

Introducción a la Programación con C++

Ejercicios

A. Garrido y J. Martínez-Baena

eug
EDITORIAL
UNIVERSIDAD
DE GRANADA



© A. GARRIDO Y J. MARTÍNEZ-BAENA
© UNIVERSIDAD DE GRANADA
© Fotografías y cubierta: ANTONIO GARRIDO
INTRODUCCIÓN A LA PROGRAMACIÓN CON C++: Ejercicios
ISBN: 978-84-338-5924-2.
Edita: Editorial Universidad de Granada.
Campus Universitario de Cartuja. Granada.

Printed in Spain

Impreso en España.

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

Índice general



1	Expresiones en C++	1
1.1	Introducción	1
1.1.1	Tipos de datos y operaciones	1
1.2	Tipos de datos numéricos	1
1.2.1	Entero vs real	2
1.2.2	Variables	2
1.2.3	Constantes	3
1.2.4	Funciones de biblioteca	3
1.2.5	Errores de aproximación de números reales	4
1.3	Tipo carácter	5
1.3.1	E/S de un carácter	5
1.3.2	Funciones de biblioteca para caracteres	5
1.4	Expresiones complejas	6
1.5	Conversiones explícitas	6
1.6	Ejercicios adicionales	6
2	La estructura de selección	9
2.1	Introducción	9
2.2	La instrucción if simple	9
2.3	La instrucción if/else	9
2.3.1	La instrucción if/else anidada	10
2.4	Condiciones compuestas: operadores lógicos	11
2.4.1	Evaluación en corto	12
2.5	La instrucción switch	12
2.6	Booleanos y enteros	13
2.7	Ejercicios adicionales	13
3	La estructura de iteración	17
3.1	Introducción	17

3.2	Patrones de diseño de bucles	17
3.2.1	El bucle do-while	17
3.2.2	El bucle while	18
3.2.3	Soluciones habituales con while	19
3.2.4	¿Repetir mientras o repetir hasta?	20
3.3	El bucle for	21
3.3.1	Inicialización, condición e incremento	21
3.3.2	Ejecutando un cierto número de veces	21
3.3.3	Omitiendo expresiones	22
3.4	Anidamiento de bucles	22
3.5	Variables lógicas y condiciones compuestas	23
3.6	Casos generales y casos límite	24
3.7	Algunos errores comunes	24
3.7.1	Funciona, pero mejor no hacerlo	25
3.8	Ejercicios adicionales	26
4	Vectores y matrices	29
4.1	Introducción	29
4.2	El tipo vector	29
4.2.1	Operaciones con vectores	30
4.2.2	Vectores de tamaño variable	32
4.2.3	Vectores y lectura adelantada	32
4.3	Vectores de vectores	32
4.3.1	Matrices	33
4.4	Ordenación y búsqueda	33
4.5	Ejercicios adicionales	34
5	Cadenas de caracteres	37
5.1	Introducción	37
5.2	El tipo string	37
5.3	Operaciones básicas con string	38
5.3.1	E/S de string	38
5.3.2	Tamaño y acceso a los caracteres	38
5.3.3	E/S combinada con otros tipos	39
5.4	Más operaciones con string	41
5.4.1	Concatenación de cadenas	41
5.4.2	Comparación de cadenas	41
5.4.3	Extracción de subcadenas	42
5.4.4	Borrado de subcadenas	42
5.4.5	Inserción de subcadenas	42
5.4.6	Reemplazo de subcadenas	43
5.4.7	Búsqueda en cadenas	43
5.5	Ejercicios adicionales	44
6	Funciones	47
6.1	Introducción	47
6.1.1	Notación	47

6.2	Parametrización de las funciones	48
6.2.1	El paso por valor	48
6.2.2	Las funciones de tipo void	49
6.2.3	El paso por referencia	50
6.2.4	Conversiones	52
6.3	Diseño de funciones	52
6.3.1	La separación entre la E/S y los cálculos	52
6.3.2	Diseño descendente (top-down)	54
6.4	Gestión de errores	57
6.4.1	Devolución de un valor de error	58
6.4.2	Errores graves irrecuperables	58
6.4.3	Precondiciones y postcondiciones	58
6.5	Valores por defecto y sobrecarga	60
6.5.1	Parámetros con valores por defecto	60
6.5.2	Sobrecarga de funciones	60
6.6	Ejercicios adicionales	61
7	Funciones con cadenas y vectores	65
7.1	Introducción	65
7.2	Ejercicios de funciones y vectores	65
7.3	Ejercicios de funciones y cadenas	67
7.4	Ejercicios adicionales	68
8	Funciones recursivas	73
8.1	Introducción	73
8.2	El caso general y el caso base	73
8.3	Funciones recursivas con varios puntos de salida	74
8.4	Múltiples casos base y/o generales	74
8.5	Ejercicios adicionales	74
9	Estructuras y pares	77
9.1	Introducción	77
9.2	Estructuras en C++	77
9.2.1	Estructuras y funciones	77
9.2.2	Anidamiento de estructuras	78
9.2.3	Estructuras y otros datos compuestos	78
9.3	El tipo par de la STL	79
9.4	Ejercicios adicionales	80
10	Flujos de Entrada/Salida	83
10.1	Introducción	83
10.1.1	Operaciones básicas de transferencia	84
10.2	Ficheros	86
10.2.1	Apertura de ficheros	86
10.2.2	Añadiendo datos a ficheros	87
10.2.3	Cierre de ficheros	88
10.3	Detección del fin del flujo	88
10.4	Control de errores en los flujos	89
10.4.1	Control de errores en ficheros	89

10.5	Reutilización de flujos	90
10.6	Otras operaciones sobre flujos	90
10.7	Flujos y funciones	91
10.7.1	Copia de flujos	92
10.7.2	Uso de flujos “genéricos”	92
10.8	Algunas recomendaciones finales	92
10.8.1	Lectura adelantada y flujos	92
10.8.2	Ventajas e inconvenientes de “no-op”	93
10.8.3	Evitar la lectura parcial de objetos	94
10.9	Ejercicios adicionales	95
A	Generación de números aleatorios	97
A.1	Introducción	97
A.2	Números pseudoaleatorios	97
B	Redirección de Entrada/Salida	101
B.1	Introducción	101
B.2	Ejecuciones con muchos datos	103
B.3	Almacenamiento externo	104
B.3.1	Añadiendo en lugar de sustituyendo	104
B.4	Redirección de E/S simultánea	104
B.5	Redirección de cerr	105
B.6	Encauzamiento	105
C	Tablas	107
C.1	Tabla ASCII	107
C.2	Operadores C++	108
C.3	Palabras reservadas de C89, C99, C11, C++ y C++11	109
	Bibliografía	111
	Índice alfabético	113

Prólogo



La enseñanza de la programación de ordenadores es un tema ampliamente discutido, no sólo por la dificultad de enseñar una serie de conceptos y habilidades que pueden resultar bastante complicados, sino también por la rapidez con que cambia la tecnología, dando lugar no sólo a nuevos lenguajes de programación, sino también a nuevas formas de abordarla.

El rápido desarrollo de las tecnologías disponibles para programar ordenadores ha tenido un efecto directo sobre la enseñanza, ya que ha incrementado el conjunto de habilidades que se supone que debe poseer un programador. Aun así, no podemos olvidar que los fundamentos de la programación son los mismos.

Un curso de fundamentos de programación presenta conceptos relativamente simples. Sin embargo, para un estudiante que comienza en la programación, los conceptos que se discuten pueden resultar bastante abstractos y difíciles de asimilar. Esta dificultad puede verse compensada en gran medida con una metodología orientada a la práctica. Por ello, tan importante como disponer de un buen manual de referencia sobre el lenguaje de estudio —C++ en nuestro caso— lo es disponer de un buen libro de apoyo en la realización de prácticas.

Precisamente, ese es el objetivo de este libro: ofrecer al estudiante un documento de trabajo para facilitar el aprendizaje aplicando de forma práctica los conceptos aprendidos en clase o en otros libros más teóricos sobre C++ y la *STL*. En concreto, nos permite ofrecer un guión de prácticas relacionadas con el libro *Fundamentos de Programación con la STL* (Garrido[5]).

Organización del cuaderno de prácticas

Para compensar la falta de experiencia del estudiante, los ejercicios del cuaderno se diseñan para que incidan especialmente en situaciones habituales en la práctica de programación. De hecho, un programador con experiencia consideraría simples muchos de los problemas que se proponen. Esta simplicidad es consecuencia de que tiene asimilados muchos esquemas de solución —patrones de diseño— que ha resuelto repetidamente.

Los temas se plantean con una serie de ejercicios que hagan explícitas estas formas de solución. Adicionalmente, en algunos casos se incluye un pequeño resumen de aspectos teóricos importantes o una discusión sobre la forma en que cierto diseño facilita la solución de problemas. Cada sección suele estar diseñada para practicar algún aspecto concreto. Además, parte de los ejercicios permiten profundizar en algunos puntos de especial relevancia y que suelen pasar desapercibidos por los estudiantes.

El libro está diseñado para comenzar desde cero el aprendizaje de la programación. Como se puede comprobar con un simple vistazo al índice de capítulos, se trata de programación estructurada y modular. Aunque C++ es un lenguaje multiparadigma, este cuaderno se ha creado para un curso de fundamentos; el objetivo es que el estudiante acabe con una base sólida que permita el abordaje de cursos más avanzados con garantías de éxito. Más concretamente, se llevan a cabo prácticas que incorporan contenidos sobre:

- Programas simples con expresiones en C++.
- Estructuras de control, incluyendo la selección y la iteración.
- Diseño de funciones.
- Tipos compuestos homogéneos: vectores, matrices y cadenas.
- Tipos compuestos heterogéneos: estructuras y pares.
- Problemas simples de E/S con ficheros de texto.

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

- La *generación de números aleatorios*. Es un tema que generalmente no se aborda directamente en las clases de teoría, sino que se supone se practicará cuando se desarrollen programas que generan valores aleatorios. Sin embargo, es un tema cuyo contenido teórico es muy relevante para poderlo usar adecuadamente. En lugar de dar una breve especificación de las funciones que ofrece el lenguaje, se incluye una exposición más detallada con el fin de que el estudiante no sólo lo use, sino de que entienda por qué funciona.
- La *redirección de E/S*. Aunque inicialmente se plantea como un tema relacionado con la E/S, se ha dejado para un apéndice puesto que es un contenido transversal para todo el curso. Desde los primeros ejercicios ya podemos aprovechar este redireccionamiento para ejecuciones rápidas desde la línea de órdenes.
- *Tablas* relacionadas con el lenguaje. En la práctica, es muy útil disponer de tablas que incluyen detalles sobre palabras reservadas, operadores del lenguaje, código *ASCII*, etc.

Entorno de programación

El curso está basado en el estándar más extendido de C++ —el del 98/03— aunque sigue siendo eficaz si está interesado en programar con el último estándar, ya que los ejercicios y las discusiones siguen siendo igualmente válidos. Desde este punto de vista, puede usar cualquier versión, aunque se recomienda al menos la del 98. En la fecha que estamos, seguro que cualquier compilador cumple con esta condición.

No es necesario usar ningún compilador o entorno de programación concreto. Se ha desarrollado para que el estudiante sea libre de optar por las herramientas que le sean más cómodas, siendo casi cualquier compilador estándar una buena opción para practicar. Sólo es necesario un editor de texto y un compilador.

Sin embargo, siendo un primer contacto con la programación, es recomendable que el estudiante pueda centrarse en los detalles del lenguaje y deje la dificultad de la gestión de proyectos para más adelante. Por ello, se recomienda algún entorno que facilite la compilación y ejecución. En nuestro caso, hemos usado *Code::Blocks*, que se ajusta perfectamente a los objetivos del curso.

Agradecimientos

Los autores desean agradecer a todos sus alumnos por haber contribuido —aun sin saberlo— a la creación de este cuaderno, especialmente a los que “*se equivocan*” porque precisamente ellos permiten encontrar los puntos débiles en el proceso de enseñar a programar.

Esta sección no puede terminar sin un agradecimiento a todos los que con su trabajo desinteresado han contribuido al software libre, creando sistemas y herramientas gratuitas que nos permiten por un lado crear este documento —íntegramente desarrollado con software libre— y por otro, mucho más interesante, ofrecer una solución totalmente gratuita para realizar este curso.

A. Garrido y J. Martínez-Baena
Mayo de 2016.

1

Expresiones en C++

Introducción	1
Tipos de datos y operaciones	
Tipos de datos numéricos	1
Entero vs real	
Variables	
Constantes	
Funciones de biblioteca	
Errores de aproximación de números reales	
Tipo carácter	5
E/S de un carácter	
Funciones de biblioteca para caracteres	
Expresiones complejas	6
Conversiones explícitas	6
Ejercicios adicionales	6

1.1 Introducción

En este capítulo se presenta el estudio de las expresiones básicas en C++. En concreto se estudiarán los siguientes aspectos:

- Tipos de datos básicos y sus operaciones.
- Expresiones.
- Funciones de biblioteca.

A pesar de la amplia gama de tipos de datos que ofrece el lenguaje, no se estudiarán en profundidad, ya que por un lado prácticamente todos los programas que se van a crear pueden resolverse con los tipos básicos, y por otro no aportan un valor añadido en un curso introductorio. Más adelante, cuando ya se posea una base de programación y se requieran soluciones más específicas y optimizadas, será fácil incorporar todos esos conocimientos.

1.1.1 Tipos de datos y operaciones

Antes de empezar, es importante que el alumno entienda que un programa puede manejar una amplia variedad de información y, para ello, incluye la posibilidad de almacenar y operar con distintos tipos de datos. Un tipo de dato implica:

- Una representación. La información la almacena y maneja un ordenador que sólo sabe trabajar con ceros y unos. Cualquier dato que quiera introducir en un ordenador deberá transformarse a ceros y unos. Es inevitable que los datos se representen siempre en ese lenguaje binario.
Como consecuencia de esa representación, los tipos de datos que maneja el ordenador tendrán ciertas características muy concretas. Por ejemplo, debido a la limitación en la cantidad de memoria que posee el ordenador, se puede limitar el tamaño de los datos. Un caso concreto sería guardar un dato de tipo entero, que sabemos que tiene un número infinito de posibilidades, pero que en nuestros programas estará limitado a un rango finito de valores.
- Un conjunto de operaciones. El lenguaje nos ofrece, para cada tipo, un conjunto de operaciones asociado. Estas operaciones nos permiten obtener nuevos datos como resultado de operar con uno o más operandos. Por ejemplo, podemos obtener un nuevo entero como resultado de la suma de dos datos del mismo tipo.

En este capítulo vamos a introducir la creación de expresiones donde aparecen datos de distintos tipos relacionados mediante un conjunto de operadores. Para escribir nuestros programas, es fundamental entender perfectamente cómo se evalúan estas expresiones y qué resultados se obtienen.

1.2 Tipos de datos numéricos

En esta sección vamos a introducir algunos ejemplos de programas simples en C++ con la intención de que el lector asimile más fácilmente las consecuencias prácticas de los conceptos y reglas que se estudian de forma más detallada en clase de teoría.

Aunque durante la exposición se incluirán algunos contenidos para recordar y enfatizar algunos detalles clave, es recomendable que revise los contenidos teóricos antes trabajar este tema. Los ejercicios se han diseñado para hacer que el lector infiera los conceptos fundamentales del tema, evitando memorizar reglas e invitando a razonar sobre los contenidos que se presentan.

1.2.1 Entero vs real

En C++ existe una amplia variedad de tipos de datos para manejar datos numéricos. En esta sección vamos a centrarnos en distinguir dos tipos de datos: entero y real. El primero almacena un número entero, con signo, en un rango limitado y el segundo almacena un número real, que puede incluir decimales.

Cuando se escriben expresiones con números debemos tener en cuenta que:

- No es lo mismo un entero que un real. Cuando el compilador evalúa una expresión que contiene únicamente enteros, el resultado es otro entero. En otro caso, cuando tenemos dos reales, o un real y un entero, se realiza una operación entre reales.
- Es necesario tener en cuenta el orden de prioridad de los operadores. Cuando no aparece ningún paréntesis, el compilador realiza antes las operaciones de producto (“*”) y división (“/”) que las de suma (“+”) y resta (“-”). Dos de estos operadores con igual prioridad se evalúan de izquierda a derecha.

Ejercicio 1.1 — Expresiones simples. A continuación le presentamos un programa que evalúa varias expresiones simples. Intente averiguar el valor de cada una de ellas sin necesidad de ejecutarlo. Cuando lo pruebe, confirme o rectifique el razonamiento de sus respuestas.

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

int main()
{
    cout << "3+5/4= " << 3+5/4 << endl;
    cout << "(3+5)/4= " << (3+5)/4 << endl;
    cout << "3+5.0/4= " << 3+5.0/4 << endl;
    cout << "3/4*4.0= " << 3/4*4.0 << endl;
    cout << "3*4/3= " << 3*4/3 << endl;
    cout << "3/4.0*4.0= " << 3/4.0*4.0 << endl;
}
```

Observe que hemos usado una instrucción `cout <<` para cada una de las líneas de salida que obtenemos en la ejecución. Esa división en líneas se debe al identificador `endl`, que indica que se debe pasar a la siguiente línea (se inserta un final de línea). Por otro lado, fíjese que las secuencias de caracteres entre comillas dobles se obtienen en la salida exactamente igual, carácter a carácter, incluyendo los espacios.

Finalmente, podemos encadenar tantos mensajes como deseemos en una única aparición de `cout <<`, simplemente añadiendo nuevos datos junto con el correspondiente par de caracteres `<<` que los separa. De hecho, el ejemplo anterior se podría haber escrito con un único `cout`.

1.2.2 Variables

En la sección anterior hemos usado enteros y reales con un valor determinado, es decir, literales. Por ejemplo, hemos incluido los valores 3, 5, 4.0, 4 y 5.0.

Podemos declarar objetos de tipo entero y real dándoles un nombre, de forma que podamos cambiar su valor cuando deseemos. Para ello, sólo tenemos que escribir una línea de declaración que incluye el tipo de dato que queremos manejar y el nombre que le damos al dato. Por ejemplo:

```
int a;      // a es una variable entera (int)
double d;   // d es una variable real (double)
```

Los valores de estas variables pueden cambiar mediante una asignación. Asignamos un nuevo valor a una variable escribiendo el nombre, el signo `=` y una expresión cuyo valor resultante será el que se asigne. Por ejemplo:

```
a= 5;      // Asigna el valor 5 a la variable "a"
d= 4.7;    // Asigna el valor 4.7 a la variable "d"
d= d+1;    // Asigna el valor 5.7 a la variable "d"
```

donde podemos ver que, en la última asignación, se ha usado la misma variable. Primero se evalúa la expresión a la derecha del signo `=` y a continuación se actualiza el valor de la variable a la izquierda.

La mezcla de datos enteros y reales no implica ninguna dificultad para el compilador. Puede observar que un dato entero se convierte a real cuando es necesario o uno real a entero perdiendo la parte fraccionaria. Por ejemplo:

```
d= 4;      // Convierte el 4 en 4.0
d= d*2.1;  // Asigna a "d" el valor 8.4
a= d;      // Asigna a "a" el valor 8;
```

Por supuesto, sin olvidar que una operación entre dos valores enteros da lugar a un resultado de tipo entero. Así, podemos decir que una división entera “ignora el resto”.

Ejercicio 1.2 — Parte entera de un número real. Escriba un programa que lea un número real, y escriba la parte entera descartando la parte fraccionaria.

1.2.3 Constantes

Para presentar el uso de constantes, vamos a escribir un programa que lea el radio de un círculo desde la entrada estándar y escriba dos datos: el área del círculo y la longitud de la circunferencia correspondiente. El código fuente puede ser el siguiente:

```
#include <iostream> // cout, cin, endl
using namespace std;

int main()
{
    const double PI= 3.1416;

    double radio;
    cout << "Introduzca radio: "; cin >> radio;

    cout << "Resultados:" << endl;
    cout << "\tÁrea: " << PI*radio*radio << endl;
    cout << "\tLongitud: " << 2*PI*radio << endl;
}
```

A continuación vemos un ejemplo de ejecución de este programa:

```
Introduzca radio: 1
Resultados:
    Área: 3.1416
    Longitud: 6.2832
```

Para resolver este problema hemos definido la constante *PI*. Observe que la declaración es similar a la de una variable, anteponiendo la palabra reservada **const** y asignándole directamente un valor concreto que no puede cambiar durante el programa.

Por otro lado, aparece un nuevo carácter en los mensajes de resultados: el carácter '*\t*'. Con esta sintaxis indicamos que el primer carácter del mensaje es un tabulador. Por tanto, las dos líneas que ofrecen los resultados se obtendrán tabuladas hacia la derecha. Note que aunque hemos escrito dos caracteres (barra invertida + letra t), representan un único carácter.

Al igual que para el caso del tabulador, existen otros caracteres que no pueden escribirse en un mensaje -entre comillas dobles- de forma directa. Para poder escribirlos, podemos usar la barra invertida. Por ejemplo:

- El carácter salto de línea se puede escribir como '*\n*'. En la mayoría de los casos no necesitaremos usar este carácter, pues disponemos del identificador **endl** que nos permite realizarlo, al menos, de una forma más legible¹.
- Cuando ponemos la barra invertida, el compilador entiende que es un carácter especial. Si queremos incluir la barra invertida en el mensaje, podemos escribir una doble barra invertida '*\\"*'.
- Si escribimos una doble comilla en un mensaje, el compilador entendería que en ese punto acaba el mensaje. Si queremos que la doble comilla sea un carácter más del mensaje, podemos añadirle delante la barra invertida: '*\\"*'.

1.2.4 Funciones de biblioteca

Los problemas a resolver requieren, en muchos casos, de operaciones más complejas que las proporcionadas por el lenguaje. Por ejemplo, podemos necesitar realizar una raíz cuadrada, una operación trigonométrica, un logaritmo, etc. Para resolver estos cálculos, necesitaríamos crear un algoritmo específico. Sin embargo, dada su importancia, el lenguaje nos lo ofrece directamente como una función de biblioteca.

Por ejemplo, para funciones como las que hemos comentado, podemos incluir el archivo **cmath**. La inclusión se refiere a usar la directiva *include* de la misma manera que hemos hecho con el archivo **iostream**. Los archivos que incluimos con esta directiva se denominan “archivos cabecera”.

Algunas funciones incluidas en el archivo **cmath** son:

- **sin (x)**: devuelve el seno de x (en radianes).
- **sqrt (x)**: devuelve \sqrt{x}
- **pow (b, e)**: devuelve b^e

En el enlace <http://www.cplusplus.com/reference/clibrary/cmath> puede consultar otras funciones de **cmath**. Observe que el nombre de este archivo también se presenta como *math.h*. Esto se debe a que este archivo también existe en lenguaje C, donde aparece con la extensión “.h”. En este lenguaje se utiliza también la directiva **#include** con la misma funcionalidad que en C++.

Los archivos de C que se “heredan” en C++ con una funcionalidad similar aparecerán sin la extensión “.h” y con una letra “c” al principio. Por ejemplo, en ese sitio web podrá encontrar, en el índice principal (<http://www.cplusplus.com/reference>), una lista de archivos cabecera de la biblioteca C. Por ejemplo, puede encontrar la función **tolower**, para lo que tendríamos que incluir el archivo cabecera **cctype**, aunque en el caso de un programa C necesitaríamos incluir **ctype.h**.

Como ejemplo de estas funciones de biblioteca, vamos a escribir un programa que obtenga la longitud de la hipotenusa de un triángulo rectángulo a partir de la longitud de sus dos catetos. Recuerde la expresión $h = \sqrt{a^2 + b^2}$.

¹No es la única diferencia; además, permite descargar el buffer.

