for (int i=0; i<cadena.size(); i++)
   cout << cadena.at(i);
for (int i=0; i<cadena.size(); i++)
   cout << cadena[i];
```

Ejercicio 7.2 Escriba un programa que lea dos cadenas (`c1` y `c2`) y cuente el número de apariciones de cada una de las letras de `c1` en `c2`.

Solución al ejercicio 7.2 Resolvemos el problema directamente en la función `main`, ya que el algoritmo es muy simple. El código puede ser el siguiente:

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

```
int main()
{
   string c1, c2;

   cout << "Introduzca dos cadenas: ";
   cin >> c1 >> c2;

   for (int i=0; i<c1.size(); ++i) {
      int contador= 0;
      for (int j=0; j<c2.size(); ++j)
         if (c1[i]==c2[j])
            contador++;
      cout << "Letra " << c1[i] << ": " << contador << endl;
   }
}
```

Ejercicio 7.3 Escriba un programa que lea una cadena y escriba en la salida estándar si es un palíndromo. Para ello, tendrá en cuenta que todas las letras de entrada están en minúscula, sin tildes y sin espacios.

Solución al ejercicio 7.3 Una primera solución podría ser la siguiente:

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

int main()
{
   string cadena;
   cout << "Introduzca cadena: ";
   getline(cin,cadena);

   bool palindromo= true;
   for (int i=0; i<cadena.size(); ++i)
      if (cadena[i] != cadena[cadena.size()-1-i])
         palindromo= false;

   if (palindromo)
      cout << cadena << " es un palíndromo." << endl;
   else cout << cadena << " no es palíndromo." << endl;
}
```

donde vemos que básicamente hemos resuelto el problema con un bucle que recorre toda la cadena, carácter a carácter, comprobando si el carácter "simétrico" es idéntico. En el momento en que encontramos dos distintos hacemos que la variable booleana `palindromo` registre que no es un palíndromo. Es interesante recordar un error bastante frecuente de estudiantes sin experiencia al programar este bucle como sigue:

```
   for (int i=0; i<cadena.size(); ++i)
      if (cadena[i] != cadena[cadena.size()-1-i])
         palindromo= false;
      else palindromo= true;
```

dando lugar a un resultado incorrecto, ya que el resultado lo determina el último carácter comprobado. Por ejemplo, una cadena con la primera y la última letra idénticas sería un palíndromo independientemente de las demás.

Es fácil ver que podríamos realizar algunas modificaciones para mejorar la eficiencia del algoritmo. En concreto puede cambiar:

- El bucle no tiene que iterar hasta el tamaño de la cadena. En caso de hacerlo estamos realizando dos veces la misma comprobación. Por ejemplo, comprobamos la primera letra con la última y la última con la primera. Bastaría con llegar a la mitad del tamaño.
- La condición de parada el bucle puede ser compuesta para incluir también la variable `palindromo`. En cuanto detectamos una letra distinta, ya podemos terminar el bucle.

Incluso podríamos realizar un cambio mayor si en lugar de usar un índice usamos dos. El código podría ser el siguiente:

```
int i= 0;
int j= cadena.size()-1;
while (i<j && palindromo) {
   if (cadena[i]!=cadena[j])
      palindromo= false;
   i++; j--;
}
```

Ejercicio 7.4 Razone brevemente por qué en el ejercicio anterior es posible concatenar múltiples *cadenas-C* cuando realmente no es válida la operación '+' entre ellas.

Solución al ejercicio 7.4 La operación es posible porque la suma es un operador que se evalúa de izquierda a derecha. Cuando realizamos la primera suma, mezclamos un objeto `string` con una *cadena-C*, siendo el resultado de tipo `string`. En la siguiente suma, volvemos a tener este tipo a la izquierda. Es decir, todas las sumas son válidas porque en todas el operando izquierdo es de tipo `string`. En ningún caso aparece una suma de dos objetos de tipo *cadena-C*.

Ejercicio 7.5 Escriba un programa que lea una línea y vuelva a escribirla sin espacios. Para ello, calcule una nueva cadena modificando la original mediante la eliminación de todos los caracteres que sean espacio.

Solución al ejercicio 7.5 La solución puede ser la siguiente:

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

int main()
{
   string cadena;
   cout << "Introduzca línea: ";
   getline(cin,cadena);

   int i= 0;
   while (i<cadena.size())
      if (cadena[i]==' ')
         cadena.erase(i,1);
      else i++;

   cout << "Cadena sin espacios: " << cadena << endl;
}
```

En este problema es especialmente interesante que se fije en cómo se ha diseñado el bucle. En lugar de usar un bucle `for` usamos el bucle `while` ya que el contador `i` sólo aumenta en caso de que el carácter no sea un espacio. Si encontramos un espacio, el borrado del carácter hace que el índice `i` ya esté situado en la posición siguiente.

Ejercicio 7.6 Escriba un programa que lea una línea e indique si es un palíndromo. Para ello, será necesario crear varias funciones. En concreto puede considerar las siguientes:

1. Una función que elimine los espacios y tabuladores de una cadena.
2. Una función que pase las mayúsculas a minúsculas.
3. Una función que devuelva la cadena inversa.

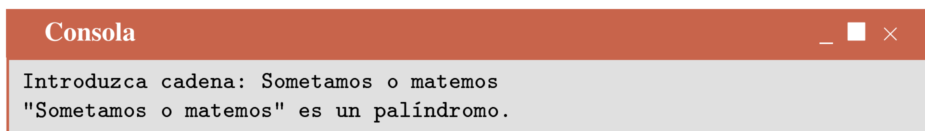
El resultado será, por tanto, un programa que procesa una cadena con las anteriores funciones y comprueba si la cadena es igual a su inversa.

Solución al ejercicio 7.6 Una solución al ejercicio puede ser la siguiente:

```
#include <iostream>
#include <string>
#include <cctype> // tolower, isspace
using namespace std;

void Minuscula(string& cad)
{
   for (int i=0; i<cad.size(); ++i)
      cad[i]= tolower(cad[i]);
}

void QuitarEspacios(string& cad)
{
   int i= 0;
   while (i<cad.size())
      if (isspace(cad[i]))
         cad.erase(i,1);
      else i++;
}

string Inversa(const string& cad)
{
   string inversa;

   for (int i=cad.size()-1; i>=0; --i)
      inversa.push_back(cad[i]);

   return inversa;
}

bool Palindromo(const string& cadena)
{
   string simplificada= cadena;
   QuitarEspacios(simplificada);
   Minuscula(simplificada);

   return simplificada == Inversa(simplificada);
}

int main()
{
   string cadena;
   cout << "Introduzca cadena: ";
   getline(cin,cadena);

   cout << '\"' << cadena << '\"';
   if (Palindromo(cadena))
      cout << " es un palíndromo." << endl;
   else cout << " no es palíndromo." << endl;
}
```

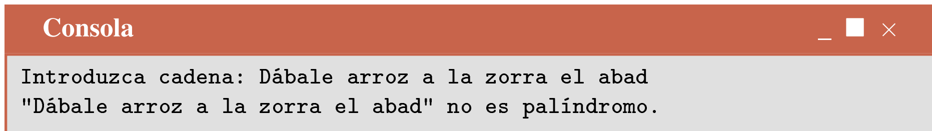
En la que hemos usado las funciones `tolower` y `isspace` del estándar para pasar a minúscula o comprobar si es un carácter de espacio, respectivamente.

Un ejemplo de su ejecución es el siguiente:



Ejercicio 7.7 Reconsidere el programa del palíndromo resuelto en el ejercicio 7.6 (página 156). Suponga que hemos resuelto el programa suponiendo una codificación *ISO-8859-1*. ¿Qué puede ocurrir si ejecutamos el programa en un sistema con codificación *UTF8*?

Solución al ejercicio 7.7 El problema se presenta cuando tenemos caracteres que se codifican de forma diferente. Un ejemplo de su ejecución sería el siguiente:



```
Consola
Introduzca cadena: Dábale arroz a la zorra el abad
"Dábale arroz a la zorra el abad" no es palíndromo.
```

donde debería haber indicado que es un palíndromo.

El problema es que la vocal con tilde no se codifica con un carácter, sino con dos. La posición cero de la cadena tiene la letra `'D'`, la posición uno tiene la primera mitad de la letra `á`, que no coincide con la penúltima letra (que sí es una `a`, sin tilde). La única forma de resolver este problema sería transformar la cadena para eliminar las tildes, o transformarla a una codificación en la que cada letra esté en un sólo carácter.

Finalmente, es interesante puntualizar que incluso en caso de tener la codificación con un sólo **`char`**, como la codificación *ISO-8859-1*, deberíamos asegurarnos de que la función **`tolower`** funciona correctamente. Si el sistema está configurado regionalmente como de lengua inglesa, las mayúsculas con tilde no se convertirán en minúsculas con tilde.

A.8 Estructuras y pares

Ejercicio 8.1 Considere el programa anterior, donde se leen y escriben los datos de una persona. Modifíquelo para que esa lectura/escritura se haga dos veces (por ejemplo, duplique las dos últimas líneas de **`main`**). ¿Qué ocurre? Proponga una solución.

Solución al ejercicio 8.1 Tras modificar el programa y ejecutarlo, introducimos los datos de una persona, el programa los imprime, y cuando comienza la segunda lectura obtenemos:

```
Consola
Introduzca apellidos: García García
Introduzca nombre: Pepe
Introduzca e-mail: pepe@site.com
Introduzca sexo (M/F): M
Introduzca fecha de nacimiento: 1/1/1970
Nombre: García García,Pepe
e-mail: pepe@site.com
Sexo: M
Fecha de nacimiento: 1/1/1970
Introduzca apellidos: Introduzca nombre:
```

El problema se presenta porque realizamos lecturas de datos de tipo entero con el operador >> y lecturas de líneas con *`getline`*. Cuando leemos un entero, la lectura se detiene con el primer carácter —posiblemente un separador— que sigue al entero. En la última lectura de la fecha de nacimiento, se queda un carácter de salto de línea en la entrada, preparado para la siguiente lectura. La lectura de una línea da lugar a una línea vacía[5].

Podemos resolverlo de distintas formas, por ejemplo eliminando el carácter salto de línea después de la fecha de nacimiento, o incluso hasta final de línea por si hay otros separadores. Aunque no sería una buena solución, pues hace que esa función de lectura sea más difícil de usar en otros casos.

[5] En la sección 7.2.4, página 148 puede encontrar más detalles.

Otra solución es cambiar la lectura con *getline* por la lectura de una línea que tenga contenido. Es decir, podemos crear una función para repetir *getline* hasta que haya caracteres con información. Por ejemplo, podemos usar *LeerLineaConContenido*:

```
bool TieneChicha(const string& str)
{
   bool tiene= false;
   for (int i=0; i<str.size() && !tiene; ++i)
      if ( ! isspace(str[i]) )
         tiene= true;
   return tiene;
}

void LeerLineaConContenido(string& str)
{
   do{
      getline(cin, str);
   }while (!TieneChicha(str));
}
```

donde podemos ver que hemos creado una función auxiliar para comprobar si una cadena tiene caracteres con información que no sea del tipo *espacios blancos*.

Por último, resulta interesante enfatizar que aunque el código del programa que se propone ilustra los contenidos del tema, en la práctica no es suficientemente robusto para gestionar correctamente los distintos casos de error. Si queremos implementar una solución más completa, deberíamos añadir o modificar el código para que fuera más útil. Considere por ejemplo la lectura del sexo, en donde podríamos comprobar que la respuesta es una de las opciones válidas, la lectura de la fecha para evitar valores sin sentido, o el correo electrónico, donde podemos esperar que tenga un formato válido.

Ejercicio 8.2 Escriba una función *Distancia* que recibe dos parámetros de tipo *Punto* y devuelve la distancia euclídea entre ellos. Recuerde que la distancia entre dos puntos p,q es $\sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$.

Solución al ejercicio 8.2 La solución podría ser la siguiente:

```
double Distancia(const Punto& p1, const Punto& p2)
{
   return sqrt((p1.x-p2.x) * (p1.x-p2.x) + (p1.y-p2.y) * (p1.y-p2.y));
}
```

donde hemos pasado las dos estructuras como una referencia constante para evitar la copia.

Ejercicio 8.3 Considere un nuevo tipo de dato *BoundingBox* definido para delimitar un rectángulo controlado por la esquina superior izquierda e inferior derecha. Use el tipo *Punto* anterior para definir esta nueva estructura. Una vez definida, escriba una función que lee un entero *n* y a continuación *n* objetos de tipo *Punto* para obtener el *BoundingBox* mínimo que los contiene. Observe que no es necesario definir ningún vector de puntos.

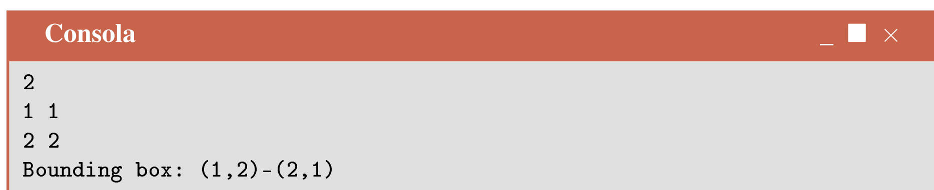
Solución al ejercicio 8.3 Para resolver el problema diseñamos la solución dividiendo el código en varias funciones. Podríamos profundizar en el diseño proponiendo otros módulos. Sin embargo, nos limitaremos a mostrar los más básicos para ilustrar los contenidos del tema y sugerir la ventaja de implementar funciones como operaciones asociadas a los tipos de datos que se definen. El código puede ser el siguiente:

```
1 #include <iostream>
2 using namespace std;
3
4 struct Punto {
5    double x, y;
6 };
```

```cpp

struct BoundingBox {
   Punto supizq, infder;
};

Punto LeerPunto()
{
   Punto p;
   cin >> p.x >> p.y;
   return p;
}
void EscribirPunto(const Punto& p)
{
   cout << '(' << p.x << ',' << p.y << ')';
}

void AmpliarPunto (BoundingBox& box, const Punto& p)
{
   if (p.x<box.supizq.x) box.supizq.x= p.x;
   if (p.y>box.supizq.y) box.supizq.y= p.y;
   if (p.x>box.infder.x) box.infder.x= p.x;
   if (p.y<box.infder.y) box.infder.y= p.y;
}

BoundingBox LeerPuntos()
{
   int n;
   cin >> n;