```
#include <iostream> // cout, cin, endl
#include <cmath> // sqrt
using namespace std;

int main()
{
    double cateto1;
    double cateto2;

    cout << "Introduzca el cateto 1: ";
    cin >> cateto1;

    cout << "Introduzca el cateto 2: ";
    cin >> cateto2;

    double hipotenusa= sqrt(cateto1*cateto1+cateto2*cateto2);

    cout << "La hipotenusa vale: " << hipotenusa << endl;
}
```

Es interesante que observe:

1. Cuando declaramos un nuevo objeto -véase hipotenusa- podemos inicializarlo con el valor de una expresión sin necesidad de escribir una nueva línea para asignarle dicho valor.
2. Para hacer el cuadrado no es estrictamente necesario el uso de la función **pow** de **cmath**, basta con multiplicar el valor por sí mismo.

Ejercicio 1.3 — Radio de un círculo. Escriba un programa que obtenga el radio de un círculo a partir de la superficie que abarca.

1.2.5 Errores de aproximación de números reales

El conjunto de los números reales es imposible de representar con un número finito de bits. Por tanto, cuando trabajamos con números reales debemos ser conscientes de que trabajamos con aproximaciones. No está garantizado que un número real concreto se pueda representar de una forma exacta dentro del ordenador.

Generalmente ignoraremos ese comportamiento, ya que la precisión que nos ofrece el tipo **double** es suficientemente buena como para resolver con éxito todos nuestros problemas. Sin embargo, es posible encontrar casos cuyo comportamiento puede parecer extraño, pero que puede entenderse perfectamente si tenemos en cuenta esta limitación.

En nuestros programas, será especialmente interesante el problema de transformar un número real en un número entero, ya que la asignación directa implica una pérdida de la parte fraccionaria.

Consideremos un nuevo problema que calcula el número de céntimos que devolver dados el precio y el dinero entregado en euros. El programa fuente puede ser el siguiente:

```
#include <iostream> // cout, cin, endl
using namespace std;

int main()
{
    double precio, entregado;

    cout << "Precio y entregado en euros: ";
    cin >> precio >> entregado;

    int centimos= 100*(entregado-precio);
    cout << "Vuelta: " << centimos << endl;
}
```

Es interesante que observe algunas novedades que aparecen en este código:

1. Se pueden declarar varias variables en una misma línea usando la coma.
2. Se pueden leer varios valores en una misma línea añadiendo nuevos operadores **>>** a una línea con **cin**.
3. No hemos añadido el acento a la palabra céntimos, pues los acentos no se admiten en el nombre de las variables.

Considere que se ha realizado una ejecución en cierto sistema —es probable que en su máquina no se obtenga el mismo valor— que ha dado lugar al siguiente resultado:



Donde podemos ver que hemos introducido en la misma línea los dos datos de entrada y el programa ha terminado sin ningún error de ejecución. Sin embargo, podemos ver que el resultado obtenido no es el correcto. Algunos autores califican como “error lógico” cuando el programa termina correctamente pero el resultado no es el deseado. Obviamente, el problema es que no hemos implementado el algoritmo como debíamos.

Ejercicio 1.4 — Cálculo de diferencia en céntimos. Comente brevemente la razón del resultado erróneo obtenido en el programa anterior y proponga una solución.

1.3 Tipo carácter

El tipo de dato que nos permite manejar un carácter es `char`. En nuestros primeros programas, usaremos este tipo de dato para manejar situaciones simples en las que intervenga un solo carácter. Más adelante se ampliarán las posibilidades cuando manejemos cadenas de caracteres.

En nuestro curso, vamos a suponer que cualquier carácter se va a representar internamente como un código ASCII. Este código requiere 8 bits únicamente, es decir 256 posibles valores para un carácter. Por tanto, cuando declaramos un objeto de tipo `char`, sabemos que internamente el compilador lo va a representar como un número que coincide con su valor ASCII².

1.3.1 E/S de un carácter

Una forma simple y directa para comprobar que el compilador trata de distinta forma un entero que un carácter es estudiar cómo se pueden realizar las operaciones de E/S de caracteres.

Ejercicio 1.5 — Lectura de enteros y caracteres. Escriba un programa que lea un entero y escriba el valor leído. Ejecútelo con un entero de varios dígitos, como “123”. Una vez realizado, cambie el tipo de la variable a `char` y vuelva a ejecutarlo con la misma entrada. Una vez comprobado el resultado, declare tres caracteres y lea/escriba los tres datos.

En el ejercicio anterior, resulta interesante estudiar el comportamiento cuando se leen tres objetos de tipo carácter. Si introduce una entrada, por ejemplo, como “1 2 3”, donde hemos introducido espacios entre los dígitos comprobará que los espacios, a pesar de ser un carácter válido, no se pueden leer. El comportamiento se debe a que la lectura interpreta los espacios como separadores, y por tanto, no podemos leerlos.

Tenga en cuenta que un `char` es un tipo que almacena un entero, ya que tiene que almacenar el código ASCII de un carácter. Por ejemplo, si asignamos el carácter ‘ ’ a un objeto de tipo `char`, tendrá un número 123 como código interno.

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

1.3.2 Funciones de biblioteca para caracteres

Al igual que hemos visto para el caso de los números reales, la biblioteca estándar de C/C++ ofrece una serie de operaciones que podemos usar para el tipo `char`. Concretamente, puede consultar el archivo de cabecera `cctype`, donde puede encontrar funciones como:

1. `isdigit (c)`, que devuelve si *c* es o no un dígito decimal,
2. `islower (c)`, que devuelve si *c* es o no minúscula,
3. `toupper (c)`, que devuelve la mayúscula que corresponde a *c*.

En el enlace <http://wwwcplusplus.com/reference/clibrary/cctype> puede consultar otras funciones incluidas dentro de este archivo de cabecera.

Si revisa estas funciones detenidamente, verá que en su descripción se usa el tipo `int` en lugar de `char`. No se trata de un error. El hecho de que un tipo carácter almacene un código ASCII, es decir, un entero codificando el carácter correspondiente hace que sea muy simple tratarlo como un número. Así, el compilador puede hacer un cálculo con un carácter -si tiene en cuenta dicho entero- o mostrar el carácter que codifica un entero -si interpreta el entero como un código ASCII-.

Es posible combinar los datos de tipo `char` con los de tipo entero, de forma que el compilador puede convertir caracteres a enteros y enteros a caracteres. Tenga en cuenta que:

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

Ejercicio 1.6 — Caracteres y código ASCII asociado. Escriba un programa que lea un carácter —en una variable de tipo `char`— y escriba en una línea `cout <<` el resultado de sumar 1 a dicho carácter. Compruebe el comportamiento para varias entradas (pruebe, por ejemplo, con alguna letra, minúscula y mayúscula y algún dígito numérico). Una vez comprobado, asegúrese de asignar el valor de esa suma a otra variable de tipo `char`, y escriba el contenido de dicha variable. ¿Qué podemos concluir?

²Más adelante veremos que podremos tratarlo como número en algunas circunstancias. A pesar de ello, un carácter es algo distinto a un entero de tipo `int`.

1.4 Expresiones complejas

Probablemente, una misma expresión se pueda escribir de varias maneras. Incluso varias expresiones se puedan compactar en una sola. La riqueza de operadores del lenguaje nos permite distintas alternativas, especialmente teniendo en cuenta el comportamiento de los operadores de asignación y los operadores de incremento y decremento.

En general, vamos a desaconsejar muchas de estas expresiones, ya que es más importante que el código sea claro y poco propenso a errores, sobre todo para alguien que comienza a programar. A pesar de ello, es interesante que las revisemos y no olvidemos que existen, puesto que cuando se es programador, no sólo se escribe código, sino que también se debe leer código de otros programadores.

Ejercicio 1.7 — Expresiones complejas. Teniendo en cuenta el siguiente trozo de código:

```
int a=1, b=2, c=3, d=4, e=5;
int res= 10;
rest= (a++*b) + ((c=d)*--e);
cout << res << ' ' << a << ' ' << b
    << ' ' << c << ' ' << d << ' ' << e << endl;
```

1. Deduzca la salida del programa sin usar el ordenador.
2. Confirme el resultado del punto anterior ejecutando el programa.
3. Modifique el programa para eliminar la asignación compuesta, la asignación como subexpresión, y los incrementos/-decrementos, obteniendo el mismo resultado. Ejecute para comprobar que es correcto.

1.5 Conversiones explícitas

Como hemos podido comprobar, cuando mezclamos distintos tipos de datos en una expresión, el compilador se encarga de analizarla y, en el caso de combinar distintos tipos, realizar las conversiones que sean necesarias. Estas conversiones se denominan implícitas, ya que se sobrentienden, y por ello las realiza el compilador de forma automática.

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

La forma más simple de hacer un casting es escribiendo:

(TIPO) EXPRESIÓN

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

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

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

Tenga en cuenta que el casting es otro operador y que tiene una prioridad alta, ya que está al nivel de los operadores unarios. Si tiene alguna duda, deberá poner la expresión entre paréntesis.

El operador de moldeado que hemos explicado es el que se usa en el lenguaje C, y que también está disponible en C++. Sin embargo, en C++ se incorporan nuevos operadores que son más seguros, ya que se diversifica con distintos tipos de casting de forma que el compilador conozca mejor nuestras intenciones. Aunque hay varios, simplemente comentaremos la sintaxis del único que usaremos en esta introducción a la programación:

`static_cast<TIPO> (EXPRESIÓN)`

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

Ejercicio 1.8 — Caracteres y número ASCII asociado. Escriba un programa que reciba como entrada un número del 0 al 25. Como resultado deberá escribir la letra (el 0 indica la 'a' y el 25 la 'z'), su correspondiente mayúscula y los dos valores ASCII correspondientes, habiendo declarado únicamente un objeto de tipo `int`.

1.6 Ejercicios adicionales

Los ejercicios que se plantean a continuación le permitirán practicar más los contenidos del tema y afianzar sus conocimientos. Se recomienda su realización.

Ejercicio 1.9 Escriba un programa que calcule el resultado de la división 5/0. Compruebe la ventana de errores de compilación así como el resultado de ejecutarlo. Considere el uso de literales de tipo `int` y de tipo `double` y observe la diferencia.

Ejercicio 1.10 Lectura de tres valores desde el teclado y salida de dos valores: la media aritmética y la desviación estándar. Las expresiones son las siguientes:

$$\bar{x} = \frac{x_1 + x_2 + x_3}{3}$$

$$\sigma = \sqrt{\frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + (x_3 - \bar{x})^2}{3}}$$

Ejercicio 1.11 Escriba un programa para convertir los grados decimales a grados, minutos y segundos. Recuerde que un grado son 60 minutos y un minuto son 60 segundos. Por ejemplo, si escribimos como entrada 34.567, deberá obtener en la salida $34^\circ 34' 1.2''$.

Ejercicio 1.12 Compruebe algunos errores de compilación arreglándolos uno a uno y recompilando el siguiente programa:

```
#include <iostream> // cout, cin, endl
#include <cmath> // sqrt

int main {
    double 1cateto;
    double 2cateto;

    cout << "Introduzca el cateto 1: ";
    cin << cateto1;
    cout << "Introduzca el cateto 2: ";
    cin >> cateto2;

    double hipotenusa= sqrt(cateto1*cateto1+cateto2*cateto2);
    cout << "La hipotenusa vale: " << hipotenusa << endl;
}
```

Ejercicio 1.13 Escriba un programa C++ que calcule el valor de la siguiente expresión:

$$\varphi = \arctan\left(\frac{c+e^2 \cdot b \cdot \sin^3 \theta}{p-e^2 \cdot a \cdot \sqrt[3]{\cos^3 \theta}}\right)$$

Por ejemplo, para los valores $a=1$, $b=2$, $c=3$, $e=4$, $p=5$ y $\theta=6$ el resultado es $\varphi=2.92803$. Para resolver este ejercicio deberá buscar información sobre las funciones de biblioteca `atan` y `atan2`. Estudie sus diferencias y determine cuál es la adecuada en este caso.

Ejercicio 1.14 La expresión que define una función gaussiana es la siguiente:

$$\text{gaussiana}(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Escriba un programa que lea los valores de la media, la desviación típica y la abscisa y que muestre a continuación el valor de la función.

Ejercicio 1.15 Escriba un programa que lea tres valores enteros por la entrada estándar y que los almacene en tres variables x , y , z . A continuación el programa procederá a rotar dichos valores colocando el valor de x en y , el de y en z y el de z en x . Finalmente el programa mostrará los nuevos valores de x , y , z .

Ejercicio 1.16 Escriba un programa que lea una letra por la entrada estándar y que muestre su correspondiente mayúscula. Si lo que escribió el usuario ya era una mayúscula o no era una letra, mostrará el mismo carácter. Busque información sobre el operador ternario `? :` para resolver este ejercicio.

Ejercicio 1.17 Escriba un programa que lea en una variable de tipo `int` un valor entero de 3 dígitos (podemos suponer que el usuario escribirá sin equivocarse un número de 3 dígitos). A continuación el programa mostrará los 3 dígitos del número: uno en cada línea.

2

La estructura de selección

Introducción	9
La instrucción <code>if simple</code>	9
La instrucción <code>if/else</code>	9
La instrucción <code>if/else</code> anidada	
Condiciones compuestas: operadores lógicos	
11	
Evaluación en corto	
La instrucción <code>switch</code>	12
Booleanos y enteros	13
Ejercicios adicionales	13

2.1 Introducción

El objetivo de este guión es que el alumno resuelva varios problemas que permitan practicar con:

1. La instrucción `if` simple. Es decir, el uso de estructuras de selección sin la parte `else`.
2. La instrucción `if/else`. En este caso aparecen dos conjuntos de instrucciones, uno para cuando la condición se cumple y otro para el caso contrario.
3. La instrucción `switch`. Aunque es una instrucción que puede resolverse con las anteriores, es recomendable tenerla en cuenta ya que, cuando tenemos una selección múltiple, puede resultar un código más sencillo y comprensible
4. El tipo de dato `bool` y las expresiones condicionales, incluyendo operadores relacionales y lógicos.

2.2 La instrucción `if simple`

Para practicar con esta instrucción vamos a proponer un ejemplo muy sencillo.

Ejercicio 2.1 — Cálculo del sueldo. En una empresa hay dos tipos de empleados: (F) fijos y (T) temporales. Se establece que el sueldo base es la cantidad bruta que cobran los temporales. Los trabajadores fijos tienen un incremento de 200 euros. A partir del sueldo bruto, hay que calcular el neto descontando un 17% de impuestos. Si el sueldo bruto supera los 1200 euros entonces los impuestos se incrementarán en un 3%.

Escriba un programa que le pregunte al usuario cuál es el sueldo base establecido para los trabajadores y el tipo de trabajador que es (en ese orden). Como resultado, el programa le indicará cuál es su sueldo bruto y su sueldo neto.

El tipo de trabajador se indicará escribiendo una letra (T o F).

Este programa también nos permite conocer mejor el comportamiento de la lectura desde la entrada estándar (`cin>>`). Como hemos visto hasta ahora, los datos leídos mediante esta instrucción están separados por espacios, saltos de línea, tabuladores, etc. En realidad, los datos pueden no estar separados por estos caracteres.

Ejercicio 2.2 — Datos sin separación. Compruebe el comportamiento del programa anterior si, cuando pregunta el primer dato (sueldo base), escribimos de forma conjunta sueldo base y tipo de trabajador (sin separación). Por ejemplo, si el sueldo fuese de 1065 euros y el trabajador fuese de tipo fijo escribiríamos 1065F.

Observará que la instrucción de lectura a veces no necesita que separemos los datos por espacios. Esto se debe a que la lectura de este tipo implica que sólo se “usan” aquellos caracteres que coinciden con el tipo del dato que estamos leyendo y, por tanto, no siempre se necesita encontrar un espacio para separar los datos.

2.3 La instrucción `if/else`

Esta instrucción permite alterar el flujo de control para seguir uno de entre dos posibles caminos.

Ejercicio 2.3 — Conversión de escalas. Escriba un programa que permita traducir entre grados *Celsius* (C), *Fahrenheit* (F), *Kelvin* (K) y *Rankine* (R). El programa preguntará en qué unidades damos la temperatura de entrada y a qué escala queremos convertir. Para ello sabemos que:

$$K = C + 273{,}15 \quad R = F + 459{,}67 \quad 9 \cdot C = 5(F - 32)$$

Tenga en cuenta que el programa pregunta la temperatura y que ésta se introduce como un número seguido de dos letras que indican las unidades. Por ejemplo: 35CF indica que queremos pasar 35° *Celsius* (C) a grados *Fahrenheit* (F).

Importante: no se permite usar operadores lógicos (`&&`, `||`, `!`). Posiblemente la primera idea que nos viene a la cabeza para resolver este problema es establecer las fórmulas para convertir de todas a todas las escalas. Esto nos da un total de $4 \times 4 = 16$ fórmulas diferentes (si tuviésemos más escalas, la cantidad de fórmulas aumenta rápidamente). Esta solución, además, necesitaría el uso de condiciones compuestas (que usan operadores lógicos). Debe pensar en una solución alternativa.

Para comprobar el resultado puede consultar este sitio web:

<http://www.calculatorsoup.com/calculators/conversions/temperature.php>

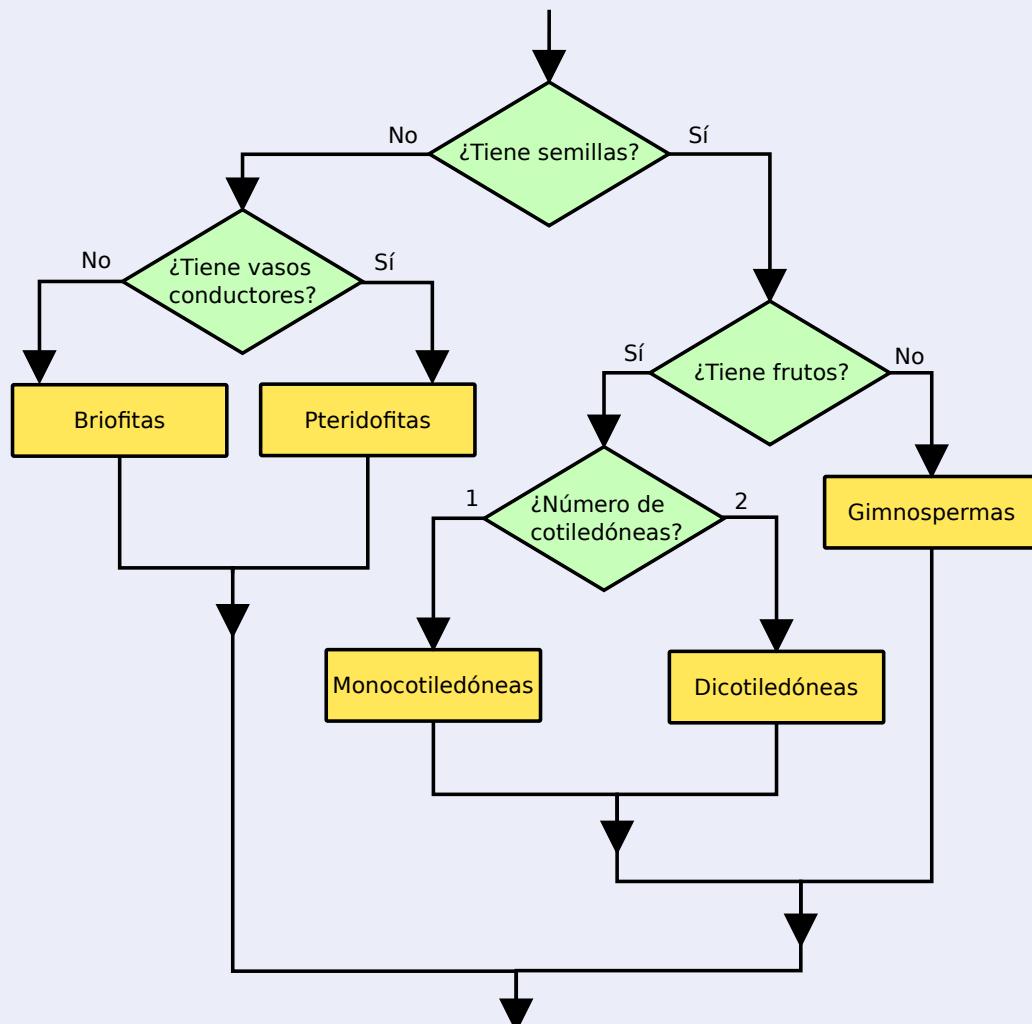
2.3.1 La instrucción if/else anidada

Esta sección ilustra el uso de instrucciones condicionales **if/else** anidadas. Recuerde que el anidamiento consiste en la inclusión de una condicional dentro de otra y que ésta puede hacerse tanto en la parte **if** como en la parte **else**.

Además, es posible realizar anidamientos de varios niveles, es decir, incluir condicionales que, a su vez, están incluidas dentro de otras.

Ejercicio 2.4 — Clasificación de plantas. Un profesor de Biología de enseñanzas medias desea disponer de un programa para clasificar plantas en función de algunas de sus características para facilitar el aprendizaje de los estudiantes. El programa irá haciendo preguntas conforme las vaya necesitando y finalizará indicando el tipo de planta del que se trata.

El árbol de decisión que determina la planta a partir de las características es el siguiente:



Escriba un programa que usando la instrucción **if/else** anidada implemente el comportamiento deseado conforme a las preguntas de la figura. A continuación puede ver dos posibles ejecuciones:

```
Consola
¿Tiene semillas? n
¿Tiene vasos conductores? s
La planta es de tipo: Pteridofitas
```

```
Consola
¿Tiene semillas? s
¿Tiene frutos? s
¿Cuántas cotiledóneas tiene (1 ó 2)? 1
La planta es de tipo: Monocotiledóneas
```

2.4 Condiciones compuestas: operadores lógicos

Las instrucciones **if/else** incluyen una condición que puede ser el resultado de una expresión compuesta con operadores lógicos, es decir, incluyendo el operador Y –o– el operador O– sobre dos valores booleanos. En esta sección, se proponen problemas para los que el alumno deberá usar las condiciones compuestas en estructuras condicionales.

Ejercicio 2.5 — Pascua. La fecha del domingo de Pascua corresponde al primer domingo después de la primera luna llena que sigue al equinoccio de primavera. El algoritmo que se presenta a continuación (denominado *Cómputus*) permite calcular esta fecha y es válido para años comprendidos entre 1900 y 2100. Para un determinado año, los cálculos que hay que realizar son:

- A = año mod 19
- B = año mod 4
- C = año mod 7
- D = $(19 * A + 24) \text{ mod } 30$
- E = $(2 * B + 4 * C + 6 * D + 5) \text{ mod } 7$
- N = $(22 + D + E)$

El valor de N corresponde al día de marzo en el que se sitúa el domingo de Pascua. En el caso de que sea mayor que 31, el valor se refiere a un día de abril. Además, hay dos excepciones:

1. Si la fecha obtenida es el 26 de abril, entonces la Pascua caerá en el 19 de abril.
2. Si es el 25 de abril, con D = 28, E = 6 y A > 10, entonces la Pascua es el 18 de abril.

Escriba un programa que lea un año y muestre el día y mes en el que se celebró o celebrará el domingo de pascua para ese año.

En la siguiente lista tiene algunos datos para verificar que el ejercicio es correcto:

- Año 2005 ⇒ Pascua el 27 de marzo
- Año 2011 ⇒ Pascua el 24 de abril
- Año 2049 ⇒ Pascua el 18 de abril
- Año 2076 ⇒ Pascua el 19 de abril

Ejercicio 2.6 — Rectángulo. Escriba un programa que lea las coordenadas que definen un rectángulo –dos esquinas opuestas– y las coordenadas de un punto en el espacio. A continuación, el programa escribe en la salida estándar si está en el interior, en el exterior, o en el borde del rectángulo determinado por los dos puntos introducidos. Además, tenga en cuenta que no sabemos las esquinas que introduce el usuario, simplemente, que son esquinas opuestas.

Nota: No use operadores aritméticos en la solución.

Ejercicio 2.7 — Fecha correcta. Escriba un programa que lea una fecha –día, mes y año– y compruebe si es válida. Para ello, use una única instrucción if/else (con una condición compuesta) que escriba si es correcta o no.

Para que una fecha sea correcta se debe cumplir lo siguiente:

1. El año debe ser mayor que cero.
2. El mes debe estar entre 1 y 12.

3. El día debe ser un valor entre 1 y un valor máximo determinado por el mes y año. Concretamente, tiene que tener en cuenta que:
- Los meses que tienen 31 días son 1, 3, 5, 7, 8, 10 y 12.
 - Los meses que tienen 30 días son 4, 6, 9 y 11.
 - El mes 2 tiene 28 días, excepto los años bisiestos.
 - Un año es bisiesto si es divisible por 4 y no por 100, o si es divisible por 400.

2.4.1 Evaluación en corto

En la mayoría de los casos, el orden de los operandos en las condiciones que incluyen operadores lógicos no es importante, ya que el resultado es el mismo. Sin embargo, pueden aparecer casos en los que sea necesario escribirlo con cierto orden, y aprovechar la evaluación en corto para evitar un error en tiempo de ejecución.

Esta situación se puede dar cuando una condición incluye un operador lógico sobre dos expresiones, y la primera expresión determina una condición que -dependiendo de su valor- podría implicar que no se puede evaluar la segunda.

Ejercicio 2.8 — Divisor. Escriba un programa que lea dos números enteros y escriba en la salida estándar si el segundo divide al primero. Para ello, use una instrucción **if/else** con una condición simple que hace uso del operador módulo (%) para saber si la condición se cumple.

Compruebe el resultado de este programa si damos como segundo número el cero.

Ejercicio 2.9 — Divisor correcto. Como habrá comprobado, el funcionamiento del programa del ejercicio 2.8 es incorrecto si el segundo número es cero. Solucione este problema. Para ello, modifique la condición de la instrucción **if/else** de forma que ahora sea una condición compuesta que verifique que:

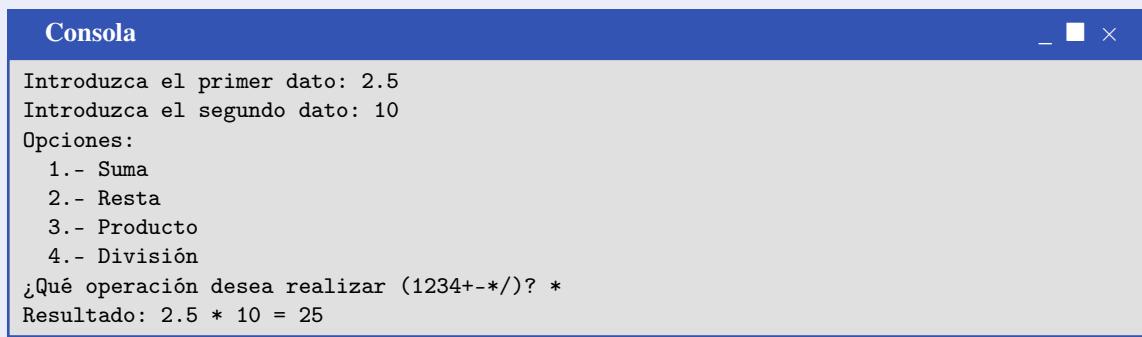
- El segundo número divide al primero (usando el operador %).
- El segundo número es distinto de cero.

Compruebe si el orden de las subexpresiones afecta al funcionamiento del programa.

2.5 La instrucción switch

Cuando el flujo de control tiene que seleccionar entre múltiples caminos dependiendo del valor de una expresión, la instrucción **switch** puede dar lugar a un código muy simple y claro. Un ejemplo típico es la selección de entre un conjunto de posibles valores de un menú.

Ejercicio 2.10 — Calculadora. Escriba un programa que lea dos valores reales y que presente un menú con cuatro alternativas: *Suma*, *Resta*, *Producto* y *División*. El programa deberá pedir al usuario que seleccione una de las cuatro y presentar en la salida estándar el resultado de la operación correspondiente. Tenga en cuenta que el usuario puede responder a la selección tanto con alguno de los 4 dígitos “1234” como con cualquiera de los cuatro operadores “+-*/”. Si la respuesta no corresponde a ninguno de ellos, terminará indicando que la selección es incorrecta. A continuación se presentan dos ejemplos de ejecución.