   BoundingBox box;
   Punto p;
   p= LeerPunto();
   box.supizq= box.infder= p;

   for (int i=1; i<n; i++) // n-1 restantes
      AmpliarPunto (box,LeerPunto());

   return box;
}

int main()
{
   BoundingBox box;

   box= LeerPuntos();

   cout << "Bounding box: ";
   EscribirPunto(box.supizq);
   cout << '-';
   EscribirPunto(box.infder);
}
```

Observe la llamada a `AmpliarPunto` —línea 42— que recibe el resultado de la función `LeerPunto` para mandarlo directamente como segunda parámetro.

Un ejemplo de ejecución de este programa es el siguiente:



Probablemente, habrá implementado el problema de otra forma. Incluso pensará que sería interesante modificar esta solución, cambiando algunas funciones o incluso ampliando con alguna adicional. Esta discusión se acerca algo más a la forma en que se diseñaría un tipo de dato y sus operaciones, que se sale de los objetivos de este tema.

Ejercicio 8.4 Suponga que disponemos de la función `OrdenFecha`. Escriba la función `OrdenPersonas` usando como único campo de ordenación la fecha de nacimiento. Se ordenarán por edad, por lo que la fecha de nacimiento posterior implica un orden menor. Una vez resuelto, escriba el algoritmo de ordenación por selección para ordenar un vector de objetos de tipo `Persona` según su edad.

Solución al ejercicio 8.4 La función `OrdenPersonas` sería muy simple, ya que devuelve casi el mismo valor entero que `OrdenFecha`. Lo único a tener en cuenta es que el orden es el contrario, por lo que bastará con cambiarlo de signo. El código podría ser el siguiente:

```
int OrdenPersonas (const Persona& l, const Persona& r)
{
   return -OrdenFecha(l.nacimiento,r.nacimiento); // Lo invertimos
}

void OrdenarSeleccion(vector<Persona>& v) {
   for (int pos=0; pos<v.size()-1; pos++) {
      int pmin= pos;
      for (int i=pos+1; i<v.size(); i++)
         if (OrdenPersonas(v.at(i),v.at(pmin)) < 0)
            pmin= i;
      Persona aux= v.at(pos);
      v.at(pos)= v.at(pmin);
      v.at(pmin)= aux;
   }
}
```

donde vemos el algoritmo de ordenación que hemos presentado anteriormente, aunque cambiando la condición de la sentencia **if−else** que ahora no tiene que usar el operador < entre dos objetos, sino comprobar el signo del entero que devuelve una función.

Ejercicio 8.5 Considere una estructura con dos campos `c1` y `c2`. Suponga que aplicamos primero un algoritmo de ordenación para ordenarlas sólo por el campo `c2` y posteriormente otro algoritmo de ordenación sólo por el campo `c1`. ¿Qué resultado obtendrá si usamos para ello algoritmos de ordenación estables?

Solución al ejercicio 8.5 El resultado es que todos los elementos quedarán ordenados por `c1`, ya que es el algoritmo que se aplica en segundo lugar. Sin embargo, aquellos que tengan el mismo valor en este campo quedan agrupados en el mismo orden en que se encontraban antes de ordenar, es decir, en el orden que dejó la ordenación por `c2`.

El efecto será que quedarán ordenados como si el criterio de ordenación tuviera en cuenta los dos campos, con el campo `c1` de mayor prioridad.

Ejercicio 8.6 Indique si los algoritmos de *selección*, *inserción* y *burbuja* son estables. En caso de que sea estable, ¿podría modificarse para que, aun obteniendo un resultado ordenado, deje de ser estable?

Solución al ejercicio 8.6 El algoritmo de *selección* no es estable, ya que es posible hacer que dos elementos iguales cambien de orden en el vector final. Por ejemplo, imagine que el primer elemento del vector es el más grande y el último el más pequeño. Cuando ordenamos, en la primera pasada, estos dos valores se intercambian. Cualquier elemento igual al mayor ha quedado a la izquierda del elemento que hemos pasado a la posición más alta.

Los algoritmos de *inserción* y *burbuja* son estables. La implementación más natural resulta estable, pues cuando dos elementos son iguales no se intercambian. Por ejemplo, en el algoritmo de inserción, al buscar una posición de inserción hacia la izquierda, el algoritmo para cuando llega a

un elemento igual. Por otro lado, el algoritmo de burbuja también es estable porque cuando dos elementos son iguales no los intercambia.

Lógicamente, estos dos últimos podrían implementarse de forma que no sean estables. Por ejemplo, el de inserción podría seguir buscando una posición más a la izquierda incluso cuando sean iguales, o el de burbuja podría intercambiar dos elementos siendo iguales. Aunque las implementaciones darían lugar a vectores ordenados, deben considerarse errores. No es aceptable una implementación que es más costosa y que además hace perder al algoritmo su característica de estabilidad.

Ejercicio 8.7 Complete el ejemplo con las dos funciones de ordenación por días y meses.

Solución al ejercicio 8.7 La solución es muy similar, incluso más simple. La única diferencia es que el número de casilleros y la asignación de cada uno de ellos es distinta. En el caso de los días establecemos 31 casilleros y en el caso de los meses 12.

Podemos crear una función `VolcarBuckets` auxiliar que resuelve el problema de volcar los casilleros de nuevo al vector original. La solución podría ser:

```
void VolcarBuckets (const vector<vector<Persona> >& buckets,
                                                     vector<Persona>& v)
{
   int dest= 0; // posición a sobreescribir
   for (int b=0; b<buckets.size(); ++b)
      for (int i=0; i<buckets[b].size(); ++i) {
         v[dest]= buckets[b][i];
         dest++;
      }
}

void OrdenarPorDia (vector<Persona>& v)
{
   vector<vector<Persona> > buckets(31); // Un casillero por día

   // Volcamos a los casilleros
   for (int i=0; i<v.size(); ++i) {
      int b= v[i].nacimiento.dia−1;
      buckets[b].push_back(v[i]);
   }
   VolcarBuckets (buckets, v);
}

void OrdenarPorMes (vector<Persona>& v)
{
   vector<vector<Persona> > buckets(12); // Un casillero por mes

   // Volcamos a los casilleros
   for (int i=0; i<v.size(); ++i) {
      int b= v[i].nacimiento.mes−1;
      buckets[b].push_back(v[i]);
   }
   VolcarBuckets (buckets, v);
}
```

Note que la función `OrdenarPorAnio` incluye al final un trozo de código para devolver de nuevo los elementos al original. Lo lógico es que se modificara para llamar a la función `VolcarBuckets`.

Ejercicio 8.8 Modifique la función `BuscarInsertar` del ejercicio anterior de forma que el vector de pares esté siempre ordenado. Para implementar la función, use el algoritmo de búsqueda dicotómica (sección 6.2.4 en página 120) a fin de mejorar la eficiencia.

Solución al ejercicio 8.8 Adaptamos la función de búsqueda binaria para localizar la clave en el vector de pares. La función devolverá no sólo si está, sino la posición donde se encuentra o debería encontrarse. Como tenemos que devolver dos valores, podemos devolverlos como un par:

first Indica si lo ha encontrado, es decir, **true** si la clave ya estaba en el diccionario.
second Indica la posición. Si lo ha encontrado, la posición donde se encuentra, y si no, la posición donde debería insertarse para dejar el diccionario ordenado.

El código podría ser el siguiente:

```
// Devuelve si está y posición donde está o debería estar
pair<bool,int> BusquedaBinaria(const vector<pair<string,int> >& v,
                               const string& clave)
{
   int izq= 0, der= v.size()-1, centro;

   while (izq<=der) {
      centro= (izq+der)/2;
      if (v.at(centro).first>clave)
         der= centro-1;
      else if (v.at(centro).first<clave)
              izq= centro+1;
           else return pair<bool,int>(true,centro);
   }

   return pair<bool,int>(false,izq);
}

// Busca la clave en el diccionario y devuelve su localización
int BuscarInsertar (vector<pair<string,int> >& diccionario,
                    const string& clave)
{
   pair<bool,int> loc= BusquedaBinaria(diccionario,clave);
   if (loc.first==false) {
      diccionario.resize(diccionario.size()+1); //Ampliamos 1
      for (int i=diccionario.size()-1; i>loc.second; i--)
         diccionario[i]= diccionario[i-1];
      diccionario[loc.second]= pair<string,int>(clave,0);
   }

   return loc.second;
}
```

A.9 Recursividad

Ejercicio 9.1 Considere el problema de calcular la potencia m^n, donde m y n son dos valores enteros. En el caso general, ese cálculo se puede definir como $m \cdot m^{n-1}$. Escriba una función recursiva que lo implemente. Recuerde que debe incluir un caso base.

Solución al ejercicio 9.1 La solución puede ser la siguiente:

```
int Potencia(int base, int exp)
{
   if (exp==0)
      return 1;
   else return base * Potencia(base, exp-1);
}
```

Ejercicio 9.2 Escriba una función recursiva `VectorAlReves` que tenga como caso base que sólo hay un elemento a imprimir.

Solución al ejercicio 9.2 Si el caso base se da cuando sólo hay un elemento a imprimir, es que el valor del parámetro `desde` vale exactamente el tamaño menos uno. En este caso, el problema se resuelve fácilmente escribiendo sólo el elemento de dicha posición.

La solución puede ser la siguiente:

```
void VectorAlReves(const vector<int>& v, int desde)
{
```

```
        if (desde==v.size()-1)
            cout << v[desde] << endl;
        else {
            VectorAlReves(v,desde+1);
            cout << v[desde] << endl;
        }
    }
```

Observe que hemos supuesto que el vector tiene algún elemento. Si queremos que la función sea válida para vectores vacíos habría que incluir una condición para que se imprimiera sólo en caso de tener elementos.

Ejercicio 9.3 Un algoritmo recursivo, también válido para este listado, consiste en listar los elementos de la mitad derecha del vector al revés, seguidos de los elementos de la mitad izquierda al revés. Escriba una función que implemente este algoritmo.

Solución al ejercicio 9.3 El caso base puede ser que haya sólo un elemento. En este caso, la solución es simplemente escribirlo. En caso de que haya más de un elemento, podemos dividir el vector en dos partes. La solución podría ser la siguiente:

```
void VectorAlReves(const vector<int>& v, int desde, int hasta)
{
    if (desde==hasta)
        cout << v[desde] << endl;
    else if (desde<hasta) {
                int centro= (desde+hasta)/2;
                VectorAlReves(v,centro+1,hasta);
                VectorAlReves(v,desde,centro);
            }
}
```

Observe que hemos cambiado la cabecera, ya que en la llamada recursiva tenemos que pasar un subvector cualquiera. Ya no es suficiente con un solo índice para indicar el subvector hasta el final, ya que para indicar la primera mitad, necesitamos decir hasta dónde.

Ejercicio 9.4 En la solución anterior buscamos un divisor siempre desde el número 2. ¿Cómo podríamos evitarlo? Modifique la función para incluir la mejora.

Solución al ejercicio 9.4 El algoritmo consiste en añadir al final del vector el conjunto de números primos divisores de n. Estos números se añaden de manera creciente, de forma que si ya hemos añadido algún número, los siguientes seguro que no serán menores. Para aprovecharlo, podemos empezar inicializando el valor de p al último primo que contiene `vec`.

La solución puede ser la siguiente:

```
void Descomponer (int n, vector<int>& vec)
{
    int p;
    if (vec.size()==0)
        p= 2;
    else p= vec[vec.size()-1];

    while (n%p!=0)
        p++;

    vec.push_back(p);
    if (p<n)
        Descomponer(n/p,vec);
}
```

donde podemos ver que la búsqueda de un número primo se realiza desde 2 en caso de que el vector `vec` esté vacío, es decir, cuando aún no hemos añadido el primer primo que divide a n.

> **Ejercicio 9.5** Considere un programa que llama a las dos funciones —*AnadirFactores* y la versión final de la función *Descomponer*— para escribir los resultados de cada una de ellas para un mismo número *n*, ¿qué diferencias habrá?

Solución al ejercicio 9.5 La diferencia está en el orden en que se añaden los primos al resultado. En la primera de ellas se añade el número primo y luego el resto de primos, mientras que en la segunda se añaden el resto de primos y luego se completa con el encontrado.

En el listado aparecerán —como es de esperar— los mismos elementos, pero en el primero será de menor a mayor y en el segundo al revés.

> **Ejercicio 9.6** Los elementos de las dos mitades se disponen en el vector auxiliar de manera ascendente, en el primer caso, y descendente en el segundo. ¿Por qué?

Solución al ejercicio 9.6 Es una forma de evitar comprobar el tamaño de los vectores que se mezclan. Si tuviéramos las dos secuencias de forma ascendente, el bucle de mezcla tendría que comprobar si una de las dos secuencias "se ha terminado". Cuando a una de las secuencias no le quedan elementos que volcar, habría que dejar de hacer comparaciones y volcar lo que reste de la otra.

En este caso, podemos seguir hasta que se acaban las dos secuencias, ya que si una se acaba, el elemento que pasará a compararse en la siguiente iteración es el último de la otra secuencia, por lo que el algoritmo descargará adecuadamente el resto de elementos hasta completar la mezcla.

Observe que no es más que un detalle para generar un algoritmo distinto. Es una mejora relativamente pequeña y no debería pensar que debe realizarse de esta forma, aunque es posible que encuentre soluciones de este tipo en la bibliografía.

> **Ejercicio 9.7** Para realizar la mezcla se crea un vector auxiliar cada vez que se llama a la función, tanto en la llamada principal como para cualquier llamada recursiva. Dado el número tan alto de llamadas, puede resultar aconsejable usar la misma variable auxiliar repetidamente en todas las mezclas. Proponga una modificación de la función, incluyendo la cabecera, para usar un único vector auxiliar (que tendrá un tamaño igual al del vector original) en todas las llamadas.

Solución al ejercicio 9.7 La solución sería disponer de un objeto *auxiliar* que fuera un vector con el mismo tamaño que el vector a ordenar. Dado que el vector se utiliza en cada llamada independientemente, no hay ningún problema en volver a usar la misma variable en todas las llamadas.

En la mayoría de las llamadas recursivas se desperdiciaría memoria porque el vector tiene el tamaño original y tal vez la mezcla sea de pocos elementos. A pesar de ello, resultaría un código más eficiente al ahorrar la creación y destrucción del objeto auxiliar con un tamaño ajustado al vector a mezclar. Además, no conseguimos mermar las posibilidades del algoritmo, pues en cualquier caso el vector auxiliar de la primera llamada sería del mismo tamaño del vector, por lo que no hay ahorro en espacio.

Una forma muy sencilla de usar el mismo vector es pasarlo por referencia. En el punto de llamada a la función de ordenar tenemos que declarar un objeto *auxiliar* tan grande como el vector a ordenar y pasarlo a la función para que lo utilice. Un esquema de su uso puede ser el siguiente:

```
void OrdenMezcla (vector<int>& vec, int izqda, int drcha,
                  vector<int>& auxiliar )
{
   // ... idéntico, sin definir "auxiliar"
```

```
}
// ...
int main()
{
   vector<int> datos; // vector a ordenar
   // ...
   vector<int> buffer(datos.size());
   OrdenMezcla (datos, 0, datos.size()−1, buffer);
   // ...
}
```

A.10 Introducción a flujos de E/S

Ejercicio 10.1 Como podemos ver, pueden definirse flujos sin buffer. ¿Tiene sentido que **cerr** no tenga buffer?

Solución al ejercicio 10.1 El objetivo del flujo **cerr** es que se puedan sacar mensajes relacionados con errores en el programa. Cuando un programa genera un error, es de esperar que se lance inmediatamente por el flujo **cerr**. Lo ideal es que el resultado se obtenga sin ninguna demora, por ejemplo para que el usuario tome alguna decisión sobre qué hacer con ese error. Por tanto, es lógico esperar que **cerr** no tenga buffer, ya que éste podría provocar un retraso en la salida de los mensajes.

Ejercicio 10.2 Considere el problema de escribir en la salida estándar la suma de todos los números enteros que se introducen en la entrada hasta el final de entrada. Escriba un programa que realice esa lectura y como resultado escriba la suma de los datos introducidos o un error en caso de que haya habido una entrada incorrecta.

Solución al ejercicio 10.2 Podemos usar el mismo flujo (**cin**) para comprobar el estado tras una lectura. En concreto, podemos incluir la lectura directamente en la condición del bucle de la siguiente forma:

```
#include <iostream>
using namespace std;
int main()
{
   int a, sum= 0;

   while (cin>>a)
      sum+= a;

   if (cin.eof())
      cout << "Suma total: " << sum << endl;
   else cout << "Error en la entrada" << endl;
}
```

Una vez finalizada la lectura de datos, podemos comprobar si hemos leído correctamente todos los datos de entrada preguntándonos si se alcanzó *EOF*.

Ejercicio 10.3 Escriba una función que, utilizando la función anterior `SiguienteLetra`, extraiga una nueva palabra de la entrada estándar. Una palabra está compuesta por una o varias letras consecutivas. Para resolverlo, la función debe buscar la siguiente letra para crear un **string** con todos los caracteres consecutivos que la forman.

Solución al ejercicio 10.3 La solución consiste en llamar a la función `SiguienteLetra` que descarta todos los caracteres que no son parte de una palabra y una vez situados al comienzo de la palabra encadenar los siguientes caracteres en un objeto **string**. El código puede ser el siguiente:

```
string SiguientePalabra()
{
   string palabra;

   SiguienteLetra(); // Llegamos hasta siguiente palabra

   while (isalpha(cin.peek()))
      palabra+= cin.get();

   return palabra;
}
```

Observe que si el flujo se termina `cin.peek()` no devuelve una letra (devuelve *EOF*) y el bucle termina.

Ejercicio 10.4 Escriba un programa que procesa un archivo que contiene un número indeterminado de líneas; cada una de ellas contiene un entero n seguido de n números reales. Por cada una de estas líneas, el programa escribe en la salida estándar una línea que contiene el entero y la sumatoria de los n números reales. Para resolverlo, haga que el programa lea dos nombres de archivo: el de entrada y el de salida.

Solución al ejercicio 10.4 Para resolver este problema usamos el esquema anterior en el que el resultado de una lectura se puede usar directamente en una condición para comprobar si ha tenido éxito. En concreto, el algoritmo consiste en abrir los dos archivos y mientras *se pueda leer una pareja de datos*, se procesa. El código puede ser el siguiente:

Listado 23 — Programa *sumatorias.cpp*.

```
1  #include <iostream>
2  #include <string>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8     string entrada, salida;
9
10    cout << "Indique nombre del archivo con los datos: ";
11    getline(cin,entrada);
12
13    cout << "Indique nombre del archivo resultado: ";
14    getline(cin,salida);
15
16    ifstream fi;
17    fi.open(entrada);
18    if (!fi)
19        cerr << "Error de apertura de " << entrada << endl;
20    else {
21       ofstream fo;
22       fo.open(salida);
23       if (!fo) {
24          fi.close();
25          cerr << "Error de apertura de " << salida << endl;
26       }
27       else {
28          int n, linea= 0;;
29          while (fi>>n) {
30             linea++;
31             double suma= 0.0;
32             for (int i=0; i<n; ++i) {
33                double dato= 0.0;
34                fi >> dato;
35                suma+= dato;
36             }
37             if (!fi)
38                cerr << "Error en secuencia " << linea << '('
```

```
39                  << n << " datos)"<< endl;
40            else fo << n << ' ' << suma << endl;
41          }
42          fi.close();
43          fo.close();
44        }
45      }
46  }
```

Observe la lectura de la línea 34 que se encuentra en un bucle `for`. No comprobamos si ha tenido éxito hasta acabar el bucle. Es después de terminar el bucle cuando comprobamos si ha habido algún dato erróneo, en cuyo caso presentamos un error con detalles sobre la secuencia que lo ha ocasionado. Recuerde que las lecturas son operaciones sin efecto cuando el flujo se encuentra en modo *fallo*.

Ejercicio 10.5 Simplifique el programa anterior para que sirva como filtro en la línea de órdenes. Para ello, debe evitar la lectura de los nombres de los archivos de entrada. En su lugar, use la entrada y salida estándar.

Solución al ejercicio 10.5 En este caso el problema se simplifica enormemente porque los flujos `cin` y `cout` ya están abiertos y disponibles. La solución es la siguiente:

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

int main()
{
   int n, linea= 0;;
   while (cin>>n) {
      linea++;
      double suma= 0.0;
      for (int i=0; i<n; ++i) {
         double dato= 0.0;
         cin >> dato;
         suma+= dato;
      }
      if (!cin)
         cerr << "Error en secuencia " << linea << '('
              << n << " datos)"<< endl;
      else cout << n << ' ' << suma << endl;
   }
}
```

Ejercicio 10.6 Considere las soluciones de los problemas 10.4 y 10.5. En estos problemas se ha creado un trozo de código muy similar para ficheros y para los flujos estándar. Escriba una función `SimplificarLineas` que tome como entrada dos flujos y que sea válida para los dos programas. Para mostrar su funcionamiento reescriba los programas de ambos ejercicios.

Solución al ejercicio 10.6 La solución al problema consiste en reescribir el código que usa `cin` y `cout` pero en una función con dos parámetros genéricos de tipo `istream` y `ostream`. La solución puede ser la siguiente:

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

void SimplificarLineas (istream& fi, ostream& fo)
{
   int n, linea= 0;;
   while (fi>>n) {
      linea++;
      double suma= 0.0;
      for (int i=0; i<n; ++i) {
         double dato= 0.0;
```

```
            fi >> dato;
            suma+= dato;
         }
         if (!fi)
            cerr << "Error en secuencia " << linea << '('
                 << n << " datos)"<< endl;
         else fo << n << ' ' << suma << endl;
      }
   }

   int main()
   {
      SimplificarLineas(cin,cout);
   }
```

La función **main** ha quedado sólo con una línea. La ventaja de esta función es que podría reutilizarse para resolver el problema 10.4. El código sería:

```
   // ...
   int main()
   {
      string entrada, salida;

      cout << "Indique nombre del archivo con los datos: ";
      getline(cin,entrada);

      cout << "Indique nombre del archivo resultado: ";
      getline(cin,salida);

      ifstream fi;
      fi.open(entrada);
      if (!fi)
         cerr << "Error de apertura de " << entrada << endl;
      else {
         ofstream fo;
         fo.open(salida);
         if (!fo) {
            fi.close();
            cerr << "Error de apertura de " << salida << endl;
         }
         else SimplificarLineas(fi,fo);
      }
   }
```

donde vemos que los parámetros de tipo **ifstream** y **ofstream** son válidos como parámetros de los tipos **istream** y **ostream**, respectivamente.

B Generación de números aleatorios

Introducción 323
El problema 323
Números pseudoaleatorios
Transformación del intervalo 325
Operación módulo
Normalizar a U(0,1)

B.1 Introducción

Muchos programas incluyen la generación automática de números aleatorios. Un ejemplo muy claro es un juego donde se tiene que avanzar con eventos que simulen aleatoriedad a fin de obtener cierta variedad en el desarrollo de distintas partidas.

Por ejemplo, imagine que deseamos crear un programa que avanza en un juego donde se lanza un dado. El programa simulará que obtenemos valores del conjunto $\{1,2,3,4,5,6\}$ hasta el final de la partida. Por ejemplo, podemos obtener:

$$1,1,6,2,4,4,3,6,2,5,5,5,1,\ldots$$

donde puede ver que se obtienen distintos valores que se pueden repetir con un orden indeterminado. Lógicamente, cuando empecemos una nueva partida, los valores que se obtienen deben ser distintos a los de la partida anterior.

En este apéndice vamos a ver cómo podemos hacer que nuestros programas puedan generar dichos valores. Para ello presentamos las funciones que están disponibles en C++ y que ya se podían usar en lenguaje C.

Existen otras utilidades disponibles sólo en lenguaje C++ —desde el estándar C++11— que resultan especialmente interesantes, no sólo porque puede realizar las mismas operaciones, sino porque ofrecen herramientas más avanzadas para crear fácilmente soluciones a problemas más complejos. Si bien no son difíciles de manejar, para entender bien estas nuevas herramientas convendría conocer algunos conceptos de teoría de la probabilidad. No es intención de este documento entrar en esa teoría, por lo que presentaremos las ideas más básicas de una forma muy intuitiva de forma que pueda resolver los problemas más simples y habituales sin gran dificultad.

B.2 El problema

El problema consiste en que un programa tiene que ser capaz de generar una secuencia de números aleatorios tan larga como deseemos. Es decir, estamos interesados en obtener una serie de

valores aleatorios:

$$x_0, x_1, x_2, \ldots, x_i, \ldots$$

Para disponer de una utilidad básica que nos resuelva este problema basta con crear una función que nos devuelve cada uno de esos valores conforme la llamamos. En lenguaje C, y por tanto en C++, se ofrece la solución más simple: crear una función **rand** que devuelve estos valores. Esta función no tiene parámetros. Sólo devuelve un nuevo valor entero aleatorio. Por ejemplo, si deseamos obtener 1000 valores aleatorios podemos escribir:

```
#include <cstdlib> // rand()
//...
for (int i=0; i<1000; ++i)
   cout << rand() << ' ';
```

La función está diseñada para obtener un valor entero en el rango `[0,RAND_MAX]`, donde *RAND_MAX* es una constante predeterminada. Tenga en cuenta que:

- Es necesario incluir el archivo **cstdlib** para disponer de esta función.
- La constante *RAND_MAX* depende de su sistema, ya que no está fijada en el estándar. Sólo sabemos que es un entero positivo, ya que el valor devuelto por **rand** es **int**.
- El entero obtenido puede ser cualquiera del intervalo `[0,RAND_MAX]` con igual probabilidad. Lo que se conoce por una distribución uniforme, ya que cualquier valor de ese intervalo, incluyendo el `0` y *RAND_MAX*, se obtiene con igual probabilidad.
- Es de esperar que el valor de *RAND_MAX* sea bastante alto. De hecho es posible que sea el entero más grande. Por ejemplo, en un sistema con tamaño de palabra pequeño, 2 bytes, podría tener el valor `32767` o si tiene un sistema de 32 bits, es probable que valga `2147483647`.

B.2.1 Números pseudoaleatorios

Como sabemos, el ordenador es una máquina determinista, es decir, que no tiene un comportamiento aleatorio, sino que las salidas son exactas y predecibles a partir de las entradas correspondientes. Por tanto, en principio es imposible conseguir generar una secuencia como la indicada anteriormente, es decir una secuencia de valores realmente aleatorios.

A pesar de ello, y gracias a los estudios estadísticos que se han realizado, existen métodos relativamente simples para obtener una secuencia que *parezca aleatoria*. Efectivamente, en la práctica, la mayoría de los problemas necesitan que los números parezcan aleatorios, es decir, que sean números que analizados estadísticamente podamos decir que se comportan como aleatorios. A éstos los llamaremos *pseudoaleatorios*.

No vamos a entrar en detalles de cómo funciona la función **rand**, pero para entender su uso podemos decir que los números se generan según cierta función interna $f(x)$ de manera que:

$$x_{i+1} = f(x_i)$$

Aunque parezca sorprendente, una idea tan simple y aparentemente tan poco aleatoria nos permite obtener la secuencia deseada. El truco está, lógicamente, en que no necesitamos números aleatorios, sino que basta con que lo parezcan. Ahora bien, todos los números están determinados por la función $f(x)$, pero no sabemos cuánto vale el primer valor con el que comienza la secuencia. Toda la secuencia depende de él, de manera que si fijamos un valor concreto, siempre obtendremos la misma secuencia. Por ello, se denomina *semilla*.

Si nuestros programas usaran siempre la misma semilla, los números pseudoaleatorios que generaríamos serían siempre los mismos. Si queremos que distintas ejecuciones den lugar a distintas secuencias, es necesario cambiar de semilla en cada ejecución. Una forma muy simple de obtener distintas semillas es usar el valor del reloj del sistema. Note que si ejecutamos dos veces distintas un mismo programa, la semilla dependería del momento en que damos la orden de ejecución.

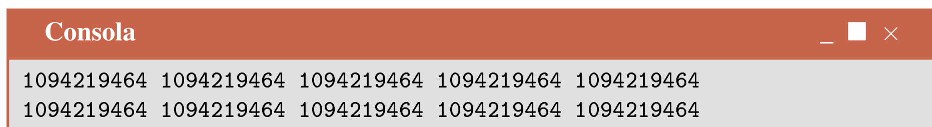
Para fijar una semilla, usamos la función **srand** de **cstdlib**, y para obtener un valor del reloj del sistema la función time del fichero de cabecera **ctime**. Un programa muy simple que genera tres valores aleatorios es el siguiente:

```
#include <cstdlib>   // rand, srand
#include <ctime>     // time
using namespace std;
int main()
{
   srand (time(0));

   int aleatorio1= rand();
   int aleatorio2= rand();
   int aleatorio3= rand();
}
```

Observe que la semilla sólo hay que fijarla al principio del programa, una única vez. A partir de ese momento, se puede usar **rand** tantas veces como se desee para obtener nuevos valores pseudoaleatorios. Por ejemplo, si ejecuta el siguiente programa:

```
for (int i=0;i<10;++i) {
  srand (time(0));
  cout << rand() << ' ';
}
cout << endl;
```

es posible que obtenga 10 valores iguales. O tal vez, si por casualidad el valor de **time** avanza durante el bucle, 2 grupos con los valores iguales. Por ejemplo, una ejecución en mi máquina ha dado lugar a:



El problema es que hemos situado el generador en un valor de semilla idéntico en cada iteración. Situamos el primer valor, generamos el valor aleatorio, y en la siguiente iteración volvemos a situar de nuevo el generador en el primer valor. Recuerde que si **time** devuelve el valor en segundos, las 10 iteraciones del bucle probablemente situarán la misma semilla. Por tanto, el valor que se genera con **rand** será el mismo.

Realmente, **time** es muy poco aleatoria. Sin embargo, nosotros queremos un valor entero distinto cada vez que lancemos el programa, por lo que resolveremos el problema generando una primera semilla al comienzo y limitándonos a usar la función **rand** en el resto del programa. Note que la semilla dependerá del segundo en el que ejecutemos el programa, por lo que resultará en ejecuciones con secuencias aleatorias distintas. Además, el que la secuencia parezca aleatoria está garantizado por la forma en que se comporta la función interna $f(x_i)$, es decir, si queremos que los valores de nuestro programa parezcan aleatorios, deberán corresponder a una secuencia generada con dicha función, una vez establecida la semilla.

B.3 Transformación del intervalo

Normalmente no nos interesará generar un número aleatorio en el intervalo $[0, RAND_MAX]$, especialmente teniendo en cuenta que no conocemos el valor de esa constante. Por ejemplo, si queremos lanzar un dado queremos un valor entero que va del 1 al 6. Para resolver el problema debemos transformar el valor generado al intervalo deseado.

B.3.1 Operación módulo

Si deseamos obtener un valor en un intervalo de enteros pequeño lo más tentador es realizar una operación de módulo con el operador %. Por ejemplo, para generar el valor de un dado podemos escribir:

```
int dado= rand() % 6 + 1;
```

En general, si queremos obtener un valor en el intervalo de enteros [MIN,MAX], ambos inclusive, podríamos calcularlo como:

```
aleatorio= rand() % (MAX−MIN+1) + MIN;
```

Es probable que si consulta código en la red, encuentre muchos ejemplos con líneas de este tipo. Este código se utiliza frecuentemente por su simplicidad, aunque no resulta especialmente recomendable cuando las características de aleatoriedad del generador son importantes para la aplicación que se está desarrollando.

Efectivamente, con la operación módulo estamos haciendo que el valor que usamos en nuestra aplicación dependa especialmente de los bits menos significativos del número generado. Por ejemplo, si realizamos una operación de módulo 100, realmente estamos cogiendo los dos últimos dígitos decimales de los enteros generados. Aunque el resultado final dependerá de la función $f(x_i)$ que se esté usando como motor de generación, es probable que el comportamiento de los bits menos significativos sea menos aleatorio de lo esperado.

B.3.2 Normalizar a U(0,1)

La variable aleatoria uniforme en el intervalo $[0,1]$ —que denotamos $U(0,1)$— es especialmente importante en la teoría de la probabilidad. De hecho, si consulta distintos algoritmos de generación de números aleatorios para distintas distribuciones de probabilidad, encontrará que incluyen la generación de uno o varios valores de esta variable.

De forma simplificada, digamos que generar un valor de una variable $U(0,1)$ es obtener cualquier valor de ese intervalo, teniendo en cuenta que todos los números de ese intervalo tienen igual probabilidad. Con la función **rand** no tenemos más que transformar el valor dividiendo por el máximo *RAND_MAX*. En concreto, podemos usar la siguiente función:

```
inline double Uniforme01()
{
   return rand()/(RAND_MAX+1.0);
}
```

En este código debería observar que sumamos el valor 1.0, que es de tipo **double**. Es interesante notar que la operación *RAND_MAX*+1 es peligrosa. Esta expresión es entera, por lo que la división del valor de **rand** entre este número sería entera y casi seguro que daría el valor cero. Podría pensar que el siguiente código resuelve el problema:

```
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:

```
int Uniforme(int minimo, int maximo)
{
   double u01= rand() / (RAND_MAX+1.0); // Uniforme01
   return minimo + (maximo-minimo+1) * u01;
}
```

C Ingeniería del software

Introducción 329
Actividades en el proceso del software 330
La necesidad de un enfoque de calidad 330
Paradigmas en el desarrollo del software 331
Modelo del sistema 332
Paradigmas de la programación 333
Desarrollo de programas 337

C.1 Introducción

La ingeniería de sistemas basada en ordenador se encarga de todos los aspectos del desarrollo y evolución de sistemas complejos, en donde el software es una parte fundamental.

La ingeniería del software resulta cada vez más importante, como un enfoque disciplinado que se refiere a la producción del software, abarcando desde las primeras etapas de análisis del problema hasta el mantenimiento y evolución una vez que se ha empezado a usar el sistema.

Por ello, es necesario definir el proceso de desarrollo de software a seguir, es decir, el conjunto de actividades y resultados que debemos llevar a cabo para conseguir un producto con unas garantías mínimas de calidad.

En muchos cursos e incluso libros básicos de introducción a la programación se tienden a usar de forma intensiva muchos términos que realmente no se conocen. Frecuentemente, los autores están tan acostumbrados a una serie de conceptos que buena parte de ellos los consideran triviales e incluso casi conocidos. Es importante incluir algunos comentarios sobre éstos, aunque sean en cierta forma divulgativos, para que el lector pueda leer los temas y las referencias bibliográficas de forma más cómoda y sin ambigüedades.

El objetivo de este apéndice es precisamente éste, es decir, realizar una introducción a los conceptos fundamentales de la ingeniería del software. No es objetivo de este libro profundizar en este tema, ya que para eso necesitaríamos no un tema, sino todo un libro.

Es importante que el estudiante empiece a entender y aplicar los conceptos más básicos del desarrollo de software incluso desde los problemas más simples, como los que en este libro se presentan. Desde este punto de vista, es posible que considere que este tema debería ser uno de los primeros, en lugar de un apéndice. Sin embargo, la experiencia indica que son conceptos que el alumno irá asimilando más con la experiencia que con la simple exposición y lectura. En mi opinión, es un tema ideal para leer al final del curso o cerca del final, cuando se planteen proyectos de programación de mayor tamaño.

C.2 Actividades en el proceso del software

Podemos distinguir un conjunto de actividades comunes a distintos modelos de desarrollo de software:

1. *Análisis y especificación*. En esta etapa consideramos el *"qué"*, es decir, se definen los requisitos del sistema, describiendo clara y precisamente el problema y lo que el cliente espera, independientemente de la forma de resolverlo.
2. *Diseño*. En esta etapa consideramos el *"cómo"*, es decir, la arquitectura del sistema, definiendo componentes, interfaces y comportamientos.
3. *Implementación*. En esta etapa tenemos que traducir la arquitectura a un lenguaje de programación, es decir, obtener el código que implementa el software que desarrollamos.
4. *Prueba*. En esta etapa consideramos el refinado de la solución, es decir, las pruebas y modificaciones necesarias para que el sistema funcione correctamente.
5. *Evolución y mantenimiento*. Finalmente, cuando comienza el uso del producto, debemos considerar una etapa en la que se eliminan fallos y se producen modificaciones para ampliar las posibilidades del programa.

Estas etapas se han representado gráficamente en la figura C.1 de forma lineal. Más adelante consideraremos distintos enfoques para llevar a cabo cada una de ellas.

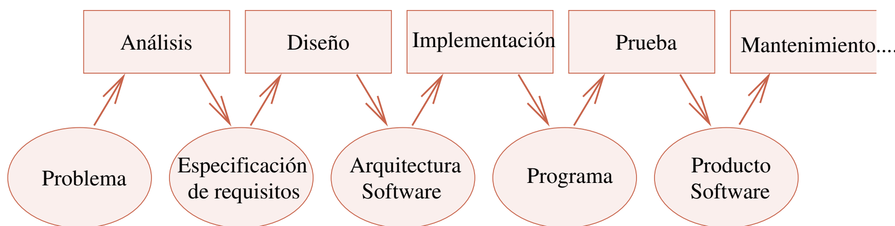


Figura C.1
Actividades en el proceso del software.

C.3 La necesidad de un enfoque de calidad

El objetivo fundamental que seguimos es la obtención de un *buen* software. Lo primero que tenemos que saber es lo que entendemos por un software de calidad. Las características más importantes que podemos desear son:

1. *Fiable*. Un programa es fiable si: por un lado, los resultados obtenidos a partir de él son correctos; por otro, tiene un comportamiento robusto frente a situaciones excepcionales (errores del usuario, errores inesperados, o incluso errores de programación).
2. *Eficiente*. El programa debe usar la menor cantidad de recursos posible, tales como tiempo de CPU, memoria principal, memoria externa, tamaño de datos transmitidos, etc.
3. *Mantenible*. Cuando se desarrolla un programa, el proceso no finaliza cuando se entrega al cliente, sino que debemos considerar un intervalo de vida indefinido. Una vez que se entrega, puede ser necesario modificarlo para corregir errores, ampliarlo para darle más funcionalidad, o incluso reutilizar alguna parte para el desarrollo de otros proyectos.
4. *De fácil uso*. El programa debe ser útil y, por tanto, se debe intentar facilitar su uso por parte de personas de diferentes capacidades.

Es habitual pensar que un programa necesita funcionar únicamente con unas restricciones mínimas, abandonando su desarrollo una vez que se comercializa. Sin embargo, es importante

considerar que el tiempo de vida es indefinido y que, una vez se haya obtenido una primera versión operativa, será necesario realizar cambios para corregir errores o ampliar su funcionalidad.

De esta forma, su comportamiento es muy distinto al hardware[1], ya que éste puede registrar errores en sus primeros momentos de vida, pero después de este intervalo de tiempo, es de esperar que haya un número de errores pequeño hasta que se estropea.

Por otro lado, el software no se estropea, sino que se deteriora. Como consecuencia de su evolución, el software es cada vez más difícil y costoso de mantener. Para ilustrar esta idea, presentamos la figura C.2, donde se muestra el nivel de calidad del software a lo largo del tiempo. Este nivel representa las características que antes indicábamos. Considere por ejemplo el número de fallos, los recursos que necesita, o la facilidad de mantenimiento.

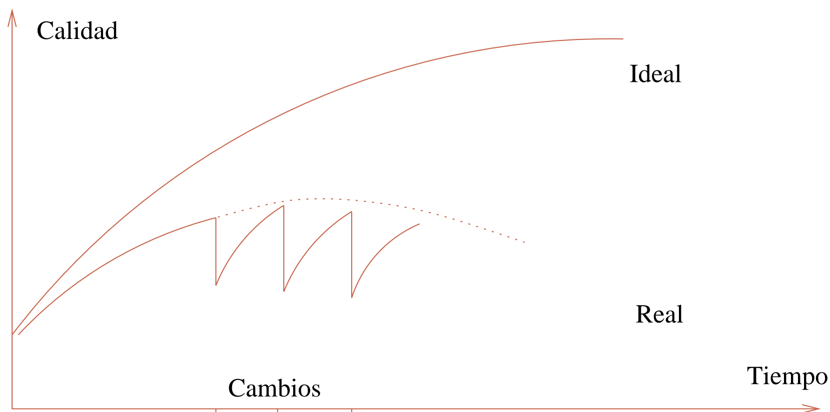


Figura C.2
El software se deteriora.

Observe que hemos presentado un conjunto de cambios que afectan claramente a la calidad del software y a la curva real que indica la forma en que se deteriora el software.

Tener en cuenta este comportamiento es fundamental. Considere, por ejemplo, sistemas cuyo tiempo de vida es muy largo. Para éstos, el costo de la etapa de evolución y mantenimiento puede llegar al 75 % del costo total. Cuanto mayor sea la calidad del software, más se acercará la curva real a la ideal, y menor será el costo y los efectos de los cambios.

Otro ejemplo puede ser el de sistemas en los que las etapas de prueba sean muy largas (programas que se pueden usar sobre múltiples configuraciones) y para los que será necesario realizar modificaciones antes de que sean lanzados al mercado. La etapa de pruebas puede suponer más de la mitad del gasto, y de nuevo, es necesario que los cambios afecten, en la menor medida posible, a la calidad y el costo.

No debemos olvidar, por tanto, que la calidad del software afecta de forma directa al costo del desarrollo.

C.4 Paradigmas en el desarrollo del software

En la historia de la ingeniería del software se han desarrollado distintos paradigmas para obtener una solución óptima en términos de calidad y costo. Podemos considerar:

[1] Hardware en el sentido más estricto. Recuerde que la mayoría de los dispositivos son una combinación de hardware y software.

1. *Lineal secuencial (Cascada)*. Considera las distintas actividades —análisis, diseño, implementación y prueba— y las lleva a cabo de forma secuencial, cerrando cada una de ellas antes de comenzar la siguiente. En la práctica, son necesarias las vueltas atrás, que se intentan evitar a causa de su elevado coste.
2. *Evolutivos*. Se mezclan las distintas actividades de forma que se obtienen distintas versiones incrementales del sistema. Las distintas versiones se van refinando hasta obtener el resultado deseado.
3. *Ensamblaje de componentes*. Considera la existencia de algunas partes necesarias para el sistema —incluso la compra a otras empresas— de forma que sólo es necesaria su integración en el sistema que se desarrolla. Además, considera que otras partes también se desarrollan como componentes que podrán reusarse en otros sistemas. Esta forma de trabajo toma un creciente interés, especialmente considerando el éxito en otros campos (por ej. la rápida y eficaz evolución de sistemas hardware, que se construyen mediante ensamblaje de componentes).
4. *Métodos formales*. Se basa en la obtención de un programa a partir de una especificación matemática y formal de un sistema. De esta forma, la verificación del sistema se lleva a cabo por medio de argumentos matemáticos que la garantizan.

Se puede usar una mezcla, especialmente en sistemas complejos. Por ejemplo, para obtener la especificación, un método evolutivo; cuando se tienen los requisitos, se desarrolla usando un método lineal secuencial, reusando algunos componentes desarrollados en otros sistemas, o incluso métodos formales para un subsistema para el que queremos garantizar que corresponde a la especificación.

C.5 Modelo del sistema

El problema de llegar al producto final desde la descripción inicial del sistema —normalmente en lenguaje natural— no es un problema trivial.