```

Consola
Introduzca el primer dato: 2.5
Introduzca el segundo dato: 10
Opciones:
 1.- Suma
 2.- Resta
 3.- Producto
 4.- División
¿Qué operación desea realizar (1234+-*/)? *
Resultado: 2.5 * 10 = 25

```

```

Consola
Introduzca el primer dato: 1
Introduzca el segundo dato: 2
Opciones:
 1.- Suma
 2.- Resta
 3.- Producto
 4.- División
¿Qué operación desea realizar (1234+-*/)? A
Error, la opción elegida no es correcta

```

2.6 Booleanos y enteros

El tipo `bool` es un tipo que no existe en C, y se añade al lenguaje C++. En el primero, el manejo de valores de tipo booleano (`true/false`) se realiza mediante algún tipo integral, codificando el valor `false` como cero y el valor `true` como distinto de cero. Cuando se diseñó el lenguaje C++, también se incluyó la posibilidad de usar los tipos de datos integrales para controlar un valor booleano y conseguir así que el código C fuera compatible con el nuevo lenguaje. Por ejemplo, si prescindimos del tipo `bool`, podríamos escribir:

```

#include <iostream>
using namespace std;

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

```

Para entender la relación entre `bool` y los tipos integrales, debemos tener en cuenta que en éstos un valor cero representa `false` y un valor distinto de cero a `true`. Por el contrario, el valor de tipo `bool false` se puede convertir a entero como cero, y el valor `true` como uno.

Ejercicio 2.11 — Conversión entre bool y entero. Considere el siguiente programa:

```

#include <iostream>
using namespace std;

int main()
{
    if (4<1<5)
        cout << "Ordenados" << endl;
    else
        cout << "Desordenados" << endl;
}

```

Compruebe su comportamiento. ¿Qué está ocurriendo?

2.7 Ejercicios adicionales

Ejercicio 2.12 Haga un programa que lea las coordenadas de tres puntos en un espacio cartesiano 2D y que nos diga si forman o no un triángulo. Además, debe indicar si el triángulo es equilátero. Si llamamos a los puntos A, B y C, sabemos que forman un triángulo cuando la mayor de las distancias AB, AC, BC es menor que la suma de las otras dos. El triángulo es equilátero si las tres distancias son iguales.

Ejercicio 2.13 Escriba un programa que lea un valor entero desde la entrada estándar y que, según sea su valor, escriba alguno (o varios) de los siguientes mensajes:

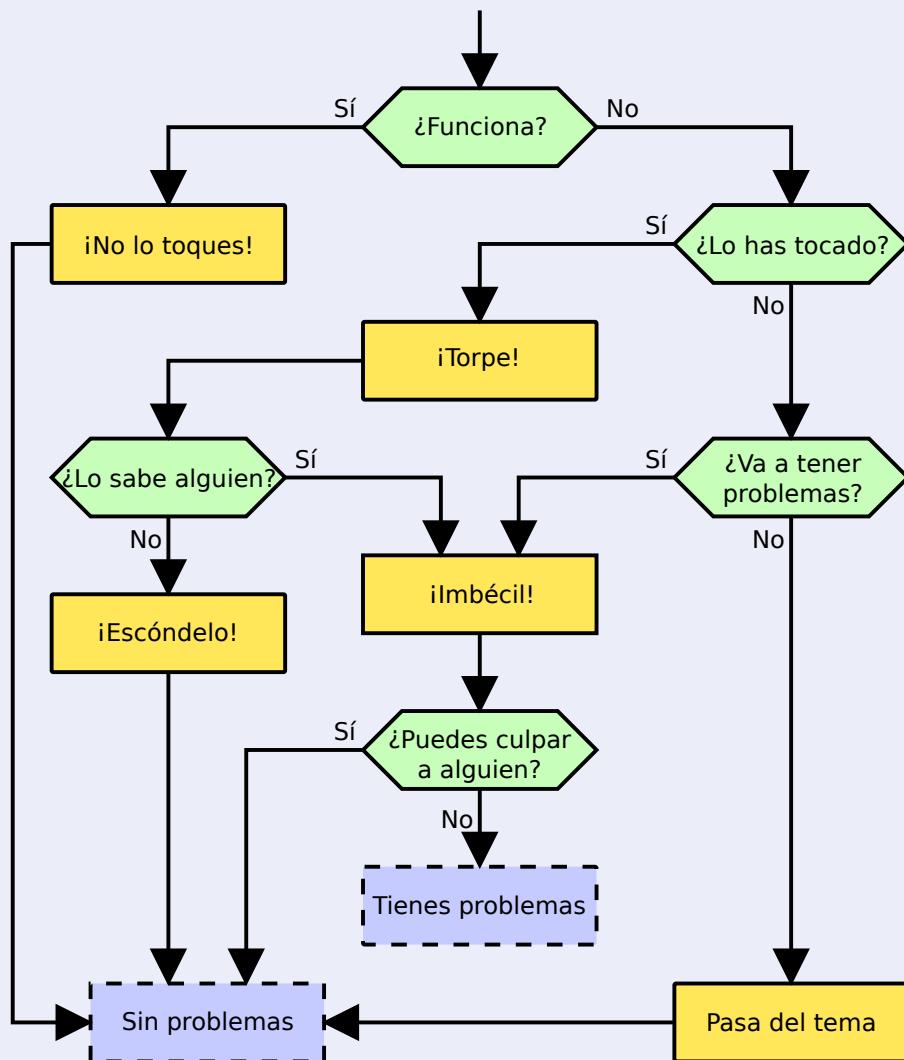
- Si es negativo escribirá “Es negativo”.
- Si está entre 0 y 5 escribirá “Si fuese una nota sería suspenso”.
- Si vale 2, 4, 8 ó 6 escribirá “Es un número positivo, par y menor que 9”.
- Si no cumple ninguna de las anteriores dirá “Número extraño”.

Ejercicio 2.14 Reescriba el ejercicio 2.3 usando la sentencia `switch`.

Ejercicio 2.15 Escriba un programa que lea un carácter desde la entrada estándar y que diga si:

- Es un dígito.
- Es una letra mayúscula.
- Es una letra minúscula.
- Es un operador aritmético (+, -, *, /)
- Es otra cosa.

Ejercicio 2.16 — Sin problemas. Un amigo informático con baja autoestima nos ha encargado que le hagamos un pequeño programa de “autoayuda” para ejecutar cada vez que realiza un cambio en la base de datos de su empresa. Tras un análisis de las preguntas y mensajes que el programa debería realizar, se ha llegado a la conclusión de que el siguiente gráfico indica todos los posibles diálogos con el usuario:



Como puede observar, se compone de una serie de preguntas a las que se responde con “sí” o “no”, y una serie de mensajes que simplemente se presentan, entre los que podemos destacar los dos mensajes terminales (“Sin problemas” o “Tienes problemas”).

Escriba un programa en el que se van presentando al usuario las distintas preguntas y mensajes que aparecen el en gráfico anterior. Tenga en cuenta que el usuario puede responder afirmativamente a cada pregunta con las 4 letras “sSyY” (cualquier otra indica una respuesta negativa). A continuación puede ver dos posibles ejecuciones:

Consola

```
¿Funciona? n  
¿Lo has tocado? n  
¿Vas a tener problemas? n  
Pasa del tema.  
Sin problemas.
```

Consola

```
¿Funciona? s  
¡No lo toques!  
Sin problemas.
```

Ejercicio 2.17 Escriba un programa que haga las funciones de una calculadora básica: suma, resta, multiplicación y división. Para ello, el programa debe leer dos números enteros y un carácter que indique la operación a realizar (+, -, *, /), mostrando el resultado a continuación. Por ejemplo, ante esta entrada:

34 12 +

el programa mostrará esta salida:

46

Ejercicio 2.18 Escriba un programa que lea un valor real correspondiente a un pago en euros y que nos indique el número mínimo de monedas y billetes necesarios para pagar dicha cantidad. Debe asumir que únicamente disponemos de monedas y billetes de 5eur, 2eur, 1eur, 50cts, 20cts, 5cts, 1cts. No tenemos billetes de mayor cuantía ni monedas de 10cts ni 2cts.

Por ejemplo, si el valor a pagar son 7'37 euros, el programa dirá que necesitamos: 1 billete de 5eur, 1 moneda de 2eur, 1 moneda de 20cts, 3 monedas de 5cts y 2 monedas de 2cts.

3

La estructura de iteración

Introducción.....	17
Patrones de diseño de bucles	17
El bucle do-while	
El bucle while	
Soluciones habituales con while	
¿Repetir mientras o repetir hasta?	
El bucle for.....	21
Inicialización, condición e incremento	
Ejecutando un cierto número de veces	
Omitiendo expresiones	
Anidamiento de bucles.....	22
Variables lógicas y condiciones compuestas	23
Casos generales y casos límite	24
Algunos errores comunes.....	24
Funciona, pero mejor no hacerlo	
Ejercicios adicionales.....	26

3.1 Introducción

El objetivo de este tema es que el alumno practique con problemas para los que se necesitan estructuras iterativas. Concretamente, en este tema abordaremos:

1. El uso de las tres estructuras de iteración: **do—while**, **while** y **for**.
2. Patrones de diseño de algoritmos iterativos tales como la entrada de datos, la selección de opciones de un menú, la entrada adelantada o los bucles contador.
3. Redirección de E/S como una forma de agilizar el desarrollo de programas que requieren una gran cantidad de entradas.
4. Algunos errores comunes en la implementación de algoritmos iterativos.

Lógicamente, no es la intención de este tema que el alumno considere los contenidos como una lista de recetas “mágicas” que le permiten enfrentarse a cualquier problema. El conocer una lista enumerada de posibles soluciones no puede sustituir la capacidad de un programador cuando tiene que resolver un problema. El hecho de resolver un problema y –sobre todo– de obtener una buena solución, deben ser consecuencia de la experiencia en la resolución de múltiples y variados problemas.

3.2 Patrones de diseño de bucles

El lector ha estudiado en teoría tres posibles estructuras de iteración: **do—while**, **while** y **for**. Además, hemos estudiado que cualquiera de ellas es prescindible, pues si eliminamos dos de ellas podríamos escribir una solución haciendo uso de la tercera (posiblemente, con la ayuda de la estructura de selección **if/else**).

En la práctica, para implementar un algoritmo, un tipo de bucle se puede adaptar mejor que otro. Cuando hablamos de adaptarse mejor nos referimos a que el código resultante es más fácil de implementar, más fácil de leer, y por lo tanto, más fácil de mantener, es decir, más fácil de depurar, modificar y ampliar. Desde este punto de vista, podemos afirmar que hemos obtenido un mejor diseño.

En general, el concepto de “patrones de diseño” se utiliza de manera generalizada en el mundo de la programación, aunque suele referirse a conceptos más específicos de diseño y menos cercanos a la implementación en un lenguaje concreto. A pesar de ello, resulta interesante enfatizar ya desde los primeros momentos que un mismo problema tiene distintas soluciones, y que unas son mejores que otras. Por tanto, resulta conveniente enfocar la exposición de algunas implementaciones como patrones, es decir, como esquemas que se repiten en distintas soluciones.

3.2.1 El bucle do-while

El bucle **do—while** se caracteriza especialmente porque su cuerpo se ejecuta, al menos, una vez. Dado que la comprobación de la condición se realiza al final, el programa habrá pasado por el cuerpo del bucle antes de llegar a decidir si se repite o no.

En la práctica, este tipo de bucle se usa con menor frecuencia que el resto. El problema, tal vez, es que se puede considerar una fuente de errores y confusión (véase [3], página 142).

En cualquier caso, parece claro que no es “fácil de leer”, ya que la condición viene al final, y el programador que lee el código lo lee de arriba hacia abajo y, por lo tanto, cuando llega a un bucle **do—while** descubre que hay una estructura iterativa y que debería saltar más abajo para enterarse exactamente de cuál es la intención del bucle antes de seguir leyendo.

Todo ello lleva a que muchos programadores usen este tipo de bucle cuando es muy clara la conveniencia de ejecutar al menos una vez el código. Un ejemplo típico es la lectura de un dato hasta que se cumpla una condición. Está claro que hay que leer el dato, al menos, una vez.

Por ejemplo, suponga que queremos realizar la lectura de una letra con la condición de que debe ser mayúscula. Una posible implementación sería la siguiente:

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

int main()
{
    char letra;
    do {
        cout << "Introduzca una letra en mayúsculas: ";
        cin >> letra;
    } while (letra<'A' || letra>'Z');
    cout << "La letra es: " << letra << endl;
}
```

Se puede ver, claramente, que resulta conveniente realizar la lectura antes de comprobar la condición. Una vez que la hayamos leído, podemos preguntarnos si tenemos que repetirla.

Ejercicio 3.1 — Lectura con do-while. Escriba un programa para leer un entero positivo. Para ello, use el bucle **do-while** de forma similar al que hemos presentado para la lectura de una letra mayúscula.

Menús y el bucle do-while

Una situación que tal vez resulte aún más habitual en el uso de este tipo de bucle es la implementación de un menú. En este caso, la idea es seleccionar una opción y ejecutar la acción asociada de forma continua. La finalización del algoritmo viene determinada por la selección de una opción “*Fin*” del menú. Obviamente, es necesario preguntar al usuario por la opción deseada antes de comprobarla.

Ejercicio 3.2 — Menú y circunferencia. Realice un programa que permita realizar algunos cálculos sobre una circunferencia. En concreto calculará el área del círculo y el perímetro correspondiente. El programa presentará un menú de opciones al usuario como el siguiente:

1. Cambiar el valor del radio
2. Mostrar el valor del radio
3. Calcular el área del círculo
4. Calcular el perímetro de la circunferencia
5. Finalizar el programa

Al comienzo, el radio de la circunferencia se inicializará a 1. La estructura del programa consistirá en un bucle de tipo **do-while** cuyo cuerpo consistirá en presentar el menú, leer una opción desde la entrada estándar y finalmente ejecutar y mostrar el resultado de la acción seleccionada mediante una estructura de bifurcación **switch**.

3.2.2 El bucle while

La característica que diferencia a este bucle del bucle **do-while** es que la comprobación para ver si hay que iterar o no se hace previa a la ejecución del cuerpo del bucle. Por este motivo cabe la posibilidad de que no llegue a iterar ninguna vez si la condición es falsa al comenzar el bucle.

Podríamos considerar que este bucle, por tanto, abarca más posibilidades que el bucle **do-while**. El número de iteraciones será siempre mayor o igual a cero mientras que en el caso del **do-while** el número de iteraciones es mayor o igual a uno. Probablemente, este es uno de los motivos por los que el bucle **while** se usa más que el bucle **do-while**.

Forzado de la primera iteración de while

Aunque la teoría dice que el bucle **do-while** se ejecuta al menos una vez, no debe pensar que se debe optar por éste en caso de que sepamos que se debe ejecutar al menos una vez. El bucle **while** también se ejecutará una vez si, al llegar a él, sabemos que la condición se evalúa como **true**. En estos casos, es posible que nos parezca más legible y fácil de entender una solución con un bucle **while**.

Ejercicio 3.3 — Forzar la primera iteración del bucle while. Considere el ejercicio 3.1. Sustituya el bucle **do-while** por otro de tipo **while**. Debe realizar este ejercicio sin añadir nuevas instrucciones de lectura de datos (**cin>>**).

Ejercicio 3.4 — Procesamiento de datos con while. Amplíe el ejercicio 3.3 con el código necesario para contar el número de dígitos que componen el número que se ha leído. Debe usar un bucle de tipo **while** para esta tarea.

Lectura de datos con centinela

Como hemos visto antes, es frecuente usar el bucle **do-while** para filtrar los datos de entrada de un programa de una forma simple. También es muy frecuente encontrar problemas en los que hay que procesar una secuencia de datos que

termina con un dato particular, al que denominamos “*centinela*”, y que indica que han finalizado los datos de entrada. En estas situaciones es habitual usar un bucle de tipo **while**.

Ejercicio 3.5 — Lectura y procesamiento de datos con centinela. Escriba un programa que lea una serie de números enteros positivos y que al terminar muestre la sumatoria de todos ellos. Para terminar, el usuario dará un valor negativo. Por ejemplo, si la secuencia de datos que da el usuario es esta:

2 6 9 3 -1

el programa dirá que su suma es 20 (observe que el valor -1 no se ha sumado al considerarse una marca especial para finalizar la introducción de datos). Considere la posibilidad de usar **while** o **do-while** y razoné cuál de los dos es más adecuado.

Lectura adelantada o anticipada

La solución al ejercicio 3.5 que usa un bucle de tipo **do-while** tiene el inconveniente de que repite dos veces la comprobación sobre la validez de los datos en cada iteración, lo cual es ineficiente. En la solución con **while** hay que tener cuidado con la primera iteración ya que antes de entrar al bucle se comprueba la condición del mismo por lo que las variables que intervienen en la misma deben ser inicializadas adecuadamente.

Una solución habitual consiste en modificar el código repitiendo la operación de lectura antes de llegar al bucle. Cuando optamos por esta solución, decimos que realizamos una lectura “*adelantada*” o “*anticipada*”.

Llamamos *lectura anticipada* a la que se realiza antes del bucle para garantizar que la primera comprobación nos permita distinguir si ese primer dato ya es correcto. Un posible esquema de una lectura adelantada es el siguiente:

```
leer primer dato
while (no acabar) {
    procesar dato
    leer siguiente dato
}
```

Esta opción parece ser contraria a un buen diseño, ya que el hecho de repetir lo mismo en dos lugares distintos puede indicar que la solución es mejorable. Sin embargo, podríamos decir que éste no es el caso, ya que queremos distinguir dos cosas distintas: la lectura del primer dato y del resto.

Ejercicio 3.6 — Lectura anticipada con while. Modifique el programa del ejercicio 3.5 para implementar la lectura anticipada con el bucle **while**.

Este esquema de lectura anticipada también se usa para las entradas de datos sencillas en las que deseamos distinguir entre el mensaje que se muestra al usuario en la lectura del primer dato y los mensajes sucesivos para cuando el usuario da un valor erróneo.

Ejercicio 3.7 — Lectura anticipada para mostrar mensajes de entrada diferentes. Se desea implementar un programa que lea un valor entero en el rango [0,100]. La primera petición del dato deberá mostrar el mensaje “*Escriba un valor entero*”. Si el usuario escribe un valor fuera del rango [0,100] el programa mostrará el mensaje “*El número debe estar en el intervalo [0,100]. Por favor, escriba otro número*” y pedirá de nuevo el dato.

Siga practicando el esquema de lectura adelantada con el siguiente ejercicio:

Ejercicio 3.8 — Lectura anticipada con centinela. Se desea implementar un programa que cuente la cantidad de datos que son positivos y negativos en una secuencia de valores enteros. Implemente un programa que lea datos hasta que se lea el valor cero. Como resultado, indicará en la salida estándar cuántos han sido positivos y cuántos negativos.

3.2.3 Soluciones habituales con while

Normalmente, el bucle **while** se utiliza en casos en los que se necesita repetir cero o más veces un trozo de código –cuerpo del bucle– y sólo necesitamos establecer la condición sin necesidad de buscar ninguna estructura especial.

Un ejemplo, para este tipo de problema, es el algoritmo de bisección para resolver una ecuación del tipo $f(x) = 0$. En el caso en que la función sea continua y se conozca un intervalo $[a, b]$ en el que el valor de la función cambia de signo, es decir $f(a) * f(b) < 0$, entonces es posible calcular el valor de la solución con un error máximo tan pequeño como queramos.

El algoritmo consiste en ir acotando por un intervalo cada vez más pequeño la solución. Para ello, podemos calcular el valor central del intervalo y evaluar la función en ese punto. Con ese cálculo, podemos decidir si la solución está en la primera mitad o en la segunda, seleccionando aquel sub-intervalo que garantiza que la función cambia de signo en sus extremos.

Por ejemplo, suponga la ecuación $f(x) = 0$, y dos valores a y b que determinan la localización de una solución de la ecuación. Podemos establecer los siguientes pasos para hacer que el tamaño del intervalo se divida por dos:

- $c = (a + b)/2$
- Si $f(a) * f(c) <= 0$ entonces $b = c$
- Si $f(c) * f(b) <= 0$ entonces $a = c$

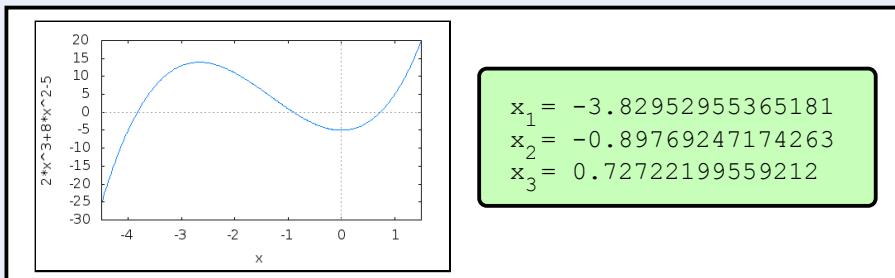
Si repetimos esa secuencia de pasos, podemos hacer el tamaño del intervalo tan pequeño como queramos. Así, si queremos dar una solución aproximada con un cierto error ϵ , basta repetir esa división hasta que el tamaño del intervalo sea menor que dicho error, es decir, hasta que $b - a < \epsilon$.

Como habrá intuido, se puede diseñar un algoritmo iterativo para calcular esa solución. En cada iteración se dividirá el tamaño del intervalo por dos, de forma que podemos iterar hasta que el tamaño sea suficientemente pequeño como para poder indicar el valor de la solución con un error máximo:

- Solución: $(a + b)/2$
- Error máximo: $(b - a)/2$

Es decir, la solución es el valor central del intervalo que hemos reducido y el error máximo es la mitad del tamaño de dicho intervalo.

Ejercicio 3.9 — Solución al algoritmo de bisección con el bucle while. Se desean calcular las raíces del polinomio $2x^3 + 8x^2 - 5 = 0$. A continuación vemos una representación gráfica de esta función junto con las soluciones exactas de la misma:



Escriba un programa para calcular una solución con la precisión que se deseé. Para ello, el programa leerá los dos extremos del intervalo (a, b) y un valor que indique el error máximo de la solución (ϵ) y aplicará el método de bisección para indicar –en la salida estándar– el valor aproximado de la solución, así como el número de iteraciones que ha necesitado para calcularlo.

Observe que si queremos las tres soluciones de la ecuación anterior, tendremos que ejecutar el programa tres veces. En cada ejecución indicaremos un intervalo en el que sepamos que está una de las soluciones. A modo de ejemplo puede probar con los intervalos $[-4, -3]$, $[-2, 0]$ y $[0, 1]$.

3.2.4 ¿Repetir mientras o repetir hasta?

Una cuestión que en muchas ocasiones causa problemas cuando se comienza a programar con estructuras iterativas es la confusión en la interpretación de la condición del bucle. Como se ha visto, los bucles en C++ iteran *mientras* la condición sea cierta. Sin embargo, a veces es más natural que pensemos en la condición que debe cumplirse para finalizar la iteración. En este caso estaríamos pensando con la lógica opuesta: iterar *hasta* que se cumpla cierta condición.

Por ejemplo, si deseamos crear un bucle que lea datos desde la entrada estándar hasta que uno de ellos sea negativo, haríamos esto:

```
int num;
do {
    cin >> num;
} while (! (num<0)); // Equivale a (num>=0)
```

es decir, hemos escrito el bucle con la condición contraria a la que hemos expresado con palabras (iterar *mientras* no sea negativo). En el texto hemos indicado *hasta* cuándo deseamos iterar pero el lenguaje sólo nos deja expresar el problema en términos de que iteremos *mientras* ocurra algo.

En este ejemplo la transformación entre una condición de tipo *hasta* por otra de tipo *mientras* ha sido sencilla: simplemente se ha negado. Si la condición es más compleja e incluye operadores lógicos como `&&` o `||` la solución sigue siendo la misma: escribir la condición opuesta. En este caso pueden ser útiles las leyes de *De Morgan* para simplificar las expresiones resultantes o incluirnos más claridad. Estas leyes dicen que:

- $! (a \mid\mid b) \equiv !a \ \&\& \ !b$
- $! (a \ \&\& \ b) \equiv !a \mid\mid !b$

Si en el ejemplo anterior deseamos leer números hasta que se lea un negativo o hasta leer el valor 4, la condición del **while** quedaría como `(! (num<0 || num==4))`, que aplicando las leyes de De Morgan se podría transformar en `(num>=0 && num!=4)`.