Desde la etapa de análisis, debemos empezar a definir un modelo del sistema preciso y sin ambigüedades, que represente el qué se quiere obtener y que, después de la etapa de diseño, también contendrá la forma de llevarla a cabo. Este modelo del sistema es muy importante para que los ingenieros puedan obtener un producto que se comporte conforme a las especificaciones iniciales.

Para representar el sistema podemos utilizar diferentes modelos:

1. *Orientado a componentes*. Se define y diseña un sistema o subsistema, como un conjunto de componentes que interaccionan. El objetivo fundamental en este tipo de modelo es poder reutilizar otros componentes ya desarrollados o poder disponer de los nuevos componentes para futuros programas.
2. *Orientado al Flujo de datos*. Se centra en la descripción de los datos que se manejan y cómo van siendo procesados hasta obtener la salida deseada. Por ejemplo, podemos describir un sistema de cálculo numérico que resuelve un sistema de ecuaciones como una función que obtiene los datos de entrada; éstos a su vez se procesan para determinar si tiene solución; si la tiene, se procesan obteniendo un conjunto de resultados, etc.
3. *Orientado a objetos*. Se centra en la descripción de los objetos que componen el sistema y la forma en que interaccionan. En muchos casos, es más fácil describir un sistema por medio de los objetos que aparecen y la forma en que se comunican. Para estos casos, un modelo orientado a objetos será mucho más simple de desarrollar. Por ejemplo, si queremos un programa que simule el comportamiento de un cruce de semáforos, una descripción muy cercana a la realidad consiste en ir describiendo que existe un cruce, una serie de colas de coches, cuando un semáforo cambia de color indica al otro que también cambie, etc. En definitiva, una descripción de los objetos, su comportamiento, y su interacción.

4. *De entidad-relación*. Se centra en la descripción de las entidades y sus relaciones. Se utilizan normalmente en la descripción de sistemas de información en los que se almacenan datos sobre entidades que están relacionadas, de forma que no sólo se almacenan los datos, sino también sus relaciones. Por ejemplo, podemos almacenar datos sobre personas y sus domicilios, dos entidades relacionadas: una persona puede tener cero o más domicilios; o un domicilio puede corresponder a varias personas.

C.6 Paradigmas de la programación

De la misma forma que han ido apareciendo nuevas propuestas para la ingeniería del software, con el tiempo, los lenguajes de programación también han ido evolucionando y soportando nuevos paradigmas para enfrentarse a los nuevos problemas. En esta sección comentamos brevemente en qué consisten y cómo el lenguaje C++ ofrece soporte para su utilización. Es necesario indicar que aunque los presentamos en una sección independiente, las evoluciones de la ingeniería del software y los lenguajes de programación están estrechamente relacionadas. Muchas discusiones tratan problemas comunes y persiguen las mismas soluciones; si queremos expresar la solución con un lenguaje de programación, será más fácil si éste es capaz de adaptarse a la forma en que se describe. Consideraremos los siguientes paradigmas:

1. Programación procedimental.
2. Programación modular.
3. Abstracción de datos.
4. Programación orientada a objetos.
5. Programación genérica.
6. Metaprogramación.
7. Programación funcional.

C.6.1 Programación procedimental

Este paradigma se basa en la creación de funciones o procedimientos como nuevas operaciones más potentes, a un nivel de abstracción mayor, más cercanas a la solución del problema.

Cuando los programas empiezan a ser más complejos, no es sencillo resolver el problema como una única función. En el desarrollo del software se estudian las tareas que necesitamos para resolver el problema y se implementan los algoritmos correspondientes como funciones o procedimientos.

Estas funciones ocultan los detalles del algoritmo que implementan y ofrecen la posibilidad de utilizarlo en base a una interfaz bien definida. El lenguaje C++ soporta este paradigma mediante la definición de funciones y los mecanismos de paso de parámetros y devolución de resultados (véase capítulo 5).

Por ejemplo, podemos definir una función que contiene todos los pasos necesarios para obtener la raíz cuadrada de un parámetro de doble precisión en coma flotante.

```
1 double raiz_cuadrada (double r)
2 {
3   // algoritmo para calcular la raíz cuadrada de r
4 }
```

Una llamada a la operación sólo necesita conocer la interfaz, obviando los detalles internos que la implementan.

Con respecto al diseño procedimental, es importante destacar que Dijkstra y otros autores propusieron utilizar un conjunto de construcciones lógicas limitadas —concretamente *la secuencia, la condición* y la *repetición*— para desarrollar un programa. De esta forma, una función sólo tiene una entrada y una salida. Además, cada una de las tareas a realizar puede anidarse, es decir, una

tarea puede ser otro conjunto de tareas y estructuras de control con una única entrada y salida (podemos considerar este anidamiento como la estructura del programa en procedimientos).

Finalmente, conviene matizar —como indica Pressman([22]— que un empleo dogmático de estas construcciones puede implicar una pérdida de eficiencia y complicar el código. Una propuesta —que asumiremos— es la de saltarnos estas construcciones estructuradas de una manera controlada para no romper el espíritu de la programación estructurada. Por ejemplo, para salir de una función cuando estamos en el interior de un conjunto de bucles anidados, aceptaremos una salida directa —sentencia `return`— si con ello mejoramos la eficiencia y claridad del código.

C.6.2 Programación modular

Cuando el problema se hace muy complejo, es recomendable dividirlo en subproblemas para que sea más fácil de resolver. Con esta metodología, por un lado podemos enfrentarnos a la búsqueda de soluciones para estos subproblemas; por otro, podemos unirlas para llegar a la solución global que deseamos.

La programación modular se basa en la creación de unidades —denominadas módulos— que corresponden a un conjunto de procedimientos y de datos relacionados entre sí. Note que una función o procedimiento lo podemos considerar un módulo simple.

Cuando se desarrolla un módulo, es necesario también definir la interfaz para acceder a su funcionalidad. Lógicamente, todos los detalles, funciones o datos, que no sean necesarios para la interfaz, se pueden obviar a la hora de utilizar dicho módulo. Por consiguiente, en un módulo podemos encontrar datos y funciones ocultas, en el sentido de que no se utilizan más que para el funcionamiento interno del módulo.

Respecto a la programación modular, podemos destacar que el lenguaje C++ ofrece:

- *Compilación separada*. Permite dividir el código en distintos ficheros. De esta forma, podemos agrupar los datos y procedimientos asociados según su almacenamiento físico. Podemos considerar dos tipos de ficheros[2], los que almacenan la interfaz o de cabecera —normalmente con extensión *.h*— y otros con las definiciones —normalmente con extensión *.cpp*—.
- *Espacios de nombres*. Permiten indicar que un conjunto de identificadores —datos, funciones, nombres de tipos, etc.— están relacionados, asignándoles un espacio de nombres (`namespace`). Posteriormente, para utilizar alguno de ellos necesitamos especificar el espacio al que pertenece, anteponiendo el nombre del `namespace` con los caracteres "`::`". Por ejemplo, para usar `cout`, necesitamos indicar que está en `std`, por tanto, en nuestro código tendremos que poner `std::cout` (para más detalles, véase 11.5).

C.6.3 Abstracción de datos

Hasta ahora, nos hemos centrado en los algoritmos y en cómo nos enfrentamos a su creciente dificultad. Sin embargo, cuando los problemas a resolver son más complicados, las estructuras de datos también son más complejas.

La solución a un problema puede implicar el manejo de una compleja estructura de datos que representa algún tipo de información en nuestro problema. Si al algoritmo añadimos la posible complejidad de los detalles de esta estructura, resultará más difícil de resolver, por lo que será más difícil obtener una buena solución. Por ejemplo, imagine el problema de mantenimiento si en el código se mezclan detalles de algoritmos con detalles de una compleja estructura de datos.

Para aliviar estos problemas, se propone el uso de la abstracción de datos. La idea básica consiste en desarrollar un módulo que resuelva los problemas de manejo de esa estructura, ocultando todos los detalles innecesarios por medio de una interfaz sencilla que ofrezca la funcionalidad que necesitamos para nuestros algoritmos.

[2]Para más detalles, véase capítulo 11.

La forma más cómoda para usar la abstracción de datos es la definición de un nuevo tipo de dato, con una interfaz adecuada y que tenga un comportamiento idéntico a los tipos predefinidos del lenguaje[3].

Por ejemplo, si desarrollamos un programa para tratamiento matemático y necesitamos manejar polinomios, puede resultar más engorroso arrastrar todos los detalles de su representación a lo largo de los algoritmos. Es más sencillo disponer de un tipo `Polinomio` que nos ofrezca la funcionalidad que deseamos, por ejemplo, sumas, productos, evaluación, etc. en una interfaz bien definida, para la que no es necesario conocer cómo se ha representado en una estructura de datos.

El lenguaje C++ nos ofrece la posibilidad de definición de nuevos tipos de datos como una *clase*, y los mecanismos necesarios para que sea cómoda su utilización, tales como los constructores para poder inicializar las variables, métodos para implementar la forma de copiar un objeto (por ejemplo, para llevar a cabo un paso por valor), sobrecarga de operadores, etc.

C.6.4 Programación orientada a objetos

En las secciones anteriores, la discusión se ha basado en un modelo en el que los datos y los algoritmos se manejan de forma independiente, por un lado tenemos los datos que trata el problema y, por otro, los algoritmos que los transforman. En este sentido, podemos decir que eran soluciones orientadas al flujo de datos.

Para poder implementar las soluciones de los modelos orientados a objetos, los lenguajes soportan este paradigma de programación basándose en los siguientes principios:

1. *Encapsulamiento*.
2. *Herencia*.
3. *Polimorfismo*.

El encapsulamiento nos permite combinar los datos y las operaciones en un mismo objeto, de forma que éste ofrece una interfaz de comunicación para relacionarse con otros objetos, haciendo inaccesibles todos los detalles internos de su funcionamiento.

La herencia es un mecanismo que permite indicar que una clase de objetos hereda las propiedades de otra. Por ejemplo, podemos tener un sistema de gestión en el que hay que procesar los datos de los empleados. Para ello, podemos crear la clase de objetos `Empleado`. En ésta, se resuelven los problemas comunes a cualquier empleado. Por ejemplo, se almacenan datos personales, tiene operaciones para calcular la antigüedad del empleado, etc.

Si queremos añadir un nuevo tipo de objeto, por ejemplo `Directivo`, no es necesario indicar que para esta clase de objetos hay que almacenar datos personales, fecha de incorporación, etc. Sólo tenemos que indicar en el programa que hereda las propiedades de un empleado. Con esa operación tan simple, ya disponemos de ese nuevo objeto con todos los datos y operaciones que habíamos indicado para `Empleado`. Sólo debemos modificarlo indicando las diferencias. Por ejemplo, añadimos un dato más que almacena el departamento del que es responsable.

El polimorfismo permite que un objeto determine, en tiempo de ejecución, la operación a realizar[4]. Por ejemplo, supongamos un sistema de diseño gráfico para el que tenemos un conjunto de objetos, como *circunferencia*, *rectángulo* y *triángulo*. Todos ellos tienen una superficie, y por tanto, pueden tener una operación para calcular y devolver su valor. Imaginemos que queremos una función para escribir en pantalla la superficie de una *figura*, siendo ésta cualquiera de las anteriores. El polimorfismo permite definir una única función que, tomando como parámetro una *figura*, escribe en pantalla su superficie. Por supuesto, en tiempo de ejecución, la función llamará a la operación correspondiente, dependiendo del tipo de figura.

[3] Por ejemplo, seguro que al lector no le preocupa excesivamente la dificultad de la implementación interna del tipo de dato **`double`** en C++.

[4] En la literatura podemos encontrar otros mecanismos —tales como sobrecarga y plantillas— clasificados como polimorfismo. Sin embargo, son características que se resuelven en tiempo de compilación.

El lenguaje C++ ofrece mecanismos para soportar este paradigma. El encapsulamiento por medio de clases, la herencia con la declaración de clases base y derivadas, y el polimorfismo mediante herencia y funciones virtuales. En Shalloway[29] puede consultar el primer capítulo para una discusión más amplia e interesante sobre este paradigma.

C.6.5 Programación genérica

En programación existen múltiples algoritmos que se repiten continuamente y que, por tanto, es necesario implementar para distintos tipos de datos o entornos de programación. Por ejemplo, en un programa podemos necesitar un algoritmo de ordenación de números enteros y de números en coma flotante. El algoritmo es idéntico, aunque en muchos lenguajes será necesario disponer de dos funciones para resolverlo.

La programación genérica hace referencia a la implementación de algoritmos para aplicarlos en múltiples ocasiones. En lugar de construir un algoritmo particular para cada caso, es posible especificar un algoritmo general compuesto de todos los pasos comunes a cada una de las implementaciones, y que el lenguaje nos permitirá aplicar para todos los casos. Como es de esperar, el único requisito es que cada paso del algoritmo general sea compatible con cada tipo particular.

La programación genérica es una técnica para la *reutilización*, que como sabemos, es un concepto muy importante en la mejora de la productividad y calidad del software.

El lenguaje C++ nos permite disponer de esta herramienta[5] haciendo uso de plantillas (*templates*).

C.6.6 Metaprogramación

El incremento del nivel de abstracción es una característica que va en aumento cuando el sistema se hace más complejo y queremos manejar herramientas muy potentes y versátiles de una forma relativamente simple. Como programador, el concepto de abstracción le acompañará a lo largo de toda su carrera. Si ya ha entendido algo sobre él con los paradigmas que se exponen en las secciones anteriores —donde se habla de abstraer estructuras de datos, algoritmos, tipos, etc.— en esta sección vamos un paso más allá: abstracción en el mismo programa. La metaprogramación hace referencia a la capacidad de generar un programa, es decir, programamos un algoritmo que genera un programa.

La metaprogramación no es un tema cerrado, bien establecido, sino que está en pleno desarrollo, ya que los lenguajes se actualizan e incorporan cada vez más posibilidades. No es fácil establecer un modelo general para describirlo. En principio, podemos hablar de características más o menos dinámicas, dependiendo de si la metaprogramación se ejecuta en tiempo de compilación o ejecución.

En la bibliografía, encontrará los términos *introspección* y *reflexión* que se asocia a la capacidad que tiene un programa para preguntarse sobre sus objetos o para manipularlos, respectivamente. Estas características permiten a un programa cambiar en tiempo de ejecución, por tanto, es una forma de metaprogramación en tiempo de ejecución.