Ejercicio 3.10 — Repetir mientras o hasta. Escriba un programa que repite la lectura de un número hasta que o bien el número sea múltiplo de 3, o bien el número sea positivo e impar. Escriba la condición en términos de *hasta* y use las leyes de *De Morgan* para simplificarla.

3.3 El bucle for

Con los bucles que hemos expuesto en las secciones anteriores podríamos escribir cualquier programa en C++. Sin embargo, el lenguaje también define el bucle **for**, que se adapta mucho mejor a algunas situaciones, generando un código mucho más fácil de entender y modificar.

3.3.1 Inicialización, condición e incremento

El bucle **for** se caracteriza porque incluye tres expresiones: inicialización, condición e incremento. Conceptualmente sigue siendo un bucle de tipo **while** como en los casos anteriores, es decir, que sigue iterando “*mientras*” se cumpla la condición.

Ejercicio 3.11 — Tabla de equivalencia entre dólares y euros. En un banco se necesita disponer de una tabla de conversión entre dólares y euros. Escriba un programa que muestre dicha tabla para un cierto número de dólares. Para ello, el programa pedirá desde la entrada estándar el cambio de moneda en curso –cuántos euros vale un dólar– y el número de dólares para los que se desea tener la conversión. Utilice un bucle de tipo **for** para resolver el problema.

Una posible ejecución sería la siguiente:

Consola

– □ ×

¿Cuántos euros vale un dólar? 0.733
¿Límite de cambio? 12
Tabla de cambio:
1 \$ = 0.733 €
2 \$ = 1.466 €
3 \$ = 2.199 €
4 \$ = 2.932 €
5 \$ = 3.665 €
6 \$ = 4.398 €
7 \$ = 5.131 €
8 \$ = 5.864 €
9 \$ = 6.597 €
10 \$ = 7.33 €
11 \$ = 8.063 €
12 \$ = 8.796 €

Como habrá comprobado si ha resuelto problemas con el bucle **while**, en muchos casos se repite un mismo esquema:

1. Hay que hacer una inicialización antes del bucle.
 2. El bucle comprueba únicamente una condición de tipo “*mientras*”.
 3. Al final del bucle hay que actualizar una variable para preparar la siguiente iteración.

El bucle **for** nos permite escribir código más fácil de leer y mantener al unir todos los elementos que indican la lógica de control del bucle en una misma línea. Además, no sólo unimos los tres elementos que indicábamos antes, sino que podemos hacer que la declaración de la variable se incluya dentro de la parte de inicialización, haciendo que el ámbito de la variable sea solamente el cuerpo del bucle. Vemos a continuación la equivalencia entre ambos tipos de bucles:

```
Inic  
while (Condic) {  
    Cuerpo del bucle  
    Incr  
}  
  
for (Inic; Condic; Incr) {  
    Cuerpo del bucle  
}
```

Es interesante enfatizar la idea de diseño que hemos lanzado en esta sección, y que aparecerá a distintos niveles cuando se plantee una discusión sobre un buen diseño. Cuando escribimos un programa, resulta recomendable concentrar los elementos que se relacionan entre sí, haciendo que el comportamiento local del programa no dependa en exceso de elementos distantes en la implementación. Cuantas menos cosas “tengamos entre manos” y menos relaciones con otras, más fácil será resolver el problema.

La estructura *inicialización/condición/incremento* es ideal para programar algoritmos que requieran de un bucle controlado por contador. Con un contador, nos referimos a alguna variable que cuente, es decir, que tiene un valor inicial, que va incrementándose (o decrementándose) y que termina cuando alcanza cierto valor final.

3.3.2 Ejecutando un cierto número de veces

Lógicamente, si el bucle “for” nos permite contar, será el más adecuado cada vez que deseemos realizar una operación un número determinado de veces. Basta con establecer un contador que empieza en cero, se incrementa de uno en uno, y sigue hasta que se ha ejecutado tantas veces como deseamos.

A veces, el contador del bucle no se usa en el cuerpo del bucle para realizar ninguna operación con él, simplemente se utiliza para llevar la cuenta del número de iteraciones.

Ejercicio 3.12 — Media y varianza. Escriba un programa que lea un número entero que indica el número de datos y a continuación tantos números reales como indique dicho entero. Como resultado, escribirá la media y la varianza de dichos datos. Tenga en cuenta que la media y varianza de n datos x_i se pueden expresar como:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad Var(x) = \frac{\sum_{i=1}^n x_i^2}{n} - \bar{x}^2$$

Por ejemplo, para los 3 datos 4, 6, 8, la media y varianza son:

$$\bar{x} = \frac{4+6+8}{3} = 6 \quad Var(x) = \frac{4^2 + 6^2 + 8^2}{3} - 6^2 = 2,66666$$

En C/C++ es habitual contar desde el valor cero. Por ejemplo, si queremos realizar un programa que escriba 10 veces un mensaje, podríamos inicializar el contador en 1 y avanzar mientras sea menor o igual que 10. Sin embargo, lo habitual es inicializarlo en cero y avanzar mientras sea menor estricto que 10. En ambos casos se itera exactamente 10 veces. La razón para que el conteo se realice con estos límites se deba a que en C y C++ los índices de las secuencias enumeradas de elementos suelen comenzar en cero¹.

3.3.3 Omitiendo expresiones

En el bucle **for** podemos omitir cualquiera de las tres expresiones que lo definen. Así, podríamos eliminar la inicialización e incremento, dejando un bucle que únicamente expresa una condición del tipo “mientras”. Observe que, en este caso, tenemos una situación casi idéntica a la del bucle **while**, pero con otra sintaxis. Lógicamente, esta forma será poco recomendable, ya que hace el código menos legible y más confuso, optando por el uso de un bucle **while** tradicional.

Un caso extremo de este caso es cuando omitimos las tres expresiones, es decir, escribimos un bucle de la forma **for** (**;** **;**). Este tipo de bucle es lícito y se comporta como un bucle infinito, es decir, evalúa la expresión como **true** de forma indefinida y, por consiguiente, ciclará indefinidamente. Para este caso, es más habitual usar un bucle con **while (true)**, que deja muy clara la intención de hacer un bucle infinito sin necesidad de forzar el uso de ciclo **for**, que ha sido diseñado para otros casos.

En cualquier caso, nosotros implementaremos algoritmos que tienen fin y, por tanto, evitaremos estos algoritmos que ciclan indefinidamente².

3.4 Anidamiento de bucles

Al igual que ocurre con otras estructuras de control los bucles también se pueden anidar, es decir, incluirse en el cuerpo de otros bucles.

Ejercicio 3.13 — Mostrar cuadrado. Escriba un programa que lea un número entero N desde la entrada estándar y que a continuación dibuje con asteriscos –en la salida estándar– un cuadrado de lado N. A continuación vemos un ejemplo de ejecución:

```
Introduzca el lado del cuadrado: 6
*****
```

¹Un dato curioso con respecto a esta forma de enumerar puede encontrarlo si consulta las cuatro libertades esenciales del software libre. Verá que se enumeran como las libertades 0, 1, 2 y 3, lo que es una clara manifestación de la naturaleza de programadores que tenían los que las escribieron.

²Realmente, cuando encuentre estos bucles verá que no es que ciclen sin fin, sino que la forma de ruptura del flujo de control se hace de otra forma.

Ejercicio 3.14 — Media de medias. Escriba un programa que lea una serie de temperaturas de diferentes ciudades con el siguiente formato:

```
N
T11 T12 ... -100
T21 T22 ... -100
T31 T32 ... -100
...
TN1 TN2 ... -100
```

En donde N es el número de ciudades. Cada fila es la secuencia de temperaturas de una ciudad. Cada fila puede tener un número diferente de medidas y siempre acabará con un valor especial menor o igual a 100 (centinela).

El programa deberá calcular la temperatura media de cada ciudad y mostrará la máxima de las temperaturas medias así como el número de ciudad a la que corresponde.

A continuación puede ver un ejemplo de ejecución:

```
Consola
4
13 15 15.6 14 -100
8 12 -100
24.5 27 23.2 24 25 -100
23.7 27.4 29.1 -100
Mayor temperatura media: 26.73333
Ciudad: 4
```

Observe que en este ejemplo hay 4 ciudades. A continuación se introducen 4 secuencias de datos (una por cada ciudad). Cada secuencia consta de un número arbitrario de mediciones acabadas siempre con un valor menor o igual a -100. En concreto las medias para cada ciudad son: 14.4, 10, 24.74 y 26.73 y por tanto, la media mayor es 26.73 que se corresponde con la ciudad número 4.

3.5 Variables lógicas y condiciones compuestas

Un esquema que puede encontrar en distintas soluciones corresponde a un bucle que itera sobre un conjunto de posibles casos y que, en cierta iteración, localiza un caso que hace innecesario seguir iterando.

Por ejemplo, cuando estamos determinando si un número es primo, podemos buscar un divisor iterando sobre los números enteros que, comenzando desde el 2, llegan hasta el número en cuestión. Lógicamente, el esquema es claramente de contador, por lo que se puede resolver con un bucle **for**. Sin embargo, al encontrar un divisor, podríamos dar por terminado el algoritmo, pues ya sabemos que la respuesta a nuestro problema es que no es primo.

En los casos en los que el problema tiene que dar por terminado el ciclo al haber encontrado la solución buscada, podemos añadir una variable booleana para resolver la finalización. Por ejemplo, imagine que tenemos una variable *fin* que indica que ya hemos resuelto el problema. Inicialmente, *fin* puede valer **false** de forma que si localizamos la solución –en el cuerpo del bucle– hacemos que *fin* valga **true**. Para que el bucle termine en ese momento, será necesario modificar la condición añadiendo esta variable junto con el operador **&&**. Por ejemplo, estableciendo una expresión lógica del tipo: (*condición* **&&** !*fin*). Así, generamos una condición compuesta con una primera parte que corresponde a la condición natural del algoritmo y una segunda que corresponde a una variable booleana para una salida del bucle si concurren ciertas circunstancias especiales.

Ejercicio 3.15 — Números en rango. Escriba un programa que lea un entero y dos números reales: n, min y max. Estos valores indican que hay n números de entrada y que deberían estar en el intervalo [min,max]. A continuación, deberá leer los n valores de entrada y confirmar que todos están en el intervalo indicado. En caso de que uno de los valores no lo esté, parará la lectura y terminará el programa indicando que hay valores fuera de rango. Resuelva el ejercicio usando un ciclo **for**. A continuación se muestran dos posibles ejecuciones de este programa:

```
Consola
N, min, max: 3 0.0 5.0
Introduzca valor 0: 2.3
Introduzca valor 1: 2
Introduzca valor 2: 5.0
Los 3 valores introducidos están en el intervalo [0.0,5.0]
```

```
N, min, max: 3 0.0 10.0
Introduzca valor 0: 12
Error. Finalización inesperada del programa: Valor fuera de rango.
```

3.6 Casos generales y casos límite

Cuando se diseña un algoritmo iterativo, normalmente pensamos en un caso general, es decir, en lo que tiene que realizar el algoritmo cuando tenemos un caso habitual. Por ejemplo, si implementamos un algoritmo para calcular una media de una serie de valores positivos hasta introducir el cero; lo normal es que pensemos que vamos a tener una serie de valores acumulados en una variable, y que cada iteración tiene que obtener el valor de entrada correspondiente y acumularlo en dicha variable para, al final, escribir como resultado la suma dividida por el número de datos que se han introducido.

Esta solución es un caso típico que sirve para cuando leemos n valores, ya sean 2, 5, ó 10.000. Sin embargo, no hemos pensado en un caso especial: que n valga cero. Esto podría ocurrir si introducimos como primer dato de entrada el cero. En este caso, intentaríamos resolver la media como una división de cero entre cero que, como sabemos, es una indeterminación.

Como puede notar, el problema de este tipo de errores es consecuencia de que tendemos a resolver problemas pensando en los casos generales, por lo que los errores tienden a aparecer para los casos especiales. Así, cuando implemente una solución, recuerde que también tiene que ser válida para estos casos, por lo que es necesario revisar el código, ya sea para confirmar que ha sido correctamente resuelto, o para modificarlo en caso de que falle.

Ejercicio 3.16 — Cotización de las acciones. Se desea estudiar cómo progresó la cotización de una acción a lo largo del tiempo. Para ello, escribiremos un programa que lee la secuencia de valores de una acción. Esta secuencia describe el comportamiento de la acción durante un período de tiempo. El programa deberá escribir como resultado el valor máximo de la acción, el mínimo, la subida máxima, y la bajada máxima. Para determinar el final de la secuencia, se introducirá un valor negativo. Un ejemplo de ejecución es:

```
Introduzca un valor: 1.6
Introduzca un valor: 1.9
Introduzca un valor: 1.4
Introduzca un valor: 1.5
Introduzca un valor: -1

Resultados a partir de los tres valores introducidos:
    Valor máximo: 1.9
    Valor mínimo: 1.4
    Subida máxima: 0.3
    Bajada máxima: 0.5
```

Para finalizar el problema, vuelva a ejecutar el programa para comprobar el resultado con las siguientes secuencias:

- Secuencia (sin bajadas): 1 2 3 4 5 -1
- Secuencia (sin subidas): 5 4 3 2 1 -1
- Secuencia (vacía): -1

3.7 Algunos errores comunes

En esta sección vamos a presentar algunos errores que se suelen cometer cuando se programan bucles. El objetivo es que el lector los revise, intentando entender exactamente lo que hacen y descubriendo cuál es el error que se ha cometido. En caso de que no lo encuentre, compile y ejecute el código y vea su comportamiento para intentar deducir si hay un error.

Ejercicio 3.17 — Error en divisores de un número.

```
int numero;
cout << "Introduzca número positivo: ";
cin >> numero;

cout << "Los divisores son: ";
int divisor=1;
while (divisor<numero)
    if (numero%divisor==0)
```

```

        cout << "Divisor: " << divisor << endl;
else
    divisor++;

```

Ejercicio 3.18 — Error en listado de los pares menores.

```

int numero;
cout << "Introduzca número positivo: ";
cin >> numero;

cout << "Los pares menores son: ";
for (int i=2; i!=numero; i+=2)
    cout << i << " ";

```

Ejercicio 3.19 — Error en cuenta regresiva.

```

unsigned int i;
cout << ";Cuenta regresiva!" << endl;
for (i=10; i>=0 ; i--)
    cout << "..." << i;

```

Ejercicio 3.20 — Error en media de 10 números.

```

double suma=0, dato;
for (int i=0; i<10; i--) {
    cout << "Introduce dato: ";
    cin >> dato;
    suma += dato;
}
cout << "Media: " << suma/10 << endl;

```

Ejercicio 3.21 — Error en tablas de multiplicar.

```

for (int i=10; i>0; --i) {
    cout << "Tabla del " << i << ":" << endl;
    for (int j=1; j<=10; ++j);
        cout << i << " x " << j << " = " << i*j << endl;
}

```

3.7.1 Funciona, pero mejor no hacerlo

También podemos ver algunos ejemplos en los que el algoritmo funciona correctamente para todos los casos, pero en los que se podría modificar el código para una mejor solución.

Ejercicio 3.22 — Si es primo ... mejorable. En este código se determina si un número es primo. Se propone que la búsqueda de un divisor se haga hasta la raíz cuadrada del número, ya que si no hay divisores hasta la raíz, no hay divisores mayores. ¿Cómo podría modificar el programa para que fuera aún más rápido?

```

int numero;
cout << "Introduzca número positivo: ";
cin >> numero;
bool es_primo=true;
for (int i=2; i<sqrt(numero) && es_primo; ++i)
    if (numero%i==0) es_primo= false;

```

Ejercicio 3.23 — Buscar primer divisor ... mala solución. En el siguiente ejemplo, determinamos el primer divisor de un número y finalizamos el bucle modificando el valor de la variable contador.

```

int numero;
cout << "Introduzca número positivo: ";
cin >> numero;
int divisor=0;
for (int i=2; i<numero; ++i)
    if (numero%i==0) {
        divisor=i;

```

```

        i=numero;
    }
if (divisor!=0)
    cout << "Localizado: " << divisor << endl;

```

Aunque la solución funciona, deberíamos evitar modificar la variable del contador dentro del cuerpo del bucle, ya que es importante que el código sea legible y fácil de modificar. Cuando vemos la cabecera de este bucle **for**, lo que parece hacer es recorrer todos los elementos menores que el número. Es necesario revisar el cuerpo para descubrir que el comportamiento es otro. Por tanto, hemos perdido la legibilidad que lo justifica frente al bucle **while**. A pesar de que la solución sea más costosa en tiempo, en principio es preferible implementar código más legible.

Corrija el código para evitar que se modifique el contador dentro del bucle. Puede optar por seguir usando un bucle de tipo **for** (con alguna modificación) o bien cambiarlo por un bucle de tipo **while**.

Finalmente, es interesante indicar que podríamos hacer que un bucle terminara directamente usando la instrucción **break**. Sin embargo, evitaremos estas soluciones, ya que rompen el paradigma de la programación estructurada y nos pueden llevar a implementaciones menos legibles, más confusas y por lo tanto, más propensas a errores y difíciles de mantener. La instrucción **break** se limitará a su uso en la instrucción **switch** como hemos visto anteriormente. De la misma forma, tampoco usaremos instrucciones como **continue** y **goto**, a pesar de que en algunos ejemplos podrían generar códigos con mejores tiempos de ejecución.

3.8 Ejercicios adicionales

Ejercicio 3.24 — Calculadora ampliada. Considere el ejercicio de la minicalculadora (ejercicio 2.10). Escriba un programa con el mismo menú, añadiendo tres opciones más:

- Modificar primer operando.
- Modificar segundo operando.
- Fin.

Para ello, tenga en cuenta que el valor inicial de los operandos es cero, y que la solución debe contener un bucle **do-while** junto con una instrucción **switch** para gestionar el menú.

Ejercicio 3.25 — Número perfecto. Un número perfecto es aquel que es igual a la suma de todos sus divisores positivos excepto él mismo. El primer número perfecto es el 6, ya que sus divisores son 1, 2 y 3 y $6=1+2+3$. Escriba un programa que muestre el mayor número perfecto que sea menor a un número dado.

Ejercicio 3.26 — Número narcisista. Un número entero de n dígitos se dice que es narcisista si se puede obtener como la suma de las potencias n -ésimas de cada uno de sus dígitos. Por ejemplo $153 = 1^3 + 5^3 + 3^3$ y $8208 = 8^4 + 2^4 + 0^4 + 8^4$ son números narcisistas. Escriba un programa que lea un número entero positivo y nos diga si es narcisista o no.

Ejercicio 3.27 — Mostrar figuras. Escriba un programa que lea un número entero N desde la entrada estándar y que a continuación dibuje con asteriscos –en la salida estándar– las siguientes figuras:

1. Triángulo de lado N
2. Cuadrado hueco de lado N
3. Rombo de altura N y de ancho $N/2$. En este caso N debe ser un número impar.

A continuación vemos una muestra de estas figuras para $N=7$:

```

*           ****       *
**         *   *     ***
***       *   *   ****
****     *   *   *****
*****   *   *   ****
*****   *   *     ***
*****   *   *       *

```

Ejercicio 3.28 — Run length encoding. El método RLE (Run Length Encoding) de codificación permite almacenar en poco espacio largas secuencias de datos. El algoritmo consiste en transformar las secuencias de valores idénticos consecutivos en parejas compuestas por el valor y el número de repeticiones.

Escriba un programa que lea una secuencia de números terminada con un número negativo y la codifique mediante el método RLE. Un ejemplo de ejecución sería el siguiente:

Consola

Introduzca secuencia: 1 1 1 2 2 2 2 2 3 3 3 3 3 3 5 -1
RLE: 3 1 5 2 6 3 1 5

donde la salida corresponde a indicar que hay 3 unos, 5 dos, 6 tres y 1 cinco.

Ejercicio 3.29 — Número feliz. Se dice que un número natural es feliz si cumple que al sumar los cuadrados de sus dígitos y repetimos esta suma con los resultados que vamos obteniendo, finalmente obtenemos el dígito 1 como resultado. Por ejemplo, el número 203 es un número feliz ya que: $2^2 + 0^2 + 3^2 = 13 \rightarrow 1^2 + 3^2 = 10 \rightarrow 1^2 + 0^2 = 1$

Además, un número es feliz de grado k si es feliz en un máximo de k iteraciones. En el ejemplo anterior, 203 es un número feliz de grado 3 (además, es feliz de cualquier grado mayor o igual que 3). Escriba un programa que lea un número n y un grado k y diga si n es feliz grado k.

Ejercicio 3.30 — Secuencia más larga. Escriba un programa que lea una secuencia de enteros positivos terminada en un valor negativo, y que escriba en la salida estándar la longitud de la subsecuencia de números ordenada de mayor longitud junto con la posición donde comienza. Tenga en cuenta que dos números están ordenados si el primero es menor o igual que el segundo. Un ejemplo de ejecución es:

Consola

Secuencia (termina con negativo): 5 9 1 4 8 8 10 5 14 -1
Máxima longitud: 5 comenzando en 2

Observe que el elemento con índice 2 –el número 1– es el comienzo de una secuencia de 5 números ordenados. Para finalizar el problema, vuelva a ejecutarlo para comprobar el resultado de las siguientes secuencias:

- Secuencia: -1
- Secuencia: 1 2 3 4 5 -1

Ejercicio 3.31 — Hipoteca. El valor de la cuota de una hipoteca, dado el capital, el interés y un plazo de devolución se puede calcular como:

$$\text{Cuota} = \frac{\text{Capital} \cdot \text{Interes}}{100 \cdot \left(1 - \left(1 + \frac{\text{Interes}}{100}\right)^{-\text{Plazo}}\right)}$$

donde, como consideramos cuotas mensuales, el interés es el mensual –interés anual entre 12– y el plazo es el número de meses que dura la hipoteca.

Como sabemos, las cuotas son idénticas todos los meses (supongamos que es a plazo fijo), pero realmente en las primeras se dedica un mayor porcentaje a intereses. Concretamente, para calcular la parte de intereses y de capital que se paga (amortización), tenemos que calcular:

$$\text{Intereses} = \text{Pendiente} \cdot \frac{\text{Interes}}{100} \quad \text{Amortizacion} = \text{Cuota} - \text{Intereses}$$

donde “interés” de nuevo es el interés mensual, y “pendiente” es el capital que aún queda por pagar. Inicialmente será el total del préstamo, y cada mes disminuirá en tanto como sea la amortización.

Escriba un programa para, a partir de los valores que definen un préstamo (capital, interés anual y el plazo en años) conocer cómo se irá pagando. Para ello, tenga en cuenta que debe escribir todas las tripletas de valores (pendiente, intereses, amortización) que corresponden a cada mes. Además, para cada año –es decir, tras doce tripletas– se dará un mensaje con el total de interés y amortización que se ha pagado durante ese año.

4

Vectores y matrices

Introducción.....	29
El tipo vector	29
Operaciones con vectores	
Vectores de tamaño variable	
Vectores y lectura adelantada	
Vectores de vectores	32
Matrices	
Ordenación y búsqueda	33
Ejercicios adicionales.....	34

4.1 Introducción

Para resolver problemas muy sencillos es suficiente disponer de objetos que almacenen un dato simple, como puede ser un entero o un carácter. Sin embargo, los problemas más complejos requieren el manejo de una gran cantidad de información, miles o millones de datos, que no se pueden manejar con este tipo de objetos. En este guión vamos a practicar con una forma de declarar un simple nombre para poder manejar múltiples datos: los vectores.

El lenguaje C ya nos ofrece la posibilidad de manejar vectores. Ese código también es válido en C++, y por tanto, podemos usar el mismo tipo de dato para resolver el problema en este lenguaje. Sin embargo, C++ nos ofrece una nueva alternativa, el tipo **vector** de la STL (Standard Template Library), un tipo de dato más elaborado y que nos ofrece nuevas operaciones, haciendo más simples y robustos los algoritmos que se resuelven con él.

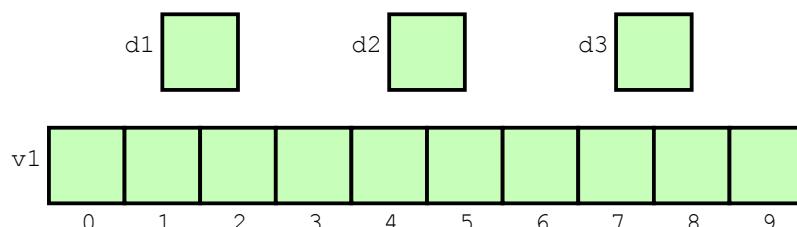
El objetivo de este guón es que el alumno practique con problemas para los que se necesitan vectores y para los que se puede usar el tipo vector de la STL como una solución cómoda y eficaz para resolverlos. No se va a realizar un estudio en profundidad, mostrando únicamente las herramientas básicas para que un alumno que empieza a programar pueda usar este tipo de dato.

4.2 El tipo vector

El tipo **vector** no es un tipo definido en el lenguaje, sino que está elaborado mediante una clase C++ que, al igual que otras herramientas, está dentro de **std**. Para disponer de este tipo, será necesario incluir el fichero de cabecera **vector**. Un objeto de este tipo es:

- **Compuesto.** El tipo puede contener múltiples objetos. Podemos decir que es un contenedor de objetos. Por ejemplo, un vector podrá contener 10 objetos enteros (de tipo **int**).
- **Homogéneo.** Todos los objetos deben ser del mismo tipo. Es decir, es un contenedor de objetos de un tipo concreto. Por tanto, no podemos tener un vector que contenga caracteres (**char**) y números reales (**double**).

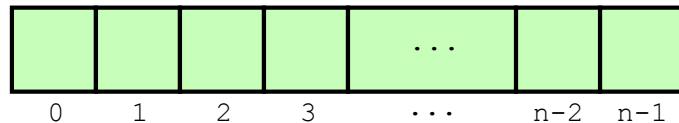
Para poder manejar cada uno de los elementos contenidos, el vector los organiza mediante un índice. Es decir, podemos distinguir dos objetos contenidos indicando dos índices distintos. En la siguiente figura se muestra un esquema con 4 objetos: 3 objetos simples y un vector.



Es interesante que observe que hemos especificado 4 nombres de objetos: **d1**, **d2**, **d3**, y **v1**; objetos que corresponderían a las variables que declaramos en nuestro programa. Con las 3 primeras referencias, accedemos a tres objetos simples, mientras que con el nombre **v1** junto con un índice, podemos acceder a 10 objetos distintos.

En el ejemplo hemos indicado que el vector tiene un tamaño 10. En la práctica, un vector tendrá un tamaño no negativo, incluyendo el tamaño cero. Este tamaño, además, podrá modificarse durante la ejecución del programa, haciendo crecer

o decrecer la capacidad del contenedor conforme se necesite para resolver el problema. En general, un vector se puede representar con el siguiente esquema:



donde hemos indicado un tamaño variable n , ya que puede variar durante la ejecución.

El índice para acceder a cualquier elemento de un vector debe ser un valor que va desde el cero al tamaño menos uno. En C/C++ todos los índices se manejarán comenzando desde el cero¹.

Por otro lado, observe que no se ha especificado ningún tipo de dato concreto, es decir, no hemos dicho que las variables contengan, por ejemplo, caracteres o enteros. De la misma forma que podemos tener una variable de tipo **char** y otra de tipo **int**, podemos tener un vector que contiene **char** o contiene **int**. Es decir, realmente el tipo **vector** no es un tipo de dato, sino que es un tipo de contenedor que da lugar a múltiples tipos de datos: vectores de enteros, vectores de caracteres, vectores de booleanos, etc.

4.2.1 Operaciones con vectores

En esta sección vamos a presentar las operaciones más sencillas y suficientes para resolver de una forma simple y eficiente la mayor parte de los problemas con vectores. En primer lugar, y para poder trabajar con ellos, será necesario crear objetos de tipo **vector**.

Para declarar una variable será necesario especificar el tipo contenido. Opcionalmente, se puede especificar el tamaño y el valor con el que inicializar todos los objetos contenidos. En concreto, la declaración se puede hacer como sigue:

```
vector<T> nombre;           // Vector de tipo T. Tamaño cero.
vector<T> nombre(n);     // Vector de tipo T. Tamaño n.
vector<T> nombre(n, v); // Vector de tipo T. Tamaño n. Inicializado a v.
```

donde podemos ver que el primer caso declara un **vector** que tiene capacidad cero y los dos siguientes con capacidad para n objetos. En el tercer caso, además hacemos que cada objeto contenido tenga un valor inicial *v*. Ejemplos concretos que pueden aparecer en nuestros programas son los siguientes:

```
vector<int> v1;           // Vector de int. Tamaño cero.
vector<double> v2;        // Vector de double. Tamaño cero.
vector<bool> v3(10);      // Vector de bool. Tamaño 10.
vector<int> v4(10,1);    // Vector de int. Tamaño 10. Inicializados a 1.
```

Como el tipo **vector** es un dato compuesto genérico, no están definidas las operaciones de comparación ni las de entrada y salida. Es decir, no hay ninguna definición que nos diga cuándo un vector es menor que otro, o la forma en que se podría hacer una lectura desde **cin** o una escritura a **cout**.

Por otro lado, sí está definida la asignación. Si un **vector** lo asignamos a otro, haremos que éste pase a tener el mismo tamaño y los mismos elementos que el primero (perdiendo los que tuviese si es que tenía alguno).

Tamaño del vector y acceso a sus elementos

Para modificar el contenido de un vector, primero debemos conocer el número de elementos que tiene y segundo, es necesario poder acceder a cada uno de los elementos que lo contienen. Para ello, disponemos de las siguientes operaciones:

- Tamaño del vector. La operación **size** –sin parámetros– aplicada sobre un **vector** nos devuelve el número de elementos que lo componen.
- Acceso a un objeto contenido. La operación **at** –con un parámetro que corresponde al índice en el **vector**– aplicada a un **vector** nos permite seleccionar uno de sus elementos.

La mejor forma de entender la sintaxis y su uso es mostrar un ejemplo:

```
#include <iostream>
#include <vector> // Para poder usar vectores
using namespace std;

int main()
{
    vector<int> v(10,1);

    // Mostramos el tamaño del vector
    cout << "El tamaño es: " << v.size() << endl;

    // Asignamos valores a las celdas del vector
    for (int i=0; i<v.size(); ++i)
        v.at(i) = i;
}
```

Algunos detalles que es interesante que observe son:

¹Seguramente se habrá dado cuenta que cuando queremos iterar en un bucle de tipo **for** un número de veces, por ejemplo 10, iteramos desde el cero, es decir, desde el 0 al 9. Un programador C/C++ está habituado a empezar en cero.

- Hemos incluido el archivo cabecera `vector` para poder usar este tipo.
- Para acceder a una operación del tipo añadimos un punto al nombre del vector junto con el nombre de la operación a aplicar (`at` o `size`).
- La operación `size` es de consulta, nos devuelve el tamaño del vector (en este caso 10).
- Cuando usamos la operación `at`, obtenemos una referencia a un objeto contenido. Por tanto, con esta operación, podemos consultar o modificar el elemento en esa posición. En el ejemplo anterior lo hemos usado dentro del bucle para modificar el contenido de todas sus posiciones.

Ejercicio 4.1 — Mostrar elementos de un vector. Complete el programa del ejemplo anterior añadiendo al final un bucle que muestre en la salida estándar todos los elementos del vector.

Es muy importante tener en cuenta que la operación `at` no se puede aplicar con un índice que esté fuera del rango válido. Por ejemplo, si un vector tiene 10 elementos, no podemos acceder al elemento 20.

Ejercicio 4.2 — Acceso a los elementos. Considere el programa del ejemplo anterior y añada las siguientes líneas para ver tres elementos concretos:

```
cout << "Elemento en 0: " << v.at(0) << endl;
cout << "Elemento en 1: " << v.at(1) << endl;
cout << "Elemento en 10: " << v.at(10) << endl;
cout << "Elemento en 10000000: " << v.at(10000000) << endl;
cout << "Elemento en size-1: " << v.at(v.size()-1) << endl;
```

Antes de ejecutar este código piense qué valores son los que se mostrarían. A continuación ejecútelo y verifique sus suposiciones.

Además de la operación `at`, podemos usar los corchetes para poder referirnos a un elemento de un vector. En lugar de escribir `v.at(i)`, usamos la sintaxis `v[i]`. La diferencia fundamental está en que:

- En la operación `at` se comprueba que el índice está en el rango correcto. Si no lo está, directamente obtenemos un resultado como el que ha visto en el ejercicio 4.2.
- En la operación `[]` no se hacen comprobaciones. Por tanto, se intenta realizar la operación aunque estemos en un índice incorrecto. Esto hace que sea muy peligroso, pues accedemos a posiciones de memoria incorrectas, aunque permite que el código no pierda tiempo en las comprobaciones si estamos seguros de que son correctas.

En la práctica, la sintaxis de corchetes es más simple y más eficiente, y además, es la misma que se usa para los vectores en C, por lo que es la más habitual. Por supuesto, su uso obliga a tener mucho cuidado, pues tenemos que garantizar que no nos salimos del rango válido.

Ejercicio 4.3 — Acceso con corchetes. Considere el ejercicio 4.2 y cambie el operador `at()` por el operador `[]`. Es decir, añada estas líneas:

```
cout << "Elemento en 0: " << v[0] << endl;
cout << "Elemento en 1: " << v[1] << endl;
cout << "Elemento en 10: " << v[10] << endl;
cout << "Elemento en 10000000: " << v[10000000] << endl;
cout << "Elemento en size-1: " << v[v.size()-1] << endl;
```

¿Qué diferencias observa en la ejecución con respecto a la versión que usa `at()`?

Ejercicio 4.4 — Media, desviación y varianza. Escriba un programa que lea un número entero que indica el número de datos a analizar y a continuación tantos números reales como indique dicho entero. Como resultado, escribirá la media, la desviación media y la varianza de dichos datos. Tenga en cuenta que la media, la desviación media y la varianza de n datos x_i se pueden expresar como:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad d_m = \frac{1}{n} \sum_{i=1}^n |x_i - \bar{x}| \quad Var(x) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

Para ello, pida un número entero positivo, declare un vector con dicho tamaño, y luego lea todos los elementos en el vector antes de realizar los cálculos.

Ejercicio 4.5 — criba de Eratóstenes. Escriba un programa que calcule todos los números primos menores que un número N dado (este número será leído desde la entrada estándar). Para ello, use la criba de Eratóstenes^a. La idea de este algoritmo consiste en escribir todos los números hasta el n , y recorrerlos de uno en uno. Si un número no se ha tachado, se tacharán todos sus múltiplos. Cuando hemos recorrido todos los números, aquellos no tachados son los números primos.

Para resolverlo puede optar por dos opciones:

1. Declarar un vector de booleanos de tamaño N . En dicho vector, el valor de una posición i (`true` o `false`) indicará si el valor i de la serie de números está o no tachado.

2. En lugar de un vector de booleanos se puede crear un vector de enteros inicializado con los números del 2 en adelante. En dicho vector se tacharán los números cambiándolos por un valor especial (por ejemplo -1).

Como resultado, el programa escribirá todos los primos resultantes.

^ahttp://es.wikipedia.org/wiki/Criba_de_Erat%C3%B3stenes

Ejercicio 4.6 — Criba de Eratóstenes 2. Modifique el ejercicio 4.5 para que, como resultado del algoritmo de la criba, tengamos un vector sólo con los números primos y sin los números tachados.

4.2.2 Vectores de tamaño variable

En la sección anterior hemos considerado un vector de un tamaño fijo desde la declaración. En la práctica, puede resultar muy cómodo poder modificar la capacidad del vector en tiempo de ejecución. Podemos decir, por tanto, que trabajamos con vectores de tamaño dinámico.

Aunque hay más posibilidades, nosotros consideraremos las siguientes operaciones:

- Añadir un elemento al final del vector. La operación es *push_back*, con un parámetro que corresponde al elemento que queremos añadir. El resultado es que el tamaño del vector aumenta en uno, al contener ese nuevo elemento en la posición *size() - 1*.
- Eliminar el elemento del final. La operación es *pop_back*, sin parámetros. El resultado es que el tamaño del vector disminuye en uno. Se pierde el último dato. Lógicamente, sólo tiene sentido si el vector tiene, al menos, un elemento.
- Vaciar un vector. La operación es *clear*, sin parámetros. El resultado es que el tamaño del vector pasa a ser cero, es decir, está vacío.

Ejercicio 4.7 — Media, desviación y varianza de números positivos. Escriba un programa similar al del ejercicio 4.4, pero teniendo en cuenta que los valores que se analizarán serán no negativos. La entrada de datos se realizará leyendo valores hasta que se lea un valor negativo. En ese momento, se obtendrán en la salida estándar los valores de la media, desviación y varianza de los datos introducidos.

Ejercicio 4.8 — Eliminar pares. Escriba un programa que lea números positivos y los almacene en un vector (la lectura será similar a la que ha hecho en el ejercicio 4.7). A continuación, el programa eliminará del vector todos los números que sean pares. Para ello, deberá recorrer el vector y cada vez que encuentre un número par, lo intercambiará con el último y a continuación lo eliminará con la operación *pop_back*. Observe que el vector resultante no mantiene el orden original de los elementos impares.

4.2.3 Vectores y lectura adelantada

En la sección 3.2.2 se estudió un esquema de procesamiento de datos denominado “*lectura adelantada*” que se usaba cuando había que procesar series de datos leídas desde la entrada estándar hasta que se cumpliese una determinada condición. Este esquema también se usa con cierta frecuencia al tratar datos almacenados en vectores. La idea es la misma con la diferencia de que ahora los datos se van cogiendo del vector en lugar de cogerlos desde la entrada estándar.

Ejercicio 4.9 — Media, desviación y varianza de números positivos con lectura adelantada. Revise el ejercicio 4.7 y asegúrese de que en la lectura de datos está utilizando un esquema de lectura adelantada. Con este esquema únicamente debería hacer inserciones con *push_back* y no debería hacer ningún borrado con *pop_back*.

Ejercicio 4.10 — Parejas de producto par. Haga un programa que lea una serie de números enteros –positivos y negativos– desde la entrada estándar hasta que el usuario introduzca un valor cero. El número total de números introducidos ha de ser par y al menos debería haber dos números. Si cuando el usuario introduce un cero el número total de números fuese impar, se descartaría el cero y se seguirían leyendo datos. La serie de datos se almacenará en un vector que incluirá como último dato el valor cero.

A continuación deberá recorrer el vector e indicar si existe al menos una pareja de números enteros consecutivos cuyo producto sea par. Observe que este bucle deberá detenerse en el momento en que encuentre una pareja que cumpla la condición o bien cuando se llegue al final del vector. Para ello se recomienda el uso de un bucle de tipo **while**. El cero no se considera para los cálculos. Las parejas que deben considerarse son las de las posiciones 0-1, 1-2, 2-3, ...

4.3 Vectores de vectores

Un vector es un contenedor de objetos que pueden ser también vectores. Para declarar un vector de vectores, simplemente tendremos que indicar el tipo **vector** dentro de los caracteres <> que encierran el tipo de objetos contenidos.

Como el tipo contenido será un vector, este tipo también contendrá un par de caracteres <> que encierran otro tipo. Sólo tendremos que tener cuidado con la sintaxis, ya que si el compilador encuentra los caracteres >> consecutivos, interpretará incorrectamente su sentido. Para poder escribirlos, tendremos que hacerlo separándolos por un espacio.

El uso de estos vectores no debería ser problemático, ya que se usa como cualquier otro vector. Sólo tenemos que tener en cuenta que cuando referenciamos un objeto contenido, es también un vector, y necesitará manejarse como tal. En el siguiente trozo de código puede comprobar esta idea:

```
vector<int> v;
vector<vector<int> > vdev; // Los >> separados para que funcione

vdev.push_back(v);
cout << vdev.size() << endl;      // Escribe 1
cout << vdev.at(0).size() << endl; // Escribe 0
vdev.at(0).push_back(5);           // Añade 5 al primer vector
vdev.push_back(vdev.at(0));        // Añade un vector igual al 0
vdev.push_back(vdev[0]);          // Lo mismo, pero con []
cout << vdev[2][0] << endl;       // Escribe 5
```

Ejercicio 4.11 — Control de ventas. Una empresa de ventas recibe una secuencia de enteros para controlar el número de ventas de cada uno de sus vendedores. Las ventas de un vendedor consisten en una secuencia de números enteros no negativos terminada con un valor -1 que indica final de secuencia. El conjunto de ventas total se compone de una secuencia de ventas de vendedores terminada con un número -2 para indicar final de vendedores. Por ejemplo, en la siguiente secuencia se muestran las ventas de 4 vendedores:

3 5 0 2 -1 4 11 2 -1 7 1 -1 8 9 5 10 2 -1 -2

donde vemos que han vendido 10, 17, 8 y 34 unidades cada uno.

Escriba un programa que lea una secuencia como la anterior, calcule el número de ventas de cada vendedor, y escriba el número de vendedor con mayores ventas así como su secuencia correspondiente.

Observe que en este ejercicio hay que implementar dos esquemas de lectura adelantada anidados. Se necesita hacer una lectura adelantada de “secuencias” acabadas con el valor especial -2. Cada una de estas “secuencias” es, a su vez, un esquema de lectura anticipada de series de datos acabadas en el valor especial -1.

4.3.1 Matrices

Un caso particular, especialmente interesante, de vectores de vectores es cuando todos los vectores contenidos tienen exactamente el mismo tamaño. Podemos decir que la estructura de datos que obtenemos es rectangular, es decir, tenemos un conjunto de $f \times c$ elementos, ya que tenemos un vector de f vectores que contienen a su vez c elementos.

El tipo vector se ha creado como una estructura 1-dimensional de objetos contenidos, aunque se puede usar para representar estas estructuras 2-dimensionales. Por ejemplo, en el siguiente trozo de código declaramos dos matrices de dos formas distintas:

```
// Matriz de 5x7 usando la variable v
vector<double> v(7,0.0); // 7 posiciones con valor cero
vector<vector<double> > matriz5x7(5,v); //Cada posición un vector de 7

// Matriz de 10x20 en una sola línea
vector<vector<int> > matriz10x20 ( 10 , vector<int>(20,0) );
```

Observe que en ambos casos las matrices están inicializadas con ceros. El tamaño de una matriz se puede obtener con `size` sobre dicha matriz (número de filas), y con `size` sobre cualquiera de los objetos contenidos (número de columnas).

Ejercicio 4.12 — Traza de una matriz. Implemente un programa que lea una matriz cuadrada desde la entrada estándar de tamaño arbitrario y que calcule y muestre su traza.

Ejercicio 4.13 — Suma de matrices. Implemente un programa que lea dos matrices rectangulares de igual tamaño y que calcule y muestre su suma (elemento a elemento).

Ejercicio 4.14 — Máximos de cada fila. Implemente un programa que lea los datos de una matriz de tamaño arbitrario (también leído desde la entrada estándar). El programa calculará y mostrará el elemento mayor de cada fila.

4.4 Ordenación y búsqueda

Los algoritmos de ordenación y búsqueda son fundamentales. En principio, es interesante practicar con ellos intentando no copiar anteriores implementaciones, sino reescribiéndolas sabiendo cómo funcionan.

Ejercicio 4.15 — Frecuencias de los datos. Escriba un programa para obtener la frecuencia de los datos de entrada. El programa recibe un entero que indica el número de datos, seguido de los datos. La salida será el conjunto de datos introducidos junto con su frecuencia. Un ejemplo de ejecución podría ser este:

Para resolverlo se recomienda:

- Ordenar los datos introducidos.
- A partir del vector de datos ordenados, obtener dos nuevos vectores, uno con los datos sin repetir y otro con las frecuencias.

Los algoritmos de ordenación son básicos para resolver otros problemas. Por ejemplo, se puede obtener la mediana de un conjunto de valores si ordenamos los datos y seleccionamos el valor central².

Ejercicio 4.16 — Mediana. Escriba un programa que reciba un entero indicando el número de datos que se van a introducir, seguido de tantos datos como indique dicho número, y obtenga como salida la mediana de los datos.

La mediana es el valor que deja por debajo a la mitad de los datos y por encima a la otra mitad. Si el número de datos es impar, la mediana corresponde al valor central, y si es par, se puede obtener como la media de los dos centrales.

Ejercicio 4.17 — Mezcla de vectores ordenados. Escriba un programa que recibe dos secuencias de elementos ordenadas y escribe el resultado como una secuencia mezcla de las dos anteriores. Cada secuencia se introduce como un entero que indica el número de elementos seguido de dichos elementos.

El programa debe asegurar que si cualquiera de las secuencias no está ordenada, se ordena antes de la mezcla.

El algoritmo de búsqueda binaria es especialmente potente gracias a la eficiencia cuando el número de elementos es muy alto. Así, si disponemos de un conjunto de datos ordenado, resulta especialmente recomendable que optemos por este algoritmo. Incluso si no estuvieran ordenados, pero vamos a realizar múltiples búsquedas, podría ser recomendable que dediquemos un coste a la ordenación para hacer que las siguientes búsquedas sean muy rápidas.

Ejercicio 4.18 — Frecuencias de los datos (2). Considere el programa del ejercicio 4.15, donde se calculan las frecuencias asociadas a una serie de datos. Se desea mejorar el programa para los casos en los que existen pocos datos y altas frecuencias. Para ello, se propone que no se ordenen los datos de entrada, sino que se creen los dos vectores solución (datos y frecuencias) y se vayan modificando con cada uno de los datos de entrada. El algoritmo, básicamente, consiste en coger un nuevo dato y, si ya se ha obtenido previamente, incrementar su frecuencia, y si no, insertarlo en la solución con frecuencia uno.

4.5 Ejercicios adicionales

Ejercicio 4.19 — Letras más usadas. Se desea realizar un programa para calcular la frecuencia de las letras en un documento. Para ello, se analizarán las apariciones de cada una de las letras de la parte básica de la tabla ASCII (los caracteres del cero al 127).

Escriba un programa que lea los caracteres de un texto hasta encontrar el carácter '#'. Para cada uno de los caracteres leídos, calculará la frecuencia de aparición. Como resultado, escribirá en la salida estándar cada uno de los pares carácter/frecuencia de mayor a menor frecuencia. Sólo se imprimirán los caracteres que hayan aparecido, al menos, una vez.

Ejercicio 4.20 — Descomposición en números primos. Escriba un programa que lea un número positivo y escriba su descomposición en números primos. El programa almacenará la descomposición en un vector.

Ejercicio 4.21 — Columnas únicas. Escriba un programa que lea una matriz bidimensional desde la entrada estándar y que indique cuántas de sus columnas son únicas. Una columna se considera única si no se repite, es decir, si no hay otra igual a ella.

²Hay algoritmos más eficientes para obtener la mediana sin necesidad de ordenar todos sus valores.

Ejercicio 4.22 — Traspuesta de una matriz. Implemente un programa que lea una matriz por la entrada estándar y que calcule y muestre su correspondiente traspuesta.

Ejercicio 4.23 — Punto de silla. Implemente un programa que lea los datos de una matriz de tamaño arbitrario (también leído desde la entrada estándar). El programa nos dirá si existe o no algún elemento que sea a la vez máximo de su fila y mínimo de su columna.

Ejercicio 4.24 — Transformaciones de una matriz. Implemente un programa que lea una matriz cuadrada y que, a elección del usuario, permita realizar:

- Una rotación de 90 grados a la derecha.
- Una rotación de 90 grados a la izquierda.
- Una rotación de 180 grados.
- Una simetría respecto al eje central vertical.
- Una simetría respecto al eje central horizontal.

5

Cadenas de caracteres



Introducción.....	37
El tipo string	37
Operaciones básicas con string	38
Más operaciones con string	41
Concatenación de cadenas	
Comparación de cadenas	
Extracción de subcadenas	
Borrado de subcadenas	
Inserción de subcadenas	
Reemplazo de subcadenas	
Búsqueda en cadenas	
Ejercicios adicionales.....	44

5.1 Introducción

El objetivo de este capítulo es que el alumno practique con problemas para los que se necesitan cadenas de caracteres y para los que se puede usar el tipo **string** como una solución cómoda y eficaz para resolverlo.

Los primeros problemas que se han presentado –para introducirse en la programación– contienen algoritmos de procesamiento de tipos simples, como son el tipo **bool**, **int** o **char**. Eso nos ha permitido crear algunas soluciones donde procesamos caracteres individuales. Sin embargo, es habitual que la información de texto aparezca como secuencias de caracteres, como por ejemplo el nombre de una persona, la dirección postal, el nombre de un producto, etc.

Para manejar tipos más complejos, el lenguaje nos ofrece la posibilidad de crear nuevos tipos de datos compuestos, como estructuras o clases (para unir varios objetos incluso de distinto tipo) o vectores (para componer varios objetos del mismo tipo). Observe que aunque los vectores sean un tipo de dato que nos permitiría gestionar secuencias de **char**, el hecho de que la información de texto sea tan importante, y con unas características tan especiales, hace que en los lenguajes se incluyan herramientas adicionales, especializadas en su procesamiento.

En el caso de C++, se ofrece un nuevo tipo de dato **string** para manejar cadenas de caracteres, adicional a las herramientas que ofrece C y que siguen disponibles. Así, en este lenguaje podemos procesar este tipo de información como:

- Una cadena de caracteres C. En este lenguaje las cadenas de caracteres se gestionan de una forma muy básica, realmente como un caso particular de vectores C con algunas características adicionales y una serie de operaciones para facilitar su manejo.
- Un **string**. Es un tipo elaborado mediante una nueva clase, que elimina los inconvenientes que tienen las cadenas C, creando una nueva interfaz más avanzada y con más posibilidades.

En este guión se introduce la clase **string** con las operaciones más importantes, de forma que el alumno pueda resolver la mayoría de los algoritmos que usen cadenas de caracteres. No se va a realizar un estudio en profundidad, mostrando únicamente las herramientas básicas para que un alumno que empieza a programar pueda usar este tipo de dato. De hecho, no será necesario ningún conocimiento sobre cadenas C para que pueda usarse a un nivel básico de programación. Si desea conocer más detalles sobre este tipo de dato, puede consultar [2] o [3] para una explicación completa, aunque su lectura se debería retrasar hasta que el lector tenga un conocimiento más profundo sobre cadenas C y clases.