Por otro lado, también es posible crear programas para que generen código en tiempo de compilación, como ocurre en el lenguaje C++. Este modelo es especialmente interesante si queremos generar código muy eficiente, ya que todo el trabajo que realiza el compilador para generar un nuevo programa no consumirá ningún tiempo en ejecución. La plantillas —*templates*— de C++ nos permiten obtener esta capacidad.

Note que las plantillas también se han comentado en la sección de programación genérica, ya que realmente éste tipo de programación se podría considerar una forma básica de metaprogramación. Sin embargo, es necesario considerarla explícitamente, dada su importancia. De hecho, los nuevos

[5] Consulte Alexandrescu[2] para una discusión mucho más amplia.

estándares del lenguaje han avanzado en gran medida en este aspecto[6], haciendo del lenguaje C++ y su sistema de plantillas una potente herramienta de metaprogramación.

C.6.7 Programación funcional

La *programación funcional* es un paradigma de programación diferenciado de la *programación imperativa*. Ésta corresponde a un tipo de programación más habitual, de hecho este libro es de programación *imperativa*. Erróneamente, ha sido considerada en muchos casos una programación más marginal, menos práctica, incluso prescindible. Sin embargo, un ingeniero informático debería conocerla; aunque no la practique, debería ser parte de su formación, lo que le permite tener una visión mucho más amplia de los problemas, incluyendo una mejor capacidad de abstracción.

La programación funcional es tan antigua como la programación imperativa. Tiene sus raíces en el *cálculo lambda*, desarrollado en los años 30 y que demostró ser equivalente a las máquinas de Turing. Históricamente, ha dado lugar a distintos lenguajes que se han especializado en proponer una forma de programación funcional en contraste a la imperativa. Algunos lenguajes de este tipo son *Haskell*, *Scheme* o *Erlang*. Son de menor uso, pero cada día se hacen más actuales cuando se reconocen las características positivas de este paradigma de programación. Características como la fiabilidad, la facilidad de depuración, de realización de pruebas o de ejecución concurrente.

El desarrollo de los lenguajes no ignora este hecho, considerando las características de la programación funcional como una forma de programación con la que ampliar y mejorar sus capacidades. De esta forma, puede encontrar lenguajes que siendo funcionales consideran la posibilidad de incluir características propias de los imperativos y a la inversa.

El lenguaje C++ es uno de estos casos, donde en los últimos estándares —especialmente desde el C++11— se han empezado a incluir características que se pueden considerar propias de un lenguaje funcional. Por un lado, ha hecho que las funciones sean un objeto de primer nivel, de forma que el lenguaje es capaz de crear objetos función[7] que se pasan o devuelven de funciones, o ha incorporado el *cálculo lambda*. Por otro lado, la forma de metaprogramación con plantillas de C++ tiene características como la inexistencia de variables mutables o el flujo de control basado en la recursión, lo que establece una forma de implementación propia de la programación funcional.

C.7 Desarrollo de programas

En la actualidad podemos considerar, fundamentalmente, dos escuelas a la hora de abordar el problema de enseñar a programar, siguiendo respectivamente dos formas de desarrollo de software:

1. *Análisis y diseño estructurado*. El sistema se modeliza con un enfoque orientado al flujo de datos. Se pueden aplican los paradigmas de programación procedimental, modular, abstracción de datos, e incluso de programación genérica, para desarrollar el software.
2. *Análisis y diseño orientado a objetos*. El sistema se modeliza con un enfoque orientado a objetos y se utiliza un paradigma de programación orientado a objetos. En C++, dada su naturaleza multiparadigma, se usa un enfoque estructurado para definir las operaciones de los objetos.

En este libro, nosotros consideramos el enfoque más clásico de análisis y diseño estructurado, en un desarrollo del software de tipo *secuencial lineal*.

[6]No sólo eso, sino que algunos aspectos del lenguaje se ven influenciados por la adaptación del lenguaje a esta nueva capacidad. Además de ampliarlo, lo remodelan para adaptarse a estas nuevas capacidades.

[7]Realmente deberíamos usar un término más propio de C++: entidad *"llamable"*.

C.7.1 Análisis

Recordemos que, en el análisis, consideramos el "qué", es decir, se definen los requisitos del sistema, describiendo clara y precisamente el problema y lo que el cliente espera, independientemente de la forma de resolverlo.

Las mayores dificultades surgen de la comunicación con el cliente, así como de la naturaleza *incompleta* y *no monótona* de los requisitos:

- *Incompleta*. El cliente no es analista de sistemas y nos intentará informar sobre todo lo que deseemos, aunque es probable que haya información que se le olvide o no considere importante a pesar de serlo. El ingeniero es el responsable de obtener toda la información posible. Recordemos que, si hay algún aspecto que no se tiene en cuenta y surge en una etapa avanzada del proyecto, puede necesitar una vuelta atrás con un alto coste.
- *No monótona*. La información que nos da el cliente puede contener contradicciones. Es posible obtener algunas conclusiones a partir de una información ya disponible y, tras una nueva entrevista, obtener nuevas características que invalidan alguna conclusión anterior.

Algunos puntos prácticos a tener en cuenta son:

- No se deben presentar informaciones de diseño o implementación, aunque pueden aparecer detalles en caso de que el cliente o la interacción con otros sistemas lo requieran.
- Una herramienta que puede resultar útil en el análisis es el uso de *guiones*, es decir, secuencias de pasos para conseguir un objetivo con el sistema. Con ello, se pueden establecer las primeras pruebas para luego probar el sistema.
- Podemos aplicar la regla del 80/20, es decir, el 80 % de la funcionalidad se consigue con el 20 % del trabajo. Se pueden determinar guiones para cubrir un 90 % del sistema.
- Además de contemplar las situaciones de funcionamiento normal del software, es necesario determinar el comportamiento del sistema en casos excepcionales, tales como errores del usuario o fallos inesperados.
- Un aspecto especialmente interesante es el de estudiar posibles modificaciones. Recordemos que una característica muy importante del software es el mantenimiento. El estudio de posibles cambios o futuras mejoras puede sugerirnos algunas decisiones de diseño que mejoren la calidad del resultado.

El resultado de la etapa de análisis es un documento de requisitos que modela el sistema a desarrollar. A partir de este documento se pueden desarrollar las siguientes etapas.

C.7.2 Diseño

Recordemos que en el diseño consideramos el "cómo", es decir, la arquitectura del sistema, definiendo componentes, interfaces y comportamientos. El diseño es fundamental para establecer la calidad del programa que obtenemos.

Algunas tareas que tendremos que llevar a cabo son:

- *Diseño arquitectónico*. Se refiere a los elementos estructurales del programa, es decir, subsistemas, componentes, o módulos que lo componen, así como a la relación que existe entre ellos.
- *Diseño de interfaces*. Abarca el diseño de las interfaces de usuario, con otros sistemas, así como las interfaces internas del programa.
- *Diseño de estructuras de datos*. Transforma los datos que maneja el programa en las estructuras de datos necesarias para representarlos y manejarlos eficazmente.
- *Diseño de algoritmos*. Se diseñan los algoritmos necesarios para el proceso de la información.

El resultado será un documento de especificación de cada uno de esos diseños.

Abstracción y ocultamiento de información

La abstracción permite estudiar complejos sistemas usando un método jerárquico en sucesivos niveles de detalle. De esta forma, un sistema se puede descomponer en distintos módulos. Dado que nuestra intención es manejar, en un momento dado, el menor número de características, cada módulo deberá exportar la menor cantidad de información, siempre que con ello permita al resto de los módulos realizar su labor de forma sencilla y correcta.

Esto nos lleva a diferenciar dos partes muy importantes cuando se desarrolla un módulo:

1. *Interfaz*. Todos los aspectos que son visibles para los demás módulos. El objetivo es que sean pocos. Es la única información necesaria y disponible para un correcto uso del módulo.
2. *Parte interna*. Todos los detalles que no son visibles para los demás módulos. Normalmente serán numerosos, pero no es necesario conocerlos para utilizar el módulo. Note que el desarrollo de esta parte puede llevarse a cabo mediante la aplicación, de nuevo, de una descomposición modular, llevándonos finalmente a una estructura de múltiples niveles de abstracción.

Por tanto, para que la descomposición sea útil, es necesario un proceso de *ocultamiento de información* de forma que el razonamiento, en la construcción del sistema a partir de sus módulos, utilice el mínimo de características relevantes —parte pública— ocultando las irrelevantes —parte privada—. Nótese que cuanto mayor es el grado de ocultamiento de información, menor será la información que tendremos que manejar: se disminuye la complejidad como consecuencia de haber minimizado la interdependencia entre módulos.

Es importante destacar que este ocultamiento no hace referencia a que el usuario del módulo no debe conocer ningún detalle interno por motivos relacionados con la privacidad del código, los derechos de copia o la explotación comercial[8]. Cuando usamos el término de ocultamiento en este documento nos referimos a la necesidad de que el usuario del módulo utilice el mínimo de características para el desarrollo de sus programas. Nuestras aportaciones para que le sea más difícil acceder a los detalles internos facilitarán un uso correcto del software.

Descomposición modular

La programación modular constituye una metodología eficaz para abordar problemas de cualquier tamaño. Sin embargo, es necesario considerar detenidamente las propiedades que deben cumplir para que la descomposición sea útil:

1. La solución de los subproblemas se puede combinar para resolver el problema original.
2. La solución de cada subproblema debe ser general para obtener módulos que, siendo igualmente útiles para el problema a resolver, puedan adaptarse a futuras modificaciones o incluso para otros problemas.
3. Las conexiones de cada módulo con el resto del programa deben ser mínimas, tanto en número como en complejidad, para obtener módulos más independientes. Es decir, tiene que existir un bajo nivel de *acoplamiento*. Algunos ejemplos para mostrar su importancia son:
    - Si los módulos tienen muchas interdependencias, modificar un módulo será complejo, ya que es más fácil provocar errores que se propagan a otros módulos.
    - La interdependencia hace más difícil diseñar un conjunto de pruebas para un módulo, pues tenemos que tener en cuenta los demás.
    - Encontrar un error puede ser más complejo, ya que puede fallar un módulo, pero tal vez sea consecuencia del error en otro relacionado.

    Los niveles más altos de acoplamiento y algunas indicaciones sobre su tratamiento son:
    - La relación del programa con elementos externos. Por ejemplo, tal vez necesitemos adaptar el programa para que se comunique con otro utilizando cierto protocolo. Por

[8] Aunque en la práctica, es la forma de proteger el programa fuente, ofreciendo al usuario todas las posibilidades sin que conozca los detalles del desarrollo.

supuesto, no podemos evitar que nuestro programa dependa de esta especificación, pero sí podemos aislar partes de nuestro código para que la dependencia sólo sea parcial. Imagine que utilizamos detalles de ese protocolo a lo largo de todos nuestros módulos. Si en un futuro queremos adaptarlo a un nuevo protocolo, sería necesario cambiarlo todo.

- El uso de datos globales, es decir, cuando varios módulos comparten datos imponiendo, por tanto, una relación entre ellos. Localizar un error puede ser costoso, porque si proviene de esos datos, el que un módulo falle puede ser un efecto colateral de alguna operación en otro módulo. Por otro lado, si queremos modificar un módulo, debemos tener en cuenta cómo afectará a los demás. El uso de variables globales no es un error, aunque debería tenerse cuidado para que se usaran en casos claros, de manera controlada y afectando al menor número posible de módulos.
- Un módulo modifica o usa datos que se mantienen en otro módulo. Este grado de acoplamiento se puede considerar máximo. No sólo tenemos que evitarlo sino que podríamos considerarlo un error grave. Por ejemplo, si un módulo mantiene una estructura de datos interna y ofrece una interfaz para acceder a ella, es un error acceder directamente a estos datos desde otro módulo. Casi estamos pidiendo que se produzca un error inesperado. Imagine que se toma una decisión de modificar el módulo que mantiene la estructura interna, decisión fácil de tomar pues modificar los detalles internos de un módulo puede tener un riesgo bajo; en este caso, cualquier código que acceda a los detalles internos quedará invalidado.

4. Cada módulo lleva a cabo una tarea bien definida en el nivel de detalle correspondiente para facilitar su solución y la independencia con el resto del programa. Es decir, tiene que existir un alto nivel de *cohesión*. Por ejemplo, imaginemos que un módulo resuelve tareas de entrada de datos, cálculos y modificación de estructuras de datos —tareas que son independientes—. En este caso, diremos que tiene muy poca cohesión. Note que será más difícil leer el código, mantenerlo (modificar una de esas operaciones puede afectar a las demás), diseñar pruebas para probar el módulo, reutilizarlo en otro programa, etc.
5. Mayor número de módulos no implica una mejor solución. El coste de desarrollo es la suma del coste de cada módulo más el coste de la integración de éstos. Así, el número óptimo de módulos no puede ser ni muy pequeño ni demasiado grande.

Una correcta aplicación de esta metodología facilitará en gran medida la resolución del problema. Concretamente, podemos destacar varios aspectos que ponen de relieve los beneficios que podemos obtener:

- Facilita el *desarrollo de programa*. La independencia entre las distintas partes del problema permite que varios programadores puedan trabajar en equipo para resolverlo, minimizando las labores de coordinación necesarias para que se obtenga una solución correcta.
- Facilita el *mantenimiento*:
  - Facilita los *cambios y mejoras*. Este tipo de descomposición permite que la mayor parte de los cambios y mejoras se apliquen sobre un módulo o un número pequeño de ellos.
  - Facilita la *prueba y depurado del software*. El conjunto de pruebas para probar el software es más sencillo, pues se puede analizar cada parte de manera independiente. Además, la detección y eliminación de errores se limita a analizar un pequeño trozo del programa.
- Facilita la *productividad*, evitando que un problema se resuelva múltiples veces:
  - Facilita la *eliminación de redundancias*. Es más fácil identificar los subproblemas que deben resolverse en distintas partes del programa.
  - Facilita la *reutilización del software*. El resultado no es una solución a un problema, sino un conjunto de soluciones a múltiples subproblemas que se combinan en el programa

que deseamos. Algunas de estas soluciones pueden ser necesarias para resolver otros problemas.