5.2 El tipo **string**

El tipo **string** no es un tipo definido en el lenguaje, sino que está elaborado mediante una clase C++ que, al igual que otras herramientas, está dentro de **std**. Para disponer de este tipo, será necesario incluir el fichero de cabecera **string**¹.

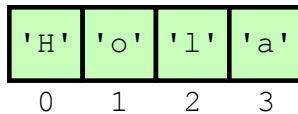
Una vez incluido este fichero de cabecera, tendremos disponible el nombre del tipo para poder declarar variables como con cualquier otro tipo básico del lenguaje, escribiendo el nombre del tipo seguido por una secuencia de una o más variables terminada en punto y coma.

Por otro lado, si queremos trabajar con valores concretos de este tipo, es decir, lo que hemos llamado literales para los tipos predefinidos del lenguaje, podemos hacerlo escribiendo el nombre del tipo con el valor concreto entre paréntesis. Por ejemplo:

```
string( "Hola" )
```

¹Note que existe otro fichero de cabecera, **cstring**, que contiene lo necesario para manejar las cadenas de caracteres C y que, por tanto, no usaremos en este capítulo.

es un **string** de tamaño 4, ya que la cadena que almacena contiene 4 caracteres ('H', 'o', 'l', 'a'). Lo podríamos representar gráficamente de la siguiente forma:



donde podemos ver que hemos representado los cuatro caracteres como cuatro **char** individuales, consecutivos y enumerados con los índices desde el 0 al 3.

Para este nuevo tipo están definidas las operaciones básicas de entrada y salida, es decir, es posible leer cadenas desde **cin** con el operador **>>** y escribir cadenas en **cout** con el operador **<<**. También se pueden hacer asignaciones a variables de tipo **string** con el operador **=**.

Ejercicio 5.1 — La cadena “Hola mundo”. Escriba un programa que declare un objeto de tipo **string**, que lo imprima en la salida estándar para comprobar que está vacío, que a continuación lo modifique asignándole un literal con el mensaje **"Hola mundo"**, y que finalmente lo escriba en la salida estándar para confirmar que almacena dicha cadena.

5.3 Operaciones básicas con **string**

El tipo **string** es un contenedor de n caracteres, donde n es el tamaño, que puede ser cero (cadena vacía) o mayor que cero si contiene caracteres. Además, están almacenados como una secuencia ordenada por medio de un índice.

Para trabajar con este tipo de dato podemos usar distintas formas de declaración de objetos. Algunas de ellas son:

1. Declarando un objeto sin parámetros: **string str**. En este caso, el objeto es una cadena de longitud cero (vacía).
2. Declarando un objeto con un parámetro: **string str(cad)** donde **cad** es una cadena de caracteres delimitada por comillas dobles.
3. Declarando el objeto con dos parámetros: **string str(n, c)** donde **n** es el tamaño y **c** el carácter para inicializar cada una de las **n** posiciones de la cadena.

5.3.1 E/S de **string**

La lectura y escritura de cadenas desde un flujo (por ejemplo, **cin**) o a un flujo (por ejemplo, **cout**) se puede realizar de una forma similar a los tipos de datos básicos. Sin embargo, es importante recordar que el operador **>>** de lectura de datos desde **cin** considera que los “espacios blancos” -espacios, tabuladores, saltos de línea, etc.- separan datos.

Por tanto, si realizamos la lectura de un objeto de tipo **string** desde **cin**, primero se ignorarán los espacios blancos que se encuentren en la entrada, y después se cargarán todos los caracteres que haya en ésta, hasta que se localice un carácter separador o el fin de la entrada.

En muchos problemas, esta solución no es suficiente, ya que es necesario leer un conjunto de caracteres que incluyen también caracteres separadores. Por ejemplo, si leemos el nombre y apellidos de una persona, o la dirección postal, donde se incluyen espacios. Para resolver estos problemas, usaremos la función **getline** con la siguientes sintaxis²:

```
getline(f, str)
```

donde **f** corresponde a la entrada (por ejemplo, **cin**) y **str** corresponde a un objeto de tipo **string**. El resultado de la llamada a esta función es que se extrae una línea desde **f** y se almacena en el objeto **str**, que pierde el valor que tuviera antes de la llamada a la función. Note, por tanto, que la función consumirá todos los caracteres de entrada hasta que encuentre un salto de línea ('**\n**'), es decir, podrá incluir espacios, tabuladores, etc.

Ejercicio 5.2 — Lectura con **getline.** Escriba un programa que declare un objeto de tipo **string**, que lo lea haciendo uso del operador **>>** y que finalice escribiendo el contenido de dicho objeto en **cout**. Como entrada a dicho programa pruebe con la cadena “me gustan los ordenadores”. Una vez comprobado el funcionamiento de ese operador, cambie la lectura de datos y utilice la instrucción **getline**. Pruebe a ejecutar el programa y a dar la misma entrada de antes.

5.3.2 Tamaño y acceso a los caracteres

Un **string** se puede ver como una secuencia de n caracteres que se indexa desde cero en adelante. A cada uno de esos caracteres se puede acceder de manera individual con **at**, pasándole el índice del carácter a procesar. Este índice debe estar en el rango válido de la cadena, es decir, debe ser un entero del intervalo $[0, n-1]$.

Para obtener el tamaño de la cadena, podemos usar **size()**, que devuelve el número de caracteres que contiene (cero en caso de estar vacía).

²La función nos ofrece más posibilidades. En temás más avanzados se estudiarán, junto con otras funciones que pueden servirnos para resolver problemas como los que se han indicado.

```
#include <iostream>
#include <string> // Para usar cadenas

using namespace std;

int main()
{
    string s;
    cout << "Introduzca una cadena "
        "de caracteres y pulse \"Intro\":"
        << endl;
    getline(cin, s);
    for (int i=0; i< s.size(); ++i)
        if (s.at(i) == ' ')
            s.at(i) = '-';
    cout << "Separaciones: " << s << endl;
}
```

Es importante que observe algunos detalles de este programa:

- En el mensaje que se presenta en el primer `cout` se han incluido comillas dobles. Ha sido necesario usar la barra invertida para que se consideren como un carácter de la cadena. Además, una cadena se puede escribir en distintas líneas como dos cadenas entre comillas dobles. Para el compilador, será igual que haber escrito una cadena más larga de una vez.
- Para realizar una operación sobre un objeto de tipo `string`, debemos escribir el nombre del objeto, seguido de un punto, y seguido con la función y parámetros que corresponden a la operación deseada.
- En el cuerpo del bucle podemos ver que `at` se puede usar para consultar un carácter (en la condición del `if`) y para seleccionar un carácter para poder asignarle un valor. Es decir, con la operación `at` podemos acceder al carácter de una posición, tanto para consultarla, como para modificarlo.
- Observe cómo los literales carácter se deben escribir con comillas simples.

El efecto de este programa es que modifica la línea que se lee, sustituyendo los caracteres espacio por caracteres `'-'`. Finalmente, escribe la nueva cadena modificada como resultado. Un ejemplo de su ejecución es:

```
Introduzca una cadena de caracteres y pulse "Intro":
Hola mundo. Esta es mi primera lectura getline.
Separaciones: Hola-mundo.-Esta-es-mi-primer-a-lectura-getline.
```

Ejercicio 5.3 — Palíndromos. Escriba un programa que lea una palabra y diga si es un palíndromo, es decir, si se lee igual de derecha a izquierda que de izquierda a derecha. Por ejemplo, la palabra “radar” es un palíndromo. Para resolverlo, no es necesario que considere casos especiales de mayúsculas, tildes, signos de puntuación, etc.

Además de la operación `at`, podemos usar los corchetes para poder referirnos a un elemento de un `string`. En lugar de escribir `s.at(i)`, usamos la sintaxis `s[i]`. La diferencia fundamental está en que:

- En la operación `at` se comprueba que el índice está en el rango correcto. Si no lo está, se generará un error que el programa podría gestionar. Los detalles de cómo se gestiona este error los estudiará en un curso más avanzado. A nivel de este curso, el programa terminará indicando el error, lo que nos dará oportunidad de corregir el acceso incorrecto.
- En la operación `[]` no se hacen comprobaciones. Por tanto, se intenta realizar la operación aunque estemos en un índice incorrecto. Esto hace que sea muy peligroso, pues accedemos a posiciones de memoria incorrectas, aunque permite que el código no pierda tiempo en las comprobaciones si estamos seguros de que son correctas.

En la práctica, la sintaxis de corchetes es más simple y más eficiente, y además, es la misma que se usa para los vectores y cadenas en C, por lo que es la más habitual. Por supuesto, su uso obliga a tener mucho cuidado, pues tenemos que garantizar que no nos salimos del rango válido.

5.3.3 E/S combinada con otros tipos

Es frecuente realizar programas en los que se alternan las lecturas de cadenas usando `getline` con la lectura de otros tipos de datos como números o caracteres. En estas situaciones hay que tener muy presente el mecanismo de E/S que utiliza C++. Para la lectura de estos otros tipos usaremos la sintaxis habitual con `cin>>`.

La lectura de datos con `cin>>` lleva implícito algún tipo de procesamiento por parte de C++. En particular, esta lectura descarta automáticamente los caracteres que se consideran separadores. Por ejemplo, dado el siguiente código:

```
char let1, let2, let3; // Lectura de tres letras
cout << "Escriba tres letras: ";
cin >> let1 >> let2 >> let3;
cout << "Las letras son: " << let1 << ", " << let2 << ", " << let3 << endl;
int num1, num2; // Lectura de dos enteros
cout << "Escriba dos números: ";
cin >> num1 >> num2;
cout << "Los números son: " << num1 << ", " << num2 << endl;
```

las siguientes podrían ser diferentes ejecuciones del mismo:

```

Escriba tres letras: abc
Las letras son: a, b, c
Escriba dos números: 27 45
Los números son: 27 45

```

```

Escriba tres letras: a     b     c
Las letras son: a, b, c
Escriba dos números: 27

45
Los números son: 27 45

```

```

Escriba tres letras: a
b
c
Las letras son: a, b, c
Escriba dos números:      27          45
Los números son: 27 45

```

es decir, el resultado de la lectura es el mismo independientemente de si hemos usado unos u otros separadores (retorno de carro, espacios en blanco o tabuladores). En el caso de los datos de tipo **char** ni siquiera se requieren esos separadores, como puede ver en la primera ejecución.

Como norma podemos pensar que `cin>>` se salta aquellos separadores que van delante del dato que estamos leyendo y que detiene la lectura justo en el último carácter que forma parte del dato leído. Considere el siguiente código:

```

int x, y;
string cad;
cin >> x >> y;
getline(cin, cad);

```

Suponga que introducimos desde el teclado la secuencia "`234 123\n palabra`". La lectura de la variable `x` extraerá los caracteres "`234`" dejando pendientes para la próxima lectura "`123\n palabra`". Observe que el espacio de separación queda pendiente. La lectura de `y` comenzará leyendo el espacio en blanco, que será descartado, y continuará leyendo "`123`" quedando pendiente de lectura "`\n palabra`". Observe de nuevo que quedan pendientes los caracteres de separación (salto de línea y espacio).

La próxima instrucción que se ejecuta es `getline`, que como recordará lee caracteres hasta encontrar el siguiente salto de línea. Al ser este el primer carácter que se encuentra, el resultado es que termina su ejecución habiendo leído la cadena vacía y quedando pendiente de lectura "`\n palabra`" (incluyendo el espacio inicial).

La solución a este ejemplo concreto pasaría por descartar el carácter especial '`\n`' antes de ejecutar `getline`. Para ello podemos usar una función que lee un único carácter sin importar que éste pueda ser o no un separador:

```
car = cin.get();
```

de manera que nos quedase este código:

```

int x, y;
string cad;
cin >> x >> y;
cin.get();           // Para eliminar el '\n'
getline(cin, cad);

```

Evidentemente esta es una solución para un caso muy concreto. ¿Qué ocurriría si en lugar de un único separador se hubiesen escrito varios separadores consecutivos? En ese caso `cin.get()` sólo extraería el primero de ellos y el código no sería correcto.

Ejercicio 5.4 — Calificaciones de alumnos. Haga un programa que lea las calificaciones que han obtenido los alumnos de una asignatura. El programa comenzará leyendo el curso (un valor **int**), el grupo (un valor **char**) y el nombre de la asignatura (un dato de tipo **string**). A continuación, el programa pedirá el número total de alumnos matriculados y

luego, para cada uno de ellos, leerá su nombre y apellidos en una única cadena de caracteres junto con su calificación numérica que será un número real.

Al finalizar la lectura de datos, el programa mostrará el curso, el grupo, el nombre de la asignatura y el nombre del alumno con mayor nota. Observe que:

- Debe respetar las indicaciones sobre los tipos de datos que se piden.
- No debe utilizar vectores: los datos se procesarán conforme se lean.
- El nombre y apellidos deben almacenarse en una única cadena.

5.4 Más operaciones con string

Las operaciones vistas en las secciones previas permiten trabajar con las cadenas de una forma muy elemental: carácter a carácter. Sin embargo, hay ciertas operaciones que son de mayor complejidad y bastante frecuentes como la concatenación de cadenas, la búsqueda o el borrado de subcadenas. En esta sección mostraremos cómo usar algunas de estas operaciones.

5.4.1 Concatenación de cadenas

Los operadores `+=` y `+` nos permiten concatenar cadenas y caracteres. El primero de ellos lo podemos usar para añadir nuevos caracteres al final de una cadena, y el segundo para crear una cadena como concatenación de otras. Tenga en cuenta que para usar el operador `+` para concatenar dos operandos, será necesario que al menos uno de ellos sea de tipo **string**. Un ejemplo de su uso es el siguiente:

```
string str; // Comienza vacía
str+= string("Prueba: "); // Añadimos una palabra
str+= string("Hola") + ' ' + string("mundo");
str+= '.'; // str vale string("Prueba: Hola mundo.")
```

Observe que la suma de la tercera línea realiza primero la suma de un **string** con un **char**, que da como resultado un **string**, y al resultado le suma la última cadena. Por otro lado, puede ver que en la última línea hemos concatenado un carácter a la cadena.

Ejercicio 5.5 — Inversión de una cadena. Escriba un programa que lea una cadena de caracteres y que muestre dicha cadena invertida. Por ejemplo, si la cadena de entrada es `"Hola mundo"` el resultado sería `"odnum aloH"`. Para ello deberá declarar una nueva variable de tipo cadena en la que almacenará la cadena invertida. Use los operadores de concatenación de cadenas.

5.4.2 Comparación de cadenas

Los operadores relacionales que hemos usado para los tipos predefinidos también están disponibles para el tipo **string**. Cuando usamos un operador relacional, se realiza una comparación lexicográfica, es decir, una cadena es menor que otra si apareciese antes en el diccionario. Por tanto, el compilador genera una comparación carácter a carácter para ordenar cadenas.

Lógicamente, el resultado de la comparación dependerá del orden que tengan los caracteres que componen la cadena, ya que al final son éstos los que se comparan uno a uno. En nuestros programas, en los que tenemos en cuenta un orden establecido por la tabla ASCII, un carácter será menor que otro si aparece antes en la tabla ASCII.

Por supuesto, dos **string** serán iguales si tienen exactamente el mismo tamaño y con los caracteres idénticos uno a uno.

Ejercicio 5.6 — Ordenación alfabética. Implemente un programa que lea los nombres de dos personas y que los muestre ordenados alfabéticamente. Veamos algunos ejemplos de ejecución:

```
Consola
Dígame el nombre de la primera persona: Antonio
Dígame el nombre de la segunda persona: Javier
La ordenación de ambos es:
Antonio
Javier
```

```
Consola
Dígame el nombre de la primera persona: Javier
Dígame el nombre de la segunda persona: Antonio
La ordenación de ambos es:
Antonio
Javier
```

Debe tener en cuenta que los caracteres en minúsculas son diferentes a los caracteres en mayúsculas. ¿Qué está ocurriendo en la siguiente ejecución?

```
Dígame el nombre de la primera persona: Javier
Dígale el nombre de la segunda persona: antonio
La ordenación de ambos es:
Javier
antonio
```

Ejercicio 5.7 — Palíndromos (versión 2). Reescriba el ejercicio 5.3 para comprobar si una cadena es o no un palíndromo. Esta vez aprovechará que sabe cómo invertir una cadena (ejercicio 5.5) para resolverlo. Deberá invertir el texto leído desde la entrada estándar y compararlo con el texto original.

5.4.3 Extracción de subcadenas

Podemos usar la operación `substr` para devolver el valor de una subcadena contenida en otra cadena. Para ello, le pasamos como parámetros el índice a partir del que empieza la subcadena y el tamaño.

Tenga en cuenta que el índice que indica el principio puede ser un valor desde 0 hasta el tamaño (ambos incluidos) y el tamaño a extraer puede ser un número que sobrepase el final de la cadena, pues indica un número de caracteres máximo. Si damos un tamaño demasiado grande, devolverá la subcadena hasta el último carácter de la original, aunque para conseguir este efecto, también podemos llamar a la función sólo con el primer parámetro.

Un ejemplo en el que usamos esta función es el siguiente:

```
string cad;
cad = string("Extraer subcadena de cadena");
string s = cad.substr(8, 9);           // s vale string("subcadena")
cout << "Subcadena desde \" " << cad << "\";" << s << endl;
s = cad.substr(21);                  // Desde el 21 -> s vale string("cadena")
cout << "Final: " << s << endl;
```

y el resultado de su ejecución sería este:

```
Subcadena desde "Extraer subcadena de cadena": subcadena
Final: cadena
```

5.4.4 Borrado de subcadenas

La función `erase` se usa para borrar caracteres de una cadena. Como en otras funciones, podemos especificar una subcadena como dos valores enteros –índice y longitud– o como un único valor entero que indica el índice y que se refiere a todos los caracteres desde esa posición hasta el final. Un ejemplo es el siguiente:

```
string str("Hola mundo");
str.erase(4,1); // Elimina el espacio -> str vale string("Holamundo")
cout << str << endl;
str.erase(4);   // Elimina desde el 4 -> str vale string("Hola")
cout << str << endl;
```

Ejercicio 5.8 — Palíndromos con caracteres no alfabéticos. Reescriba el ejercicio 5.3 para comprobar si una frase es o no un palíndromo. Esta vez deberá tener en cuenta la posibilidad de que la frase contenga espacios en blanco o signos de puntuación y eliminarlos antes de proceder a la comprobación. Tenga en cuenta también que la frase podría tener mayúsculas y minúsculas. Ejemplos de palíndromos válidos serían: “Dabale arroz a la zorra el abad” o “Nada, yo soy Adan”. Observe que no usamos letras con tilde.

5.4.5 Inserción de subcadenas

Podemos usar la operación `insert` para insertar unas cadenas dentro de otras. Los parámetros son:

1. La posición donde insertar. Esta posición también podría ser el tamaño de la cadena, es decir, una posición detrás de la última, lo que haría que equivaliese a una concatenación.
 2. La cadena a insertar. Se puede dar como una cadena entre comillas dobles o un `string`.
- Un ejemplo de uso de esta función es el siguiente:

```
string str("Esto está mal");
str.insert(4, " no");           // str vale string("Esto no está mal")
string punto(".");
str.insert(str.size(), punto); // str vale string("Esto no está mal.")
```

5.4.6 Reemplazo de subcadenas

Podemos reemplazar una subcadena por otra. Para ello, podemos usar la operación *replace*.

Esta función tiene tres parámetros:

- Los dos primeros son similares a los de la función *substr*, pues determinan la subcadena que será reemplazada.
- El tercero es la nueva subcadena, que puede darse con un valor concreto como una serie de caracteres entre comillas dobles o como un **string**.

Un ejemplo en el que se usa esta operación puede ser el siguiente:

```
string str1;
str1= string("Esto es un ejemplo");
string str2("otro");
str1.replace(8,2,str2);      // str1 vale string("Esto es otro ejemplo")
str1.replace(4,0, " sí que"); // str1= string("Esto sí que es otro ejemplo")
```

Ejercicio 5.9 — Composición de una frase. Escriba un programa que lea una línea inicial que contiene una cadena.

Esta línea contendrá caracteres '`\#`'. El programa, a continuación leerá tantas líneas como caracteres '`\#`' tenga la línea inicial, reemplazando cada uno de ellos con las correspondientes líneas leídas. Finalmente, escribirá la cadena resultante. Para probarlo, introduzca las siguientes líneas:

```
Hay dos maneras de #: una es #, y la otra es #. #.
diseñar software
hacerlo # sea obvia su falta de eficiencias
tan # que
simple
hacerlo # no haya # obvias
tan # que
complejo
deficiencias
C.A.R. Hoare
```

Observe que las líneas que se introducen pueden, a su vez, tener nuevos caracteres que sustituir.

5.4.7 Búsqueda en cadenas

Para localizar subcadenas dentro de una cadena, disponemos de las operaciones *find* y *rfind*, que permiten buscar una cadena en otra, la primera desde el principio y la segunda desde el final.

Estas operaciones se pueden llamar con un único parámetro: la cadena a buscar, ya sea pasando una cadena entre comillas dobles o un objeto de tipo **string**. En este caso, *find* localiza la primera ocurrencia y *rfind* la última.

También es posible usarlas con dos parámetros, el primero como hemos indicado y el segundo para pasarle un índice a partir del cual empezar a buscar. De esta forma, podemos obtener no sólo la primera o última ocurrencia, sino todas las que aparezcan en la cadena.

Cuando la función localiza la cadena, devuelve como resultado el índice donde se encuentra. En el caso de no encontrarla, devuelve la constante **string**::*npos*. En el caso de que desee declarar variables que controlen posiciones, debería usar el tipo **string**::*size_type*, que en la práctica va a ser equivalente a algún tipo entero sin signo.

Un ejemplo de uso de estas funciones es:

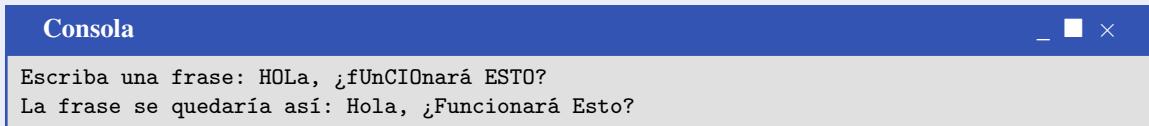
```
string str("Solía pensar que el cerebro era el órgano más importante "
          "del cuerpo, hasta que me di cuenta de quién era el que me "
          "estaba diciendo eso (Emo Philips)");
string cad("el");
string::size_type pos= str.find(cad);
if (pos!=string::npos) {
    cout << "La palabra \" " << cad << "\" se repite... ";
    if (str.find(cad,pos+1)!=str.rfind(cad))
        cout << " más de dos veces.";
    cout << endl;
}
```

Ejercicio 5.10 — Ocurrencias de una palabra. Escriba un programa que lea múltiples líneas de texto hasta que se encuentre una que contenga únicamente los 5 caracteres "#FIN#". Después, solicitará una cadena e indicará las veces que se repite en las líneas de texto introducidas. El programa repite la pregunta y busca la nueva cadena hasta que le damos la cadena "#FIN#".

Realice el programa acumulando las líneas de texto en un único objeto de tipo **string**, separando las distintas líneas con un carácter '**\n**', para localizar sobre él las cadenas a buscar.

5.5 Ejercicios adicionales

Ejercicio 5.11 — Mayúsculas. Haga un programa que lea una frase cualquiera y que la transforme de la siguiente forma: todas las letras deberán transformarse en minúsculas salvo aquellas que sean el comienzo de una palabra, que deberán transformarse en mayúsculas. Se considera que la separación entre palabras es cualquier carácter que no sea una letra (espacio, dígitos, signos de puntuación, etc). Veamos un ejemplo de ejecución:



Ejercicio 5.12 — Cambio de base. Escriba un programa que lea un número entero y una cadena que corresponden a la base y al número en dicha base. Observe que el segundo parámetro se lee como cadena porque es posible que contenga letras.

Para visualizar el número en distintas bases, el programa solicitará repetidamente la base en la que se desea expresar, escribiendo el resultado en la salida estándar. Esta pregunta se repetirá hasta que se dé una base menor que dos para indicar que el programa termine.

Tenga en cuenta que el número en base B mayor que 10 se puede almacenar en una cadena con los dígitos decimales y los B-10 primeras letras del alfabeto. Para resolver el problema, escriba algoritmos para pasar de base B a 10 y al revés.

Ejercicio 5.13 — Simplificar espacios. Escriba un programa que limpie una cadena de texto leída desde la entrada estándar como una línea. Esta limpieza consistirá en eliminar los espacios iniciales, los espacios finales, y en caso de que haya varios espacios consecutivos intermedios, simplificarlos en un único espacio. Tenga en cuenta que si la cadena tiene sólo espacios, deberá convertirla en la cadena vacía. Veamos un ejemplo de ejecución:



Ejercicio 5.14 — Codificación. Escriba un programa para codificar un texto, es decir, un conjunto de líneas que termina con una línea especial "#FIN#". Para ello, cambiará unas letras por otras predefinidas. El resultado del texto es otro texto –las mismas líneas– con los caracteres cambiados. Por ejemplo, si encuentra alguna de las letras de la siguiente cadena:

" !\"#\$%&' ()*+,,-./0123456789;:<=>?@ABCDEFGHIJKLMN"
"OPQRSTUVWXYZ[\\]^`abcdefghijklmnopqrstuvwxyz{|~^"

deberá cambiarse por la correspondiente de la siguiente cadena:

"oEJ1v2?i:TL[BPzwdY%6~;3QN| (5tA_&K=, up!RhIHx@*f8]—" "7Fa)sn^g1m+OW\}Sae0cMbG'\\" "#ZV DXvU<CS/Jr.k9'4"

Así, si se introduce el siguiente texto:

"Hay una antigua historia sobre una persona que quería que su ordenador fuese tan fácil de utilizar como su teléfono. Estos deseos se han hecho realidad, ya no sé cómo usar mi teléfono"

Bjarne Stroustrup

#ETIN#

Se obtendrá como resultado lo siguiente:

jISroCVSoSV<'bCSoG'U< y'SoU gyroCVSoDcyU VSoXCCoXCCyíSoXCCoUC
y0cVSO yoMCcUco<SVoMáé#o0coC<#.Syoe Z oUCo<c#éM V zo!U< U
oCUC UoUcoGSVoGceG oycS#!OSOBorSoV oUéoe6Z oCUSvoZ'o<c#éM V i

```
, "SyVcoa<y CU<yCD  
#FIN#
```

Pruebe a introducir el texto codificado para ver el resultado de volver a codificarlo.

Tipos compuestos: 2

6

Funciones

Introducción.....	47
Notación	
Parametrización de las funciones	48
El paso por valor	
Las funciones de tipo void	
El paso por referencia	
Conversiones	
Diseño de funciones	52
La separación entre la E/S y los cálculos	
Diseño descendente (top-down)	
Gestión de errores	57
Devolución de un valor de error	
Errores graves irrecuperables	
Precondiciones y postcondiciones	
Valores por defecto y sobrecarga	60
Parámetros con valores por defecto	
Sobrecarga de funciones	
Ejercicios adicionales.....	61

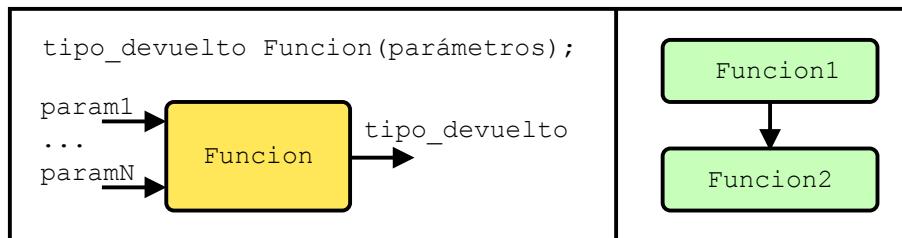
6.1 Introducción

El objetivo de este capítulo es que el alumno practique con problemas para los que se necesitan funciones. En este guión se pretende:

1. Practicar con pasos por valor y por referencia, entendiendo que el paso por valor crea nuevos objetos a partir de los originales, mientras que las referencias no son más que otros nombres para los mismos objetos.
2. Entender cómo un problema relativamente complicado se puede separar en subproblemas de menor tamaño, más fáciles de resolver y de reutilizar.
3. Comprobar que la modificación de una función, sin cambiar su especificación, facilita el mantenimiento de programas al aislar los detalles internos del algoritmo, del código de llamada que lo usa.
4. Exponer nuevas posibilidades con funciones: la sobrecarga de funciones y los parámetros por defecto.
5. Realizar algunas indicaciones sobre el diseño, para que el resultado de crear una solución en base a módulos sea más efectivo.
6. Ofrecer una primera discusión sobre la gestión de errores en módulos. Con ello, podremos entender mejor el funcionamiento de la función `main`, así como otras herramientas del lenguaje, como la macro `assert` o la función `exit`.

6.1.1 Notación

A lo largo de este capítulo verá que aparecen algunas indicaciones sobre el diseño de las funciones que se pide implementar. En concreto, podrá ver diagramas como los de la siguiente figura:



El diagrama de la izquierda muestra el prototipo de una función que recibe unos argumentos y devuelve un dato junto con un dibujo que enfatiza el hecho de que las funciones se comportan como “cajas negras” que reciben unos datos, realizan un cierto procesamiento de los mismos y devuelven datos.

Cuando el software se modulariza en varias funciones, casi con toda probabilidad algunas de ellas harán uso de otras. El diagrama de la derecha muestra un ejemplo en el que una función *Funcion1* hace uso de otra llamada *Funcion2*, es decir, en algún punto de la implementación de *Funcion1* se hace una llamada –o más– a *Funcion2*.

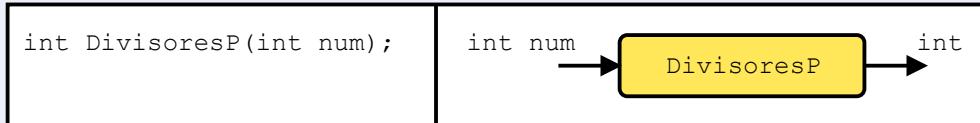
6.2 Parametrización de las funciones

En esta sección se estudiarán los mecanismos básicos que usan las funciones para comunicarse entre ellas: el paso de parámetros y la devolución de resultados.

6.2.1 El paso por valor

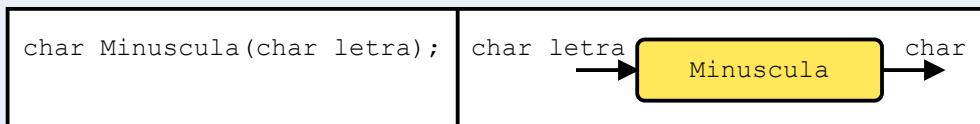
La forma más sencilla que hemos presentado para pasar los datos a una función es el *paso por valor*. En este caso, los parámetros formales se definen para almacenar una copia del dato de entrada que queremos pasar a la función.

Ejercicio 6.1 — Divisores propios. Implemente una función que reciba un número entero positivo y que devuelva el número de divisores propios que tiene (divisores distintos del propio número). El 1 se considera divisor propio de cualquier número mayor que 1.



Implemente a continuación una función `main()` que pida valores al usuario desde la entrada estándar y que muestre cuántos divisores propios tiene cada uno. El programa finalizará cuando el usuario dé un valor menor o igual que cero como entrada.

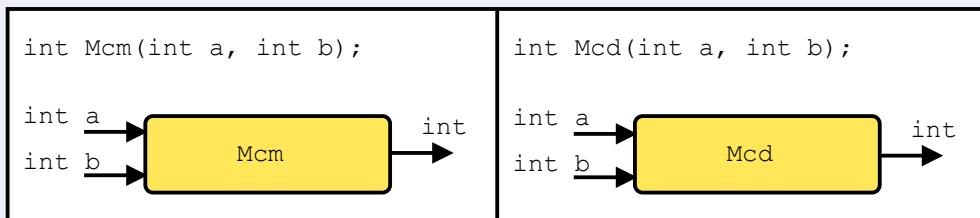
Ejercicio 6.2 — Conversión a minúsculas. Implemente una función que reciba un carácter y que en caso de ser una letra devuelva su correspondiente minúscula. En otro caso devolverá el mismo carácter. No se permite el uso de funciones de biblioteca (en particular `tolower()`).



Implemente una función `main()` que lea una letra desde la entrada estándar y que muestre su correspondiente minúscula.

Aunque en los ejemplos anteriores hemos creado funciones que reciben un dato de entrada y devuelven uno de salida, las funciones pueden ser más complejas y recibir varios datos de entrada.

Ejercicio 6.3 — Máximo común divisor y mínimo común múltiplo. Implemente dos funciones: una para calcular el máximo común divisor y otra para calcular el mínimo común múltiplo de dos números enteros (positivos o negativos). Los prototipos deben ser estos:



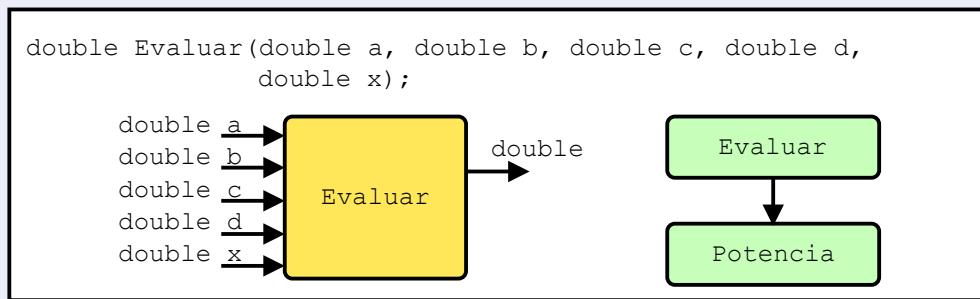
A continuación implemente un `main()` que pregunte dos números al usuario y que muestre el resultado de ejecutar ambas funciones. Para calcular el máximo común divisor utilice el algoritmo de Euclides^a

^ahttp://es.wikipedia.org/wiki/Algoritmo_de_Euclides

Ejercicio 6.4 — Potencia de un número. Diseñe e implemente una función que reciba dos números enteros a y b y que devuelva la potencia “ a elevado a b ” (a^b). Esta función no puede utilizar ninguna función de biblioteca (en particular `pow()`).

Es más simple resolver un problema reutilizando módulos desarrollados anteriormente. Por ejemplo, podemos reutilizar la función anterior en un nuevo problema.

Ejercicio 6.5 — Evaluación de un polinomio. Escriba un programa que lea 4 números a , b , c y d que definen el polinomio $ax^3 + bx^2 + cx + d$, que lo evalúe en un punto y que muestre ese valor. Para ello, defina la función *Evaluar* que recibe los cuatro valores junto con un valor x donde evaluarlo, y que devuelve un número resultado de la evaluación. Para resolverlo, por un lado deberá reutilizar la función *Potencia* del ejercicio 6.4, y por otro, evitar declarar ninguna variable local en la función *Evaluar*.



Ejercicio 6.6 — Bisección de un polinomio de grado 3. Considere el ejercicio 3.9 en el que se aborda el problema de encontrar las raíces de un polinomio mediante el método de bisección. Reescriba dicho programa de manera que haga uso de la función *Evaluación* del ejercicio 6.5 para buscar una raíz de un polinomio de grado 3. El programa lee los 4 valores que definen el polinomio, los dos valores de los extremos del intervalo donde buscar, y un valor de precisión en la solución. Como resultado escribe en la salida estándar la solución encontrada.

Para ello, tendrá que definir una nueva función *Bisección*, que recibe todos esos datos y devuelve la solución.

Además de permitir la reutilización de código, otro aspecto fundamental de las funciones es que al abstraer el algoritmo concreto que se usa para su implementación, resulta mucho más simple modificarlas, ya sea para arreglar errores o para mejorar su comportamiento.

Ejercicio 6.7 — Algoritmo de evaluación de Horner. Para mejorar la eficiencia del programa del ejercicio 6.6 se ha decidido modificar la función que evalúa el polinomio para que use el algoritmo de *Horner*. Tenga en cuenta que este algoritmo se basa en reescribir la expresión del polinomio de la siguiente forma:

$$p(x) = ax^3 + bx^2 + cx + d = (((ax + b)x + c)x + d)$$

Con este algoritmo se reduce notablemente el número de multiplicaciones y por tanto se mejora la eficiencia. Observe que sólo hay que modificar la parte interna de la función *Evaluación*. Puesto que la cabecera de dicha función no se ha modificado, no es necesario modificar el resto de funciones del programa.

6.2.2 Las funciones de tipo void

Las funciones de tipo **void** tienen la peculiaridad de que no devuelven ningún dato de forma explícita con **return**. Se utilizan cuando ocurre alguna de estas circunstancias:

1. La función no necesita devolver datos a quien realiza su llamada. Un ejemplo característico son las funciones que se dedican sólo a escribir datos en la salida estándar.
2. La función necesita devolver más de un dato. Aunque uno de los datos de salida podría devolverse con **return** y otros mediante parámetros pasados por referencia, a veces se opta por devolverlos todos de forma homogénea a través de los parámetros y no usar **return**. Este caso lo veremos en la siguiente sección.

En el siguiente ejemplo vemos una función que recibe como argumento un entero y muestra en la salida estándar la tabla de multiplicar de dicho número:

```

#include <iostream>
using namespace std;

void ImprimirTabla(int n)
{
    for (int i=1;i<=10;++i)
        cout << n << " x " << i << " = " << n*i << endl;
}

int main()
{
    int n;

    cout << "Indique la tabla que desea imprimir: ";
    cin >> n;
    ImprimirTabla(n);
}
    
```

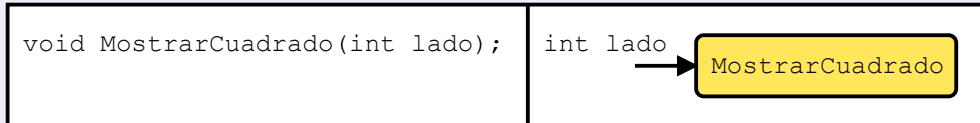
Observe que la forma de usarla es distinta a la de otras funciones que devuelven datos –en este ejemplo la llamada es una línea independiente–. Por ejemplo, la función **sqrt** devuelve la raíz cuadrada, por lo que la llamada a la función se inserta en el punto donde queremos usar el valor devuelto. Las funciones **void** no devuelven nada, por lo que no tiene sentido llamarlas en casos como:

- En mitad de una expresión: no devuelve nada, no puede ser un operando.
- A la derecha de una asignación: no devuelve nada, no devuelve el valor a asignar.
- En una instrucción de salida de datos (**cout** <<): no devuelve nada, **cout** no tiene nada que escribir.
- Etc.

Por ejemplo, no tendrían sentido llamadas a la función como las siguientes:

```
y = 2 * ImprimirTabla(n);
x = ImprimirTabla(n);
cout << ImprimirTabla(n);
```

Ejercicio 6.8 — Mostrar cuadrados. Considere de nuevo el ejercicio 3.13. Implemente una función que reciba como argumento un valor entero y que muestre en la salida estándar un cuadrado cuyo lado sea igual a ese valor entero.



Una vez hecha la función, realice un programa que muestre los cuadrados de lado 1, 2, 3, 4, ..., N siendo N un valor entero mayor que cero leído por la entrada estándar. A continuación se muestra un ejemplo de ejecución:

```
Diga un valor para N: 3
*
**
**
***
```

6.2.3 El paso por referencia

Hasta ahora todas las funciones que hemos visto reciben datos de entrada para su procesamiento mediante el mecanismo del paso por valor. En esta sección, vamos a insistir en la posibilidad de que una función pueda tener *parámetros por referencia*. En este caso, el argumento de llamada o parámetro actual será una variable que se modificará dentro de la función.

Aunque inicialmente pueda pensar que este tipo de paso es más complicado, lo cierto es que desde el punto de vista del compilador se puede considerar más simple. En el paso por valor, cada parámetro formal es un nuevo objeto que se inicializa con el valor que ofrecemos en el correspondiente argumento de llamada. Sin embargo, en el paso por referencia la situación es más simple, ya que el compilador simplemente tiene que tener en cuenta que hay un nuevo nombre para un objeto que ya existe en el punto de llamada. Dicho de otra forma, se ahorra tener que reservar un objeto y, en su lugar, simplemente crea un nombre que también se refiere¹ a un objeto del código de llamada. Por ejemplo, consideremos una función que lee los extremos de un intervalo:

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

void LeerIntervalo(double& izq, double& der)
{
    do {
        cout << "Introduzca dos valores para definir el intervalo: ";
        cin >> izq >> der;
    } while (izq>=der);
}

int main()
{
    double primero, segundo;

    LeerIntervalo(primerito, segundo);
    cout << "Resultado: " << primero << " " << segundo << endl;
}
```

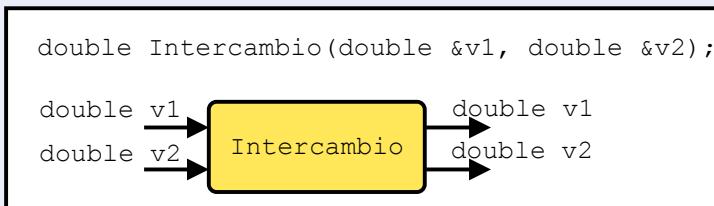
¹Note que estamos presentando este parámetro formal como *una referencia* a otro objeto.

En este programa, sólo existen los objetos *primero* y *segundo* que se han declarado en la función **main**. Dentro de la función *LeerIntervalo*, simplemente disponemos de dos nombres adicionales para esos objetos. Es decir, se refieren a los mismos objetos.

Lógicamente, cuando modificamos una referencia dentro de una función, estamos modificando el objeto correspondiente en el código de llamada. De hecho, para el compilador no hay diferencia entre el objeto *primero* y el objeto *izq*. Son dos nombres para una misma cosa. Por supuesto, lo que sí está muy claro es que en la función **main** el nombre *izq* es desconocido, así como en la función *LeerIntervalo* el nombre *primero* también lo es.

En el ejemplo anterior, hemos visto una función que necesitaba dos parámetros por referencia para poder obtener el resultado del algoritmo. Es decir, eran parámetros de salida. Obviamente, el paso por referencia puede usarse para pasar valores de entrada y salida.

Ejercicio 6.9 — Intercambio de valores. Escriba una función *Intercambiar* para intercambiar el valor de dos variables de tipo **double**. Es decir, en la llamada se pasarán dos variables y, al finalizar, los valores de ambas estarán intercambiados. Implemente también un **main ()** para probar la función.



Observe en este esquema que hay más de un dato de salida de la función. Puesto que no es posible devolverlos mediante **return** se usa el mecanismo del paso por referencia, indicando así cuáles son los argumentos de la función que se consideran de salida de la misma.

Ejercicio 6.10 — Ordenar 2 valores. Escriba una función *Ordenar2* para ordenar los valores de dos variables de tipo **double**. Es decir, en la llamada se pasarán dos variables y, al finalizar, la primera variable tendrá el menor y la segunda el mayor de los valores. Para resolverlo, use la función *Intercambiar* del ejercicio 6.9.

Como puede comprobar en la solución del ejercicio, desde la función *Ordenar2* pasamos dos variables a la función *Intercambiar*. Realmente, estos dos objetos no están creados ahí, sino que son dos objetos que se han pasado, a su vez, desde otra función. Dicho de otra forma, son dos referencias a dos objetos que, declarados fuera de *Ordenar2*, se pasan a la función de intercambio.

Ejercicio 6.11 — Ordenar 3 valores. Escriba una función *Ordenar3* para ordenar los valores de tres variables de tipo **double**. Es decir, en la llamada se pasarán tres variables y, después de ella, la primera variable tendrá el menor, la segunda el mediano y la tercera el mayor de los valores. Para resolverlo, use la función del ejercicio 6.10 teniendo en cuenta que para que tres valores queden ordenados, basta con ordenar los dos primeros, luego los dos segundos y de nuevo los dos primeros.

Expresiones en los parámetros actuales

Como se ha dicho antes, en un paso por referencia el parámetro formal es una referencia al parámetro actual. Esto implica que cualquier modificación que se haga al parámetro formal también se está haciendo sobre el actual porque, de hecho, son lo mismo.

Ejercicio 6.12 — Literales y expresiones en los parámetros actuales por referencia. Considere el ejemplo que se ha usado al comienzo de esta sección para leer los valores de un intervalo y los ejercicios 6.9 y 6.10. Compruebe el efecto de realizar las siguientes llamadas a función:

```
LeerIntervalo(2,7);
Intercambiar(3,7);
Ordenar2(3.12+8.67, sqrt(7.31));
```

Podrá observar que se producen errores de compilación ¿Qué sentido encuentra en esos errores?

No tiene sentido que el parámetro actual sea algo diferente a una variable puesto que dentro de la función (el parámetro formal) se va a tratar como tal. La función podría usar dicho parámetro a la izquierda de una asignación y, por tanto, en ningún caso podría ser una expresión o un literal: debe ser el nombre –la referencia– de un objeto que puede almacenar un valor.

6.2.4 Conversiones

Después de presentar ejemplos de paso por valor y por referencia, resulta interesante destacar el comportamiento que podemos obtener cuando pasamos objetos de otro tipo, es decir, cuando el parámetro formal es de un tipo y el argumento que pasamos en la llamada es de otro.

El alumno conoce las conversiones automáticas del compilador —*conversiones implícitas*— que probablemente ha visto en la práctica cuando mezcla distintos tipos en expresiones o hace asignaciones de un tipo a otro. En el caso de las funciones ocurre lo mismo, es decir, el compilador puede aplicar conversiones de tipos.

Las conversiones se pueden aplicar tanto en el paso por valor como en el paso por referencia. Sin embargo, como hemos visto, son dos operaciones distintas, ya que el paso por referencia crea un nuevo nombre de un objeto que existe. Así, la forma en que funciona es distinta:

- En el paso por valor se crea un objeto que es copia del original. Por esta razón, es mucho más fácil entender y aplicar la conversión. De forma similar a cuando hacemos una asignación y se convierten los tipos desde el resultado de la expresión de la derecha en el objeto de la izquierda, en una llamada a función el resultado de una expresión se puede convertir al tipo de dato del parámetro formal.
- En el paso por referencia no tenemos la misma situación. Dado que estamos creando un nuevo objeto con el mismo nombre que otro existente, no podemos hacer una conversión, ya que al modificar el objeto dentro de la función, estaremos modificando un objeto en el código de llamada. Por tanto, en principio² tendremos en cuenta que dado que debemos darle un objeto del mismo tipo para crear el parámetro como sinónimo, no vamos a realizar conversiones.

Observe que este funcionamiento implica que cuando queremos pasar un parámetro por valor, podemos ofrecer cualquier expresión, incluyendo literales, constantes, expresiones, llamadas a otras funciones, etc. Siempre que el resultado de esa expresión sea un valor *compatible* con el parámetro. En cambio, ahora mismo, en el paso por referencia le pasaremos un objeto con el mismo tipo que el parámetro.

Finalmente, es importante tener en cuenta que en la devolución también podemos provocar una conversión automática. Por ejemplo, en la siguiente función devolvemos un entero a partir de un carácter. Los tipos no coinciden, y por tanto, el compilador convierte el carácter a su correspondiente valor entero en la tabla ASCII:

```
int Char2Ascii(char c)
{
    return c;
}
```

Como podemos ver, se realiza una conversión implícita de carácter `-char-` a entero `-int-`.

6.3 Diseño de funciones

A continuación se verán algunas pautas a tener en cuenta a la hora de decidir qué funciones debemos incluir en nuestros programas. Una norma ampliamente aceptada es la separación entre procesos de cálculo y procesos que interactúan con el usuario (por ejemplo, operaciones de E/S).

También se trabajará un método de diseño denominado *diseño descendente* que permite resolver problemas de cierta complejidad usando las funciones como una herramienta que permite descomponerlos en problemas más simples.

6.3.1 La separación entre la E/S y los cálculos

Es recomendable que el diseño de funciones se realice minimizando el acoplamiento³ y maximizando la cohesión⁴. Si quiere obtener implementaciones con estas características, debería evaluar algunos aspectos como la reusabilidad o la facilidad de modificación que tienen.

Una situación que surge incluso en los programas más simples, se refiere a la entrada y salida de datos. En general, nuestros programas implementan algoritmos que leen una serie de datos de entrada, los procesan, y escriben los resultados en la salida. Desde un punto de vista del diseño, no deberíamos mezclar las operaciones de E/S con el procesamiento de los datos.

La mayoría de las funciones de un programa se dedican exclusivamente a tareas de cálculo. La E/S de datos suele delegarse o bien a la función `main()` o bien a alguna función específica. De esta forma se consigue independizar los cálculos de la forma en que se obtienen los datos —la entrada— y la forma en que se presentan —la salida—.

Como consecuencia de esto, obtendremos algoritmos que se pueden *reutilizar* en contextos diversos sin importar ni de dónde provienen los datos de entrada a los mismos ni lo que se pueda hacer con los resultados a continuación.

Por ejemplo, considere que hubiésemos hecho esta implementación de la función *Evaluar* del ejercicio 6.5:

```
void Evaluar()
{
    double a, b, c, d;
    cout << "Introduzca los coeficientes del polinomio: ";
    cin >> a >> b >> c >> d;
    double x;
```

²Decimos que en principio ya que es fácil encontrar múltiples ejemplos de conversiones de tipos cuando estamos usando una referencia. Más adelante se estudiarán nuevas posibilidades y veremos que, efectivamente, se pueden realizar. Además, el paso por referencia y sus conversiones son un pilar fundamental para hacer funcionar la programación dirigida a objetos con C++.

³Grado de dependencia entre funciones.

⁴Grado de relación entre las partes que forman una función.