C.7.3 Implementación

La etapa de implementación es relativamente sencilla, ya que está muy determinada por el diseño que se ha obtenido. Sin embargo, es importante establecer algunos criterios para un mejor resultado. Algunos aspectos que tendremos en cuenta son:

- Aunque hay que seguir las indicaciones de diseño, todavía es posible alguna libertad de innovación y flexibilidad. Por ejemplo, se puede desarrollar un componente de forma más general para facilitar la reusabilidad, o se puede realizar algún cambio que mejore la eficiencia del código.
- Es importante seguir algún estilo de codificación que mejore la legibilidad. Por ejemplo, podemos separar los nombres compuestos con un símbolo especial, nombrar los tipos de datos con la primera letra mayúscula, nombrar las macros con todas las letras en mayúscula, etc.
- Algunas veces es necesario escoger entre legibilidad y eficiencia. Es posible utilizar una codificación mucho más críptica para obtener mejoras en tiempo de ejecución. En general, si una codificación legible está en los valores de eficiencia que nuestros requisitos exigen, es innecesario hacer que la lectura sea más difícil.
- Es recomendable acompañar el código con comentarios puntuales sobre algunos aspectos importantes para facilitar la comprensión. Esto no significa que la calidad aumenta proporcionalmente al número de comentarios. Por ejemplo, hay que evitar los comentarios redundantes.
- Para facilitar el mantenimiento, podemos incluir información de diseño en el código. Así, es más sencillo modificar un fichero fuente al tener disponible diseño, implementación y su relación.

C.7.4 Pruebas

Las actividades referidas a las pruebas se suelen situar después del análisis y diseño. Esta situación es consecuencia de la idea de que el programa se debe probar antes de ser entregado al cliente. Sin embargo, las actividades relacionadas con las pruebas se deben llevar a cabo a lo largo de todo el desarrollo, desde las primeras etapas, en donde ya se diseñan pruebas y se realizan revisiones, hasta las pruebas finales del producto.

Para probar un módulo que hemos implementado, podemos considerar dos tipos de pruebas:

1. Pruebas de *caja blanca*. Se conocen los detalles de implementación internos, por tanto, se utiliza el conocimiento sobre la estructura del módulo para generar los casos de prueba.
   Por ejemplo, cuando un programador desarrolla un trozo de código normalmente piensa en un caso general para escribir las líneas que lo resuelven. Es de esperar que los casos especiales, o los casos límite, provoquen un error con mayor probabilidad. Se pueden diseñar más fácilmente si conocemos los detalles internos.
   Podríamos afirmar que un programador experimentado realiza de una forma inconsciente este tipo de pruebas. Por ejemplo, después de implementar un caso general, revisa mentalmente el comportamiento con casos especiales. Se podría considerar una manera informal de realizar algún tipo de prueba de caja blanca.
   Otro ejemplo de las ventajas de este tipo de pruebas es el estudio de los distintos caminos de ejecución. Conociendo la implementación, podemos asegurarnos de que las pruebas cubren un porcentaje alto del código.
2. Pruebas de *caja negra*. En este caso no se conocen los detalles internos. Se considera la interfaz y su especificación para el diseño de los casos de prueba.

Un ejemplo es un módulo que utiliza un parámetro para seleccionar el método de procesamiento. Si tenemos, digamos, tres valores posibles, conviene diseñar al menos tres pruebas para esos valores.

Una estrategia de prueba puede ser, primero, distinguir distintos tipos de entradas. Podemos particionar el conjunto de entradas en casos que pueden tener un procesamiento similar, y seleccionar para cada subconjunto un caso de prueba. Note, además, que también podemos considerar casos límite.

Las pruebas del software se pueden situar a distintos niveles:

1. Pruebas de *módulo*. Se prueban los módulos de forma independiente. Podemos decir que son pruebas de los componentes internos de un módulo que garantizan que funciona correctamente según sus especificaciones. En este caso, son de especial interés las pruebas de caja blanca. Las pruebas de módulo son normalmente realizadas por el equipo de implementación.
2. Pruebas de *integración*. Se prueba la interacción entre módulos. Este conjunto de pruebas consiste en ir añadiendo módulos de forma progresiva y pasando un conjunto de casos de prueba. En este caso, son de especial interés las pruebas de caja negra. Las pruebas de integración son normalmente realizadas por el equipo de diseño e implementación.
3. Pruebas de *aplicación*. En este caso, se considera todo el software desarrollado. Por tanto, se pueden realizar pruebas de ejecuciones reales; por ejemplo, guiones obtenidos en la primera etapa. Normalmente realizadas por el equipo de análisis.

No piense que las pruebas son en gran medida la interacción de un usuario con una parte del programa. Para el desarrollo de las pruebas, será necesario desarrollar código adicional. Por ejemplo, para sustituir una parte del código, almacenar parámetros para obtener los resultados, generar valores de entrada, simular situaciones de interacción con el usuario, etc.

Incluso puede ser muy recomendable realizar tareas de prueba automatizadas. Por ejemplo, puede generar software que lanza una batería de pruebas prediseñadas para confirmar que el programa obtiene unos resultados de acuerdo a las especificaciones. Estas pruebas se pueden lanzar repetidamente, por ejemplo, conforme incluimos modificaciones o mejoras en el programa.

D Tablas

Tabla ASCII 343
Operadores C++ 344
Palabras reservadas de C89, C99, C11, C++ y C++11 346

D.1 Tabla ASCII

En la figura D.1 se presenta la tabla de codificación *ISO-8859-15* del alfabeto latino. Es similar a la *ISO-8859-1* (véase figura 7.1 en la página 157) 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.

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.

	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	SGCI	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬	SHY	®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	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 D.1
Tabla de codificación *ISO-8859-15*.

D.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	::*nombre*
`::`	Resolución de ámbito	*espacio_nombres*::*miembro*
`::`	Resolución de ámbito	*nombre_clase*::*miembro*
`->`	selección de miembro	*puntero*->*miembro*
`.`	Selección de miembro	*objeto*.*miembro*
`[]`	Índice de vector	*nombre_vector*[*expr*]
`()`	Llamada a función	*nombre_función*(*lista_expr*)
`()`	Construcción de valor	*tipo(lista_expr)*
`++`	Post-incremento	*valor_i*++
`--`	Post-decremento	*valor_i*--
`typeid`	Identificador de tipo	typeid(*type*)
`typeid`	... en tiempo de ejecución	typeid(*expr*)
`dynamic_cast`	conversión en ejecución	dynamic_cast<*tipo*>(*expr*)

continúa en la página siguiente

continúa de la página anterior

Operador	*Nombre*	*Uso*
	con verificación	
`static_cast`	conversión en compilación	static_cast<*tipo*>(*expr*)
	con verificación	
`reinterpret_cast`	conversión sin verificación	reinterpret_cast<*tipo*>(*expr*)
`const_cast`	conversión const	const_cast<*tipo*>(*expr*)
`sizeof`	Tamaño del tipo	sizeof(*tipo*)
`sizeof`	Tamaño del objeto	sizeof *expr*
`++`	Pre-incremento	++*valor_i*
`--`	Pre-decremento	--*valor_i*
`~`	Complemento	~*expr*
`!`	No	!*expr*
`+`	Más unario	+*expr*
`-`	Menos unario	-*expr*
`&`	Dirección de	&*valor_i*
`*`	Desreferencia	`*`*expr*
`new`	reserva	new *tipo*
`new`	reserva e iniciación	new *tipo (lista_expr)*
`new`	reserva (emplazamiento)	new *(lista_expr) tipo*
`new`	reserva (con inicialización)	new *(lista_expr)tipo(lista_expr)*
`delete`	destrucción (liberación)	delete *puntero*
`delete[]`	... de un vector	delete[] *puntero*
`()`	Conversión de tipo	(*tipo*) *expr*
`.*`	Selección de miembro	*objeto*`.*`*puntero_a_miembro*
`->*`	Selección de miembro	*puntero*`->*`*puntero_a_miembro*
`*`	Multiplicación	*expr*`*`*expr*
`/`	División	*expr*/*expr*
`%`	Módulo	*expr*%*expr*
`+`	Suma	*expr*+*expr*
`-`	Resta	*expr*-*expr*
`<<`	Desplazamiento a izquierda	*expr*<<*expr*
`>>`	Desplazamiento a derecha	*expr*>>*expr*
`<`	Menor	*expr*<*expr*
`<=`	Menor o igual	*expr*<=*expr*
`>`	Mayor	*expr*>*expr*
`>=`	Mayor o igual	*expr*>=*expr*
`==`	Igual	*expr*==*expr*
`!=`	No igual	*expr*!=*expr*
`&`	Y a nivel de bit	*expr*&*expr*
`^`	O exclusivo a nivel de bit	*expr*^*expr*
|	O a nivel de bit	*expr*|*expr*
`&&`	Y lógico	*expr*&&*expr*
||	O lógico	*expr*||*expr*
`?:`	expresión condicional	*expr*?*expr*:*expr*
`=`	Asignación	*valor_i*=*expr*
`*=`	Multiplicación y asignación	*valor_i*`*=`*expr*
`/=`	División y asignación	*valor_i*/=*expr*
`%=`	Resto y asignación	*valor_i*%=*expr*

continúa en la página siguiente

continúa de la página anterior

Operador	***Nombre***	***Uso***
`+=`	Suma y asignación	*valor_i*`+=`*expr*
`-=`	Resta y asignación	*valor_i*`-=`*expr*
`<<=`	Desplazar izq. y asignación	*valor_i*`<<=`*expr*
`>>=`	Desplazar der. y asignación	*valor_i*`>>=`*expr*
`&=`	Y y asignación	*valor_i*`&=`*expr*
`^=`	O exclusivo y asignación	*valor_i*`^=`*expr*
`|=`	O y asignación	*valor_i*`|=`*expr*
`throw`	Lanzamiento de excepción	`throw` *expr*
`,`	Coma	*expr* `,` *expr*

Tabla D.1
Operadores de C++

Estos operadores están incluidos en el estándar C++98. En C++11 aparecen tres más:

`sizeof...`, **`noexcept`**, *`alignof`*

D.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 D.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.

Tabla D.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	

Bibliografía

[1] Aho, Hopcroft y Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.

[2] Andrei Alexandrescu. *Modern C++ Design. Generic Programming and Design Patterns Applied.* Addison-Wesley. 2001.

[3] Brassard y Bratley. *Fundamentos de algoritmia*. Prentice Hall, 1997.

[4] Carrano, Helman y Veroff. *Data abstraction and problem solving with C++. Walls and Mirrors*. Tercera edición. Addison-Wesley, 2001.

[5] Deitel y Deitel, *C++ How to program*. Octava Edición revisada. Prentice Hall. 2011.

[6] B. Eckel, *Thinking in C++* (2ª edición). Prentice-Hall. 2000. Disponible en versión electrónica en *http://www.bruceeckel.com/*

[7] Martin Fowler, *Refactoring. Improving the design of existing code.* Addison-Wesley. 2000.

[8] Edward A. Fox y otros, *Practical minimal perfect hash functions.* Communications of the ACM, vol. 35, N.1, Enero 1992.

[9] Michael L. Fredman, Robert Endre Tarjan, *Fibonacci Heaps and their uses in Improved Network Optimization algorithms*, Journal of the ACM, Vol. 34, N. 3, Julio 1987. Páginas 596-615.

[10] A. Garrido. *Fundamentos de programación en C++*, Delta publicaciones, 2006.

[11] George, T.Heineman, William, T.Council, *Component-based software engineering*, Addison-Wesley, 2001.

[12] B. Jenkings, *Algorithm Alley*, Dr. Dobb's Journal, septiembre 1997. Disponible en *http://burtleburtle.net/bob*.

[13] Nicolai Josuttis, *The C++ standard library. A tutorial and handbook (2nd edition)*. Addison-Wesley, 2012.

[14] Donald E. Knuth, *The Art of Computer Programming*, vol. 2 (seminumerical algorithms). Addison-Wesley, 3rd Edition, 1997.

[15] Donald E. Knuth, *The Art of Computer Programming*, vol. 3 (Sorting and Searching). Addison-Wesley Publishing Company, 2nd edition, 1998.

[16] B. Liskov, John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1987.

[17] K. Mehlhorn, S. Naher. *LEDA. A platform for combinatorial and geometric computing*. Cambridge, 1999.

[18] S. Meyers, *Effective C++: 50 specific ways to improve your programs and designs*(segunda edición), Addison Wesley, 1992.

[19] S. Meyers, *More effective C++: 35 new ways to improve your programs and designs*, Addison Wesley, 1996.

[20] R. Morris, *Scatter Storage Techniques*, Communications of the ACM, Vol. 11, N. 1, Enero 1968.

[21] Peña Marí, *Diseño de Programas, Formalismo y Abstracción*. Prentice Hall, 1997.

[22] Pressman, *Ingeniería del software. Un enfoque práctico*. MacGraw Hill, 1993.

[23] William H. Press y otros. *Numerical recipes in C++: The art of scientific computing*. Cambridge University Press, 2002.

[24] A. Prieto, A. Lloris, J.C. Torres, *Introducción a la informática*, tercera edición, McGraw-Hill, 2002.

[25] Charles E. Radke, *The use of quadratic residue research*, Communications of the ACM, Vol 13, N. 2, Febrero, 1970.

[26] Ravenbrook's Memory Management Reference, *http://www.memorymanagement.org*. Ravenbrook Limited.

[27] Herbert Schildt, *Manual de referencia C (cuarta edición)*. Mcgraw-Hill, 2000.

[28] Sedgewick, *Algorithms*, Addison Wesley, 1989.

[29] Alan Shalloway, James R. Trott. *Design Patterns explained. A new perspective on object-oriented design.*. Addison-Wesley. 2002.

[30] Ian Sommerville, *Software engineering (6th edition)*. Addison-Wesley, 2001.

[31] B. Stroustrup, *El lenguaje de programación C++. Edición Especial*. Addison-Wesley, 2002.

[32] B. Stroustrup, *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.

[33] B. Stroustrup, *The design and Evolution of C++*. Addison-Wesley, 1994.

[34] B. Stroustrup, *C++ Style and Technique FAQ*. http://www.research.att.com/b̃s/bs_faq2.html.

[35] H. Sutter. *Exceptional C++: 47 engieering puzzles, programming problems, and solutions*. Addison Wesley, 2000.

[36] H. Sutter. *More exceptional C++: 40 new engieering puzzles, programming problems, and solutions*. Addison Wesley, 2002.

[37] H. Sutter. *Exceptional C++ Style*. Addison Wesley, 2005.

[38] H. Sutter, A. Alexandrescu. *C++ Coding Standards*. Addison Wesley, 2005.

[39] H. Sutter, *To new, perchance to throw, Part 2*. C/C++ Users Journal, 19(5), May 2001.

[40] Clemens Szyperski,*Component Software. Beyond Object-Oriented programming*. Addison-Wesley, 1998.

[41] David Vandevoorde, Nicolai M. Josuttis, *C++ Templates. The complete guide.* Addison Wesley. 2003.

[42] J.S. Vitter, *Implementations for coalesced hashing*. Communications of the ACM, Vol. 25, N. 12, diciembre 1982.

[43] M.A. Weiss, *Data Structures and problem solving using C++*. Addison-Wesley, 2000.

Índice alfabético

A

abs, 292
abstracción, 339
acoplamiento, 339
algoritmo, 15
ALU, *véase* unidad aritmético-lógica
ámbito
  de una variable, 75, 88, 101
  espacio de nombres y, 249
  global, 101
  operador de, 101
  operador de resolución de, 249
análisis
  ingeniería del software y, 330, 338
análisis léxico, 18
análisis semántico, 18
análisis sintáctico, 18
and, 50
anidamiento
  bucles, 72
  de estructuras, 166
  if/else, 47
apertura de archivo, 226
aplicación
  pruebas de, 106, 342
archivo
  apertura y cierre, 226
argumento
  de función, 83
argumentos, 83
array-C, 109
ASCII, 10, 156
  extensiones, 157
asignación
  de estructura, 167
  operador de, 28
  operadores compuestos, 67
asociatividad de operadores, 27
auto, 260
  cadena-C/string y, 266

B

bad, 218
badbit, 218
banderas
  de flujos E/S, 218
biblioteca, 247
  compilación separada y, 246
  funciones de, 35
binaria
  búsqueda, 120
binario
  sistema, 6
bisección
  raíz usando, 62
bloque de sentencias, 45
bool, 24
  en C (y C++), 54
  expresión de tipo, 50
  literal, 25
break, 53, 77
bucket sort, 141, 173, 181
bucle, 59
  anidado, 72
  cuerpo del, 59

buffers
  flujos y, 212
burbuja
  ordenación por, 126
búsqueda, 118
  binaria, 120, 197
  dicotómica, 120
  secuencial, 119
    garantizada, 119

C

C++11, 257
C11, 15
C89, 14
cabecera
  archivos de, 22, 239
  función, 83, 86
cadena de caracteres
  literal, 25
cadena-C, 145
  auto en C++11, 266
  open(de flujo) y, 233
caja blanca
  pruebas de, 341
caja negra
  pruebas de, 341
cálculo lamda, 337
calidad
  ingeniería del software y, 330
calificación,espacio de nombres y, 249
campo
  de estructura, 163
carácter especial
  literal, 26
característica, 9
caracteres
  representación de, 9
cascada
  paradigma, 332
case, 53
caso base, recursividad, 193
centinela, 61
cerr, 30, 214
char, 24
  codificación, 156
  literal, 25
ciclo, 59
cierre de archivo, 228
cin, 30, 212, 214
  tipo asociado a, 231
clear, 222
clog, 30, 214
close, 228
cmath, 36
codificación
  de caracteres, 156
  de la información, 5
  editor y, 162
  longitud fija/variable, 160
cohesión, 340
cola
  recursividad de, 202
coma
  operador, 67
comentarios
  en programas C++, 21
compilación, 17, 237
  errores de, 17
  separada, 236
compilador, 16
componentes
  modelo orientado a, 332
  paradigma de, 332
computadora, 1
  E/S de datos, 1
  esquema de, 2
  funcionamiento de, 3
concatenación
  de string, 152
condicional
  estructura, 43
const, 37, 101
  referencia y, 117, 167
constante
  global, 101
    múltiples ficheros, 254
constantes
  declaración de, 37
contador
  bucle por, 63
contenedor, 109
  de char, 147
contexto
  de ejecución de función, 91
  de función, 190
continue, 77
conversiones
  aritméticas, 28
copia
  de estructura, 167
counting sort, 180
  generalizado, 182
cout, 30, 212, 214
  flush(), 213
  tipo asociado a, 231
cpu, 2
c_str, 233
Ctrl-D, 217
Ctrl-Z, 217
cuerpo
  de bucle, 59
  función, 82

D

datos
  abstracción de, 334
  expresiones, 23
  representación de los, 8
decimal
  sistema, 5
declaración
  de variables, 24, 253, 258

struct y, 165
función, 84
llaves y paréntesis, 260
declaración de uso, 251
decremento
operador, 65
default, 53
define, 240
definición
función, 82
variable, 253
descomposición modular, 339
diagrama
de flujo, 56
diccionario, 178
dicotómica
búsqueda, 120
dinámico
eficiencia de vector de la STL, 136
vector de la STL, 132
directiva de uso, 251
directivas del preprocesador, 19
diseño
ingeniería del software y, 330, 338
do/while, 71
double, 24
literal, 25
string y (C++11), 268

E

ejecución
caminos de, 45
de programa, 3
errores de, 17
ejecutable
fichero, 18
encapsulamiento, 335
endif, 242
enlace
externo, 253, 255
interno, 253, 254
enlazado, 17, 237, 247
bibliotecas y, 36
errores de, 17
enteros,representación de, 8
entidad-relación
modelo, 333
entrada
estándar, 30, 214
flujo de, 212
EOF, 216, 220, 319
eof, 218
final del flujo y, 220
eofbit, 218, 220
espacio blanco, 23, 34
espacio de nombres, 248
anónimo, 254
std, 251
especificación
de funciones, 104
herramientas para, 105
semántica, 104
sintáctica, 104
estructura, 163
anónima, 165
asignación, 167
campo de, 163
copia, 167
definición, 164
inicialización, 165, 270
miembro de, 163
operador punto, 165
ordenación, 170
Euclides
algoritmo de, 62
evolutivos
paradigmas, 332
exit, 98
exp, 36
expresión
booleana, 50
extern, 253
externo
enlace, 255

F

fabs, 36
fail, 218
failbit, 218
Fibonacci
sucesión de, 203
fichero, *véase* archivo
fichero objeto, 17
flags
de flujos E/S, 218
flujo
buffer y, 212
diagrama de, 56
flujo de datos
modelo de, 332
programación orientada a, 335
flujo de E/S, 30, 212
archivo y, 225
C++98 y, 232
copia de, 230
estado de un, 218
modificación del, 221
expresiones booleanas y, 219
flags, 218
operaciones básicas, 216
paso a función de, 230
redireccionamiento, 214
for, 63
basado en rango, 261, 267
while y, 64
formal
parámetro, 82
formato libre, 23
fuente
fichero, 22, 237
función
argumento, 83

argumentos de una, 83
cabecera, 83
cabecera de, 86
contexto de, 190
contexto de ejecución, 91
cuerpo de, 82
declaración de, 84
definición, 82
diseño de, 98
errores en una, 106
especificación de, 104
llamada a, 83
pila y llamada a, 189
programación estructurada y, 101
recursiva
diseño, 193
ejemplos, 186
implementación, 195
tipo de una, 83
void, 91
función de biblioteca
rand, 324
srand, 325
time, 325
funcional
programación, 337

G

generación de código, 18
get, 217
tuplas y (C++11), 271
getline
string y, 148
gibibyte, 7
gigabyte, 7
global
constante, 101
múltiples ficheros, 254
variable, 100
múltiples ficheros, 252
good, 218
goodbit, 218
goto, 76
GUI, 30

H

Hanoi
torres de, 203
hardware
definición, 2
herencia, 335
heterogéneo
tipo de dato, 163, 270
hexadecimal
sistema, 6
homogéneo
tipo de dato, 109, 145

I

if, 44
if/else, 47
ifdef-endif, 242
ifndef-endif, 242
ifstream, 226
ignore
función de flujos, 223
tuple (C++11), 274
implementación, 16
ingeniería del software e, 330, 341
include, 19, 238
incremento
operador, 65
información
codificación de la, 5
ingeniería
del software, 329
inicialización
de objetos y tipos simples, 258
de pares y tuplas (C++11), 272
estructuras, 165, 270
listas de, 263
pares y tuplas, 270
tipos simples en C++11, 258
variables y constantes, 37
inserción
ordenación por, 123
int, 24
literal, 25
string y (C++11), 268
integración
pruebas de, 106, 342
interfaz, 30
de usuario, 30
función e, 86
interno
enlace, 254
intérprete
de órdenes, 214
intérprete, 16
introspección, 336
isalpha, 36
isdigit, 36
isgraph, 36
islower, 36
ISO-8859-1, 157
iterativa
estructura, 43

K

kibibyte, 7
kilobyte, 7

L

lazo, 59
lectura adelantada, 70
lenguaje máquina, 12
lenguajes de programación, 12
léxico
análisis, 18
lineal secuencial

paradigma, 332
literal, 25
string, 146
log, 36

M

máximo común divisor, 62
macros
directivas y, 241
main, 21, 97
make_pair, 177
make_tuple, 272
mantenimiento, 82, 98
ingeniería del software y, 330
mantisa, 9
matriz, 137
declaración
vector de la STL, 138
no rectangular, 141
rectangular, 139
vector de la STL y, 138, 261
mebibyte, 7
megabyte, 7
memoria
masiva, 2
posición de, 2
principal, 2
unidades de, 7
metaprogramación, 336
mezcla
de secuencias ordenadas, 131
ordenación por, 206
miembro
de estructura, 163
modelo del sistema, 332
modular
programación, 334
modularización, 98, 339
acoplamiento, 339
cohesión, 340
programación estructurada y, 101
módulo
pruebas de, 342
move semantics, 265, 267
mover/copiar un objeto, 265, 267

N

namespace, 249
NDEBUG, 241
no-op
flujos y, 222
not, 50
npos (string), 155
numeración
sistemas de, 5
números aleatorios, 323
pseudoaleatorios, 324

O

objeto
fichero, 17, 237
inicialización de, 258
objetos
modelo orientado a, 332
programación orientada a, 335
octal
sistema, 6
ocultamiento de información, 99, 104, 339
ofstream, 226
open, 226
banderas y, 233
operadores
aritméticos y relacionales, 46
asociatividad de, 27
lógicos, 50
precedencia de, 27
prioridad y, 26
relacionales, 44
optimización
traducción y, 18
or, 50
ordenación
mezcla, 206
selección, 205
ordenación
algoritmos de, 121
burbuja, 126
cartero, 173
comparación de algoritmos, 128
estable, 172
indirecta, 129
inserción, 123
mezcla ordenada, 131
por casilleros, 141, 173, 181
por contero, 180
generalizado, 182
selección, 121
órdenes
intérprete de, 214

P

pair, 175, 270
diccionario y, 178
make_pair, 177
tie (C++11), 273
palabras reservadas, 25, 346
Par, de la STL, 175
parámetro
formal, 82
paradigmas
programación y, 333
software y, 331
peek, 223
pila
gestión de llamadas, 189
pipe, *véase* tuberías
polimorfismo, 335
portabilidad, 13
posición
de memoria, 2
postcondición, 104

postdecremento, 65
postincremento, 65
postman'sort, 173
pow, 36
precompilador, 19
precondición, 104
predecremento, 65
preincremento, 65
preprocesador, 19, 238
preprocesamiento, 19
procedimental
    programación, 333
procedimiento, 92
proceso del software, 330
programa
    definición, 1
    ejecución, 3
    traza del, 86
programación
    funcional, 337
    imperativa, 337
    lenguajes de, 12
programación estructurada, 76, 101, 333
programación genérica, 336
programación
    orientada a objetos, 335
prototipo
    función, 86
prueba, 106
    caja blanca, 341
    caja negra, 341
    de aplicación, 106
    de integración, 106
    de módulo, 106
    ingeniería del software y, 330, 341
pseudoaleatorio, 324
put, 217

R

rand, 324
range based for, 261, 267
rango
    bucle basado en, 261
reales
    representación de, 9
recursividad, 185
    casos base, 193
    casos generales, 194
    de cola, 202
    directa, 186
    indirecta, 186
    infinita, 197
    profundidad de, 192, 201
recursivo vs iterativo, 201
redireccionamiento de E/S, 214
referencia
    constante, 117
    for basado en rango y, 262
    paso por, 93
reflexión, 336
registro, 163
relacionales
    operadores, 44
representación de la información, 8
requisitos
    en el análisis, 338
reservadas
    palabras, 25, 346
return, 83
    funciones void y, 103
    posición de, 102
reusabilidad, 82, 98

S

salida
    estándar, 30, 214
        de error, 30, 214
        de logging, 214
    flujo de, 212
secuencial
    búsqueda, 119
    estructura, 43
selección
    doble, 47
    múltiple, 53
    ordenación por, 205
selección
    estructura de, 43
    ordenación por, 121
    simple, 44
semántico
    análisis, 18
semilla, 324
sentencia, 16
sentencia compuesta, 45
setprecision, 295
sintáctico
    análisis, 18
software
    definición, 2
    desarrollo de, 15
    deterioro del, 331
sqrt, 36
srand, 325
static, 253
std
    espacio de nombres, 251
stod, 269
stoi, 269
string, 146, 266
    clear(), 154
    conversiones en C++11, 268
    declaración con llaves C++11, 266
    declaración de, 146
    E/S, 147
    erase(), 153
    find, 155
        auto en C++11 y, 267
    funciones y, 156
    getline, 148
    literal, 146
    npos, 155

replace(), 154
rfind, 155
size_type, 155
subcadena, 153
substr(), 153
struct, 164
switch, 53, 77

T

tan, 36
tebibyte, 7
terabyte, 7
tie (C++11), 273
time, 325
tipo de dato
deducción del, 260
definido por el usuario, 164
heterogéneo, 163, 270
homogéneo, 109, 145
predefinido, 24
tokens, 23
análisis léxico y, 18
tolower, 36
tope
pila y, 189
to_string, 268
toupper, 36
traducción
proceso de, 16
unidad de, 255
traza del programa, 86
tubería (pipe), 216
tuplas en C++11, 270
tuple, 271
tie y, 273

U

UAL, *véase* unidad aritmético-lógica
undef, 241
unget, 223
unicode, 160
unidad
de traducción, 255
unidad aritmético-lógica, 2
unidad central, 2
unidad de control, 2
unidades de memoria, 7
using, 251
declaración, 251
directiva, 251
usuario, 16
utf, 160
utf16, 160
utf32, 160
utf8, 160

V

valor
paso por, 93
variable, 24
declaración de, 24, 253, 258
definición, 253
global, 100
local
a fichero, 253
paso por, *véase* referencia
vector, 110
de la STL, 109, 261
acceso con [], 112
acceso con at(), 112
búsqueda, 118
clear(), 136
declaración de, 110
dinámico, 132
dinámico (eficiencia), 136
funciones y, 117
inicialización en C++11, 263
matrices y, 138, 261
pop_back(), 133
push_back(), 132
resize(), 136
size(), 114
vector-C, 109
void, 91
return y funciones, 103

W

while, 59
windows-1252, 158
Wirth, Niklaus, 109