```

cout << "Introduzca el punto donde evaluar el polinomio: ";
cin >> x;
double eval = a*x***x + b*x**x + c*x + c;
cout << "El resultado de la evaluación es: " << eval << endl;
}

```

Como puede observar, se ha prescindido de los parámetros de la función y se ha incluido la E/S dentro de la propia función. Este diseño la hace completamente inservible para resolver cualquier otro problema que no sea exactamente "Leer un polinomio, leer un punto, evaluarlo en dicho punto y mostrar el resultado". En particular no serviría si pretendemos resolver el ejercicio 6.6 puesto que:

- El punto en donde se evalúa el polinomio no se lee desde la entrada estándar sino que es el programa quien lo calcula.
- Los coeficientes del polinomio no han de leerse en cada evaluación del mismo. Se leyeron una única vez antes de ejecutar el algoritmo de bisección.
- El resultado de la evaluación del polinomio no se utiliza para mostrarlo en la salida estándar, sino para realizar otros cálculos.
- Finalmente, si optásemos por realizar un programa que utilizase un método de interacción con el usuario diferente al de la E/S estándar, por ejemplo una interfaz gráfica, no tendría sentido enviar datos o pedir datos desde consola ya que ésta no sería visible para el usuario.

Podemos ver que la mezcla de distintas operaciones de esta función la hace menos útil para otros programas. Por otro lado, si fallara el resultado final, la corrección de errores implicaría tener en un mismo bloque todos los detalles mezclados, lo que dificulta la localización y corrección del error.

Ejercicio 6.13 — Fecha con formato. Construya un programa que lea una fecha desde la entrada estándar (día, mes y año), y que la muestre en la salida estándar con un formato que indique el día de la semana y el nombre del mes. También se indicará si el año es bisiesto o no. A continuación se muestran un par de ejecuciones de ejemplo:

```

Consola
Introduzca una fecha (día, mes, año): 21 2 2011
La fecha es: Lunes, 21 de febrero de 2011

```

```

Consola
Introduzca una fecha (día, mes, año): 13 8 2004
La fecha es: Viernes, 13 de agosto de 2004 (Bisiesto)

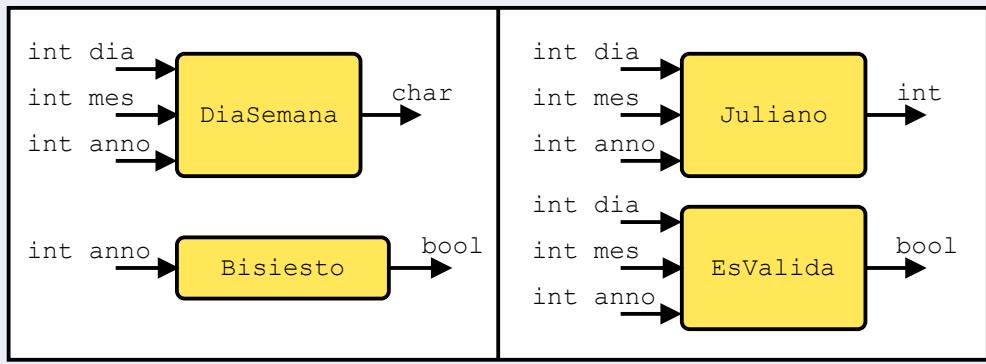
```

Los cálculos de este programa se modularizarán implementando las siguientes funciones:

- Función que reciba como entrada un año y que devuelva un booleano indicando si dicho año es o no bisiesto. Un año es bisiesto si verifica que es divisible por 4 y no por 100, excepto los divisibles por 400 que sí lo son.
- Función que reciba como entrada una fecha (día, mes y año) y que devuelva el día juliano astronómico con el que se corresponde. Use para ello la siguiente expresión^a

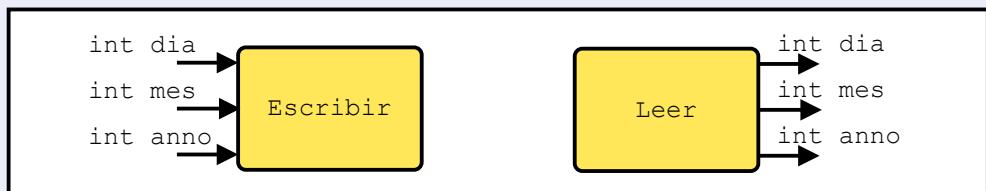
$$\begin{aligned} \text{Día juliano} = & (1461 * (\text{año} + 4800 + (\text{mes} - 14) / 12)) / 4 + \\ & (367 * (\text{mes} - 2 - 12 * ((\text{mes} - 14) / 12))) / 12 - \\ & (3 * ((\text{año} + 4900 + (\text{mes} - 14) / 12) / 100)) / 4 + \\ & \text{día} - 32075 \end{aligned}$$

- Una función que reciba una fecha (día, mes y año) y que devuelva una letra que codifique el día de la semana (L=Lunes, M=Martes, X=Miércoles, J=Jueves, V=Viernes, S=Sábado, D=Domingo). El día de la semana se calcula dividiendo el día juliano entre 7 y quedándose con el resto (operación módulo). Si el resto vale 0, el día es lunes, si vale 1 martes, y así sucesivamente hasta el 6 que se corresponde con el domingo.
- Una función que reciba una fecha (día, mes y año) y que devuelva un booleano indicando si se trata o no de una fecha válida.



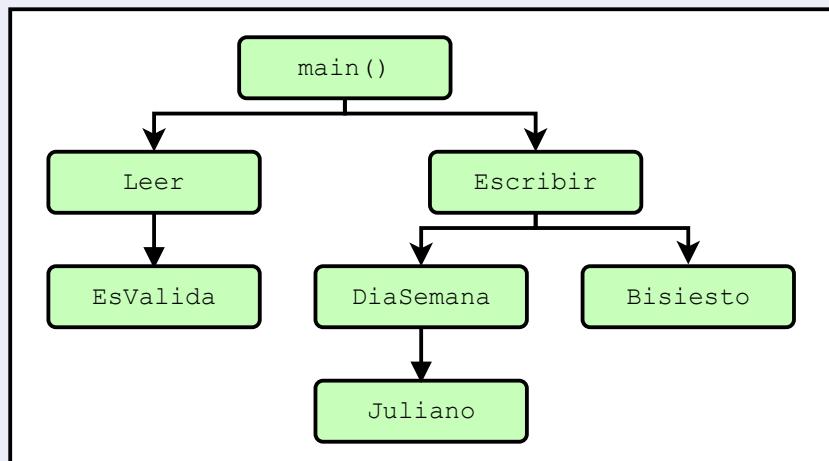
La E/S de este programa será tarea exclusiva de la función `main()` junto con las dos funciones que se piden a continuación. Ninguna otra función puede leer o escribir datos en la E/S estándar.

- Función que recibe como entrada una fecha y que la escribe con el formato expresado en el ejercicio previo.
- Función que lee una fecha desde la entrada estándar y que la devuelve mediante parámetros por referencia. Esta función deberá garantizar que la fecha leída es válida.



El programa comenzará pidiendo una fecha al usuario y a continuación la mostrará con el formato especificado. Puesto que el usuario puede escribir fechas sin sentido, se deberán validar los datos de entrada. En caso de que el usuario escriba una fecha incorrecta, deberá pedirla de nuevo.

Observe que ahora la función `main()` se dedica casi en exclusiva a llamar a otras funciones: una para leer la fecha con un formato y otra para escribirlo en el formato pedido. A continuación puede ver un esquema en el que se muestran las dependencias entre las distintas funciones que forman la solución. Estas dependencias muestran qué función hace uso de cuál otra:



^ahttp://www.hermetic.ch/cal_stud/jdn.htm

6.3.2 Diseño descendente (top-down)

En esta sección se pretende ilustrar el uso de funciones para modularizar programas. Dicha modularización se consigue como resultado de un *diseño descendente* (o *top-down*). El objetivo es construir programas complejos de una forma sencilla haciendo uso de la abstracción funcional.

Esta técnica consiste en diseñar programas pensando primero en soluciones abstractas para, poco a poco, ir concretándolas. En nuestro caso esto pasaría por comenzar a resolver los programas escribiendo la función `main()` de manera abstracta, es

decir, con llamadas a funciones que serán las encargadas de realizar las tareas concretas. Esta misma idea se aplicará a cada función que implementemos que a su vez se podrá descomponer en otras funciones.

Ejercicio 6.14 — Primos circulares. Debe construir una aplicación que pregunte un número al usuario y que le diga si es o no primo circular. Supongamos un número N formado por los dígitos $n_1n_2\dots n_k$. Decimos que un número es primo circular si son primos todos los números que se obtienen rotando los dígitos de N, es decir, si son primos los números formados por los dígitos:

$$\begin{aligned} &n_1n_2\dots n_k \\ &n_kn_1n_2\dots n_{k-1} \\ &n_{k-1}n_kn_1n_2\dots n_{k-2} \\ &\dots \\ &n_2n_3\dots n_{k-1}n_kn_1 \end{aligned}$$

Por ejemplo, el número 7937 es primo circular porque son primos los números: 7937, 7793, 3779 y 9377.

La aplicación pedirá datos ininterrumpidamente hasta que el usuario decida finalizar la ejecución. La entrada de datos sólo aceptará enteros positivos. A continuación vemos un ejemplo de ejecución:

```

Consola
Bienvenido/a
Dígame un número: 876123
No es primo circular
¿Desea continuar (S/s/N/n)? s
Dígame un número: -47653
El número debe ser un entero positivo: 7937
Es primo circular
¿Desea continuar (S/s/N/n)? S
Dígame un número: 1193
Es primo circular
¿Desea continuar (S/s/N/n)? N
Hasta la próxima

```

Para hacer el diseño descendente construirá el programa a partir de la función `main()` (es decir, el algoritmo principal). Esta función será bastante abstracta, de manera que cada paso será una tarea a realizar en una función independiente. Por ejemplo, podemos pensar en el siguiente algoritmo:

```

Dar la bienvenida
Repetir {
    Leer un entero positivo n
    Comprobar si n es primo circular
    Mostrar mensaje con resultado de la comprobación
} Mientras desee continuar
Despedirse

```

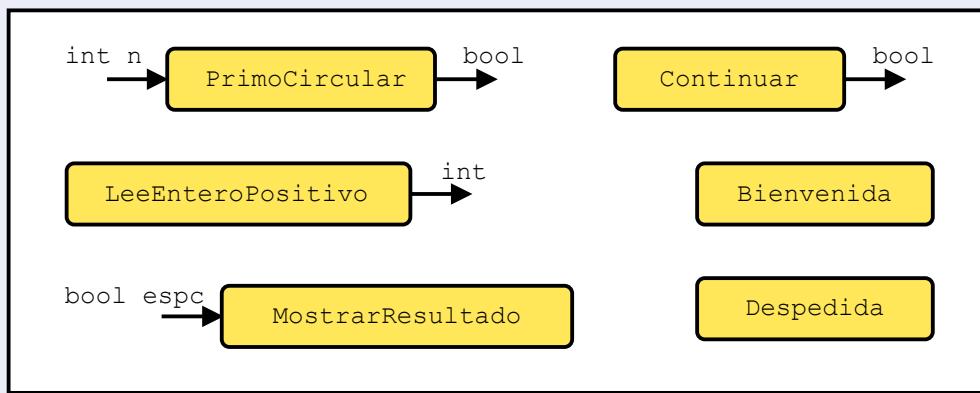
Que podemos reescribir en C++ así:

```

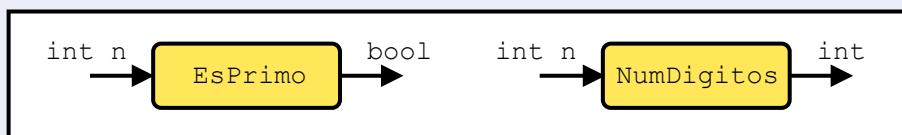
int main() {
    Bienvenida();
    do {
        int n = LeeEnteroPositivo();
        bool pcirc = PrimoCircular(n);
        MostrarResultado(pcirc);
    } while (Continuar());
    Despedida();
}

```

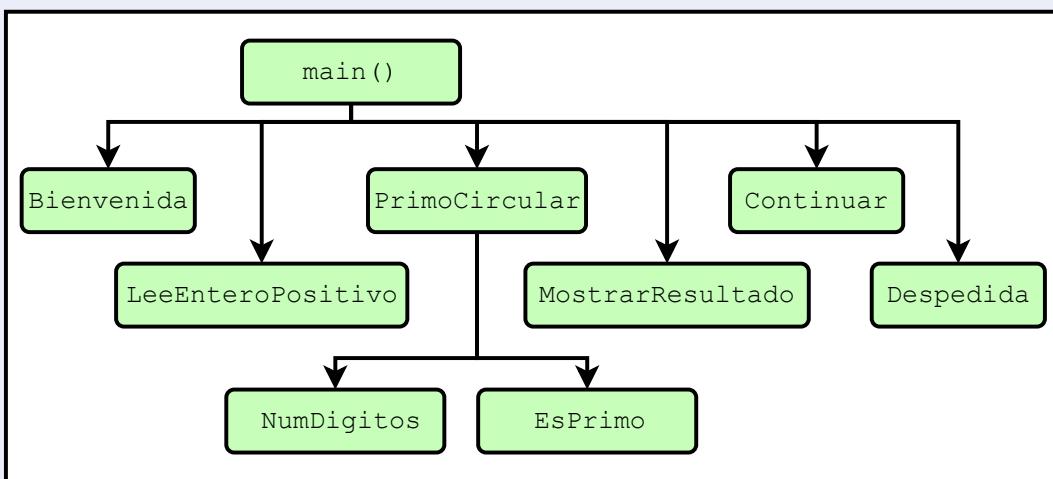
Observe que se mantiene el mismo nivel de abstracción del algoritmo pero usando la sintaxis propia de C++. Se hace una llamada a función para cada tarea del algoritmo y se usan algunas variables para establecer la comunicación entre funciones. De esta forma se obtienen los siguientes módulos:



En el caso del módulo *PrimoCircular()*, debe aplicar de nuevo un esquema de diseño descendente ya que es más complejo que los otros y debe realizar varias tareas que se pueden implementar de forma independiente. Dicho módulo debería descomponerse de manera que haga uso de, al menos, estos otros módulos:



Finalmente, el esquema con las relaciones entre módulos debería ser similar a este:



Ejercicio 6.15 — Números de Stirling de segunda clase. En combinatoria, los números de Stirling de segunda clase nos indican el número de formas de particionar un conjunto de n elementos en k subconjuntos no vacíos. La expresión que permite calcularlos es la siguiente:

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

La siguiente tabla muestra los primeros números de Stirling de segunda clase:

		k							
		0	1	2	3	4	5	6	7
n	0	1							
	1	0	1						
	2	0	1	1					
	3	0	1	3	1				
	4	0	1	7	6	1			
	5	0	1	15	25	10	1		
	6	0	1	31	90	65	15	1	
	7	0	1	63	301	350	140	21	1

Y a continuación tiene el código C++ para calcular un número de Stirling $\{n \text{ sobre } k\}$.

```
// Factorial de k
int fact_k = 1;
for (int i=2; i<=k; i++)
    fact_k *= i;

// Calculamos números de Stirling de segunda clase
int stir=0;
for (int j=0; j<=k; j++) {
    // Factorial de j
    int fact_j = 1;
    for (int i=2; i<=j; i++)
        fact_j *= i;

    // Factorial de k-j
    int fact_kj = 1;
    for (int i=2; i<=k-j; i++)
        fact_kj *= i;

    // Coeficiente binomial (k sobre j)
    int coefbin_kj = fact_k / (fact_j*fact_kj);

    // Potencia j elevado a n
    int pot_jn=1;
    for (int i=0; i<n; i++)
        pot_jn *= j;

    // Cálculo de (-1)^(k-j)
    int signo = ((k-j)%2==0) ? 1 : -1;

    // Acumulamos término en la sumatoria
    stir += signo*coefbin_kj*pot_jn;
}
stir /= fact_k;
```

Como puede apreciar, el programa es relativamente extenso y propenso a errores. Por ejemplo, sería relativamente fácil cambiar por error el nombre de alguna variable en alguna expresión. Además, la localización de ese tipo de errores puede llevarnos un tiempo considerable.

Implemente un programa que muestre una tabla como la anterior con los primeros números de Stirling tal que k y n sean menores que un cierto dato leído por la entrada estándar.

En este ejercicio debe modularizar el programa diseñando e implementando las funciones que se indican a continuación. Es importante que siga exactamente el orden propuesto para la construcción de estas funciones, de esta forma podrá apreciar mejor el potencial que ofrece el uso de la abstracción funcional en la construcción de programas.

1. Aplicación `main()` que calcula e imprime la tabla que se pide.
2. Cálculo del número de Stirling de 2^a clase n sobre k .
3. Cálculo del coeficiente binomial n sobre k .
4. Cálculo del factorial de un número n .
5. Cálculo de la potencia a elevado a b siendo a y b enteros. No utilice `pow()`.

Al seguir ese orden concreto de implementación se encontrará que debe usar –llamar– a las funciones antes de que estas hayan sido implementadas, por ejemplo, cuando esté con la función del segundo punto deberá utilizar las funciones de los puntos 3, 4 y 5 (pero que aún no han sido implementadas). Haga el ejercicio mental de suponer que ya están implementadas y utilícelas como si lo estuvieran realmente. Recuerde que lo único que debe conocer para usar una función es su prototipo.

Aunque al implementar una función esté suponiendo que ya existen otras, a la hora de escribir el código debe respetar la norma que dice que “una función debe ser –al menos– declarada antes de ser usada por primera vez”. Por ejemplo, la función que calcula el factorial debe aparecer en su programa antes de la función que calcula el coeficiente binomial ya que la segunda hace uso de la primera.

6.4 Gestión de errores

Hemos presentado las funciones como módulos que ofrecen una interfaz para realizar una tarea determinada, ocultando los detalles internos. Por tanto, la función se comporta como una *caja negra* de la que no sabemos nada (desde el punto de

vista del que la utiliza). Lo único que tenemos que conocer son los efectos de la llamada, incluyendo cuáles son los datos de entrada y cuáles los de salida.

Ahora bien, una función puede encontrarse con problemas en su ejecución, ya sea porque los datos que le ofrecemos son incorrectos o porque encuentra un error que impide su ejecución. Por ejemplo, podemos pedir a una función que calcule la raíz cuadrada de un número negativo (dato incorrecto) o que saque en la impresora cierto texto, y no pueda hacerlo al no detectar una impresora en el sistema (operación imposible).

Por tanto, es necesario que una función tenga un mecanismo para enfrentarse a estas situaciones, dando una solución al problema mediante una gestión adecuada del error.

6.4.1 Devolución de un valor de error

Cuando una función no consigue tener éxito, debería informar al código de llamada de alguna forma, y la primera opción que tenemos es devolver un valor que indique que ha habido un error. Una solución simple es devolver un entero que indique el resultado de la función, o un booleano indicando si ha habido éxito o no.

En principio, el entero resulta más interesante, ya que el booleano sólo puede informar de dos posibilidades -éxito o no- y el entero puede dar mucha más información al poder asignar múltiples valores -distintos tipos de error-. Cuando devolvemos un entero, podemos codificar el valor cero como indicativo de éxito y cualquier otro valor como indicativo de error. Note que al haber múltiples errores, podemos además asignar un tipo de error a cada uno de los valores que puede tomar el entero.

Precisamente, ya conocemos una función que tiene este comportamiento: la función `main`. Esta función devuelve un entero que indica si el programa ha terminado con éxito.

Habrá observado que en nuestros programas la función `main` no tiene una instrucción `return`. Esta situación es excepcional, puesto que es la única función que, aún devolviendo un valor, puede prescindir de esta instrucción. Podemos pensar que cuando el programa llega al final de `main` y no ha encontrado la sentencia `return`, añade por defecto `return 0;`.

El hecho de que `main` devuelva un valor se debe a que es necesario que los programas tengan una forma de indicar si la terminación ha sido exitosa o no. Note que la mayoría de los programas que se lanzan en un sistema son ejecutados por otros programas, y para que éstos funcionen, necesitan saber que las tareas se han realizado correctamente.

Ejercicio 6.16 — Sistema de dos ecuaciones y dos incógnitas. Considere el siguiente sistema de 2 ecuaciones con 2 incógnitas, así como sus correspondientes soluciones:

$$\begin{aligned} ax + by = c \\ dx + ey = f \end{aligned} \quad x = \frac{ce - bf}{ae - bd} \quad y = \frac{af - cd}{ae - bd}$$

El sistema es compatible determinado (tiene una solución) si $ae - bd \neq 0$ mientras que tiene cero o infinitas soluciones en caso contrario.

Escriba un programa para obtener la solución de un sistema compatible determinado. Para ello, deberá leer los 6 valores que determinan el problema y obtener por pantalla el correspondiente par de valores (x,y) -solución del sistema- o un mensaje indicando que no es compatible determinado. Para ello, defina una función que, a partir del sistema, obtiene las soluciones. Además, esta función devuelve un entero que indica si ha habido algún error, ya que es posible que el sistema no tenga una solución.

6.4.2 Errores graves irrecuperables

Cuando un programa se encuentra con un error irrecuperable, es posible que la única salida sea terminarlo directamente, sin cerrar tareas pendientes ni recuperar ningún tipo de información. La idea, por tanto, sería que el programa acabe devolviendo un error distinto de cero, es decir, que `main` devuelva un número distinto de cero y termine el programa.

El problema es que tal vez estemos en una función que no sea `main`, y que deseemos romper la ejecución del programa sin necesidad de volver a ella. Para resolver esta situación, C++ ofrece una función de la biblioteca estándar (disponible incluyendo `cstdlib`) que tiene la siguiente cabecera:

```
void exit(int estado);
```

que provoca la terminación directa del programa, devolviendo el entero `estado` como resultado del programa, es decir, como el valor que devuelve `main` y que indica el estado de finalización del programa.

Lógicamente, es una forma de terminación anormal, y sólo se debería usar en casos extremos en los que la situación sea inmanejable y no haya más remedio que terminar. Por ejemplo, imagine que el sistema se queda sin memoria, en este caso, no hay mucho que hacer, ya que incluso para terminar de otra forma necesitaríamos tener memoria (por ejemplo para realizar una tarea de aviso de terminación al usuario). Por tanto, evitaremos su uso.

6.4.3 Precondiciones y postcondiciones

Cuando diseñamos una función, especificamos la forma en que debería llamarse y los efectos que tiene dicha llamada. Por ejemplo, en el caso de la función `sqrt`, se supone que vamos a pasarle un número positivo, ya que no está definida para números negativos.

Llamamos *precondiciones* a las condiciones que tienen que cumplirse antes de la llamada a la función, y *postcondiciones* a las que se cumplirán si hemos llamado a la función con dichas precondiciones. Por tanto, podemos decir que una precondición de la función `sqrt` es que el número que le pasamos no es negativo, y una postcondición es que el valor devuelto es igual a la raíz cuadrada del parámetro de entrada.

El establecimiento de precondiciones es una forma simple de evitar errores. Cuando establecemos una precondición para llamar a una función, estamos diciendo que el que use la función debe asegurarse de que esa condición se cumple antes de la llamada. Por otro lado, el que implementa la función, debe asegurarse de que, si se han garantizado las precondiciones, las postcondiciones también se cumplirán.

Ahora bien, ¿qué ocurre si no se cumplen las precondiciones? En este caso, no hay ninguna garantía de que la función obtenga un resultado válido, por lo que el que diseña los detalles de la función será el responsable de decidir qué comportamiento tiene en este caso.

Lógicamente, es de esperar que el que usa la función garantice esas condiciones, por lo que el programador puede suponer que se cumplen y no comprobar nada. En consecuencia, si no se cumplen las precondiciones al efectuar la llamada, los efectos pueden ser desastrosos, aunque el responsable no ha sido el que implementó la función, sino el que la llamó sin asegurarse de que se cumplían dichas condiciones.

Ejercicio 6.17 — Precondiciones en la resolución del sistema de dos ecuaciones. Considere el ejercicio 6.16, donde se resuelve un sistema de dos ecuaciones con dos incógnitas. Vuelva a escribir el programa rediseñando el módulo que resuelve el sistema de ecuaciones. En este caso, se establece como precondición que el sistema sea compatible determinado. Note que la función que resuelve el sistema no tiene que devolver ningún valor de error.

Debería añadir una nueva función que tiene como parámetros el sistema de ecuaciones y devuelve si es compatible determinado. Esta función no se debe llamar desde la que resuelve el sistema sino antes de llamarla para asegurarse de que se cumple la precondición.

Control de precondiciones con assert

Una vez establecidas las precondiciones de una función, debemos asegurarnos de llamarla asegurándonos de que se cumplen. Por lo tanto, dentro de la función, no es necesario comprobar que la llamada ha sido correcta.

Suponga que hemos terminado un programa y está en explotación. Los usuarios lo van a ejecutar sin ningún error, pues nos hemos asegurado de que funciona perfectamente. En este caso, las precondiciones de las funciones deben estar cumpliéndose en todos los casos. Si incorporamos código dentro de la función para ver si la condición se cumple o no, estaríamos perdiendo tiempo, pues sabemos que el programa es correcto y, por tanto, todas se cumplen.

Sin embargo, durante el desarrollo del programa, podemos equivocarnos en las llamadas a las funciones. Aunque sepamos que una condición se debe cumplir, es probable que nos olvidemos, o por culpa de otros errores arrastrados, la condición no se cumpla. El hecho de no comprobarla puede hacer que la depuración sea mucho más complicada.

Por ejemplo, suponga que hacemos un programa y nada más empezar -en la primera función- introducimos datos erróneos, aunque el programa sigue ejecutándose hasta el final. Tal vez, cuando veamos los resultados finales sepamos que algo ha ido mal, pero no sabemos dónde encontrar ese error, pues ha podido producirse en cualquier punto. Sería mejor haber parado el programa nada más detectar el fallo en la primera función.

Desde este punto de vista puede parecer recomendable comprobar las precondiciones. Para resolver este problema, el lenguaje nos ofrece la posibilidad de establecer *aserciones* con la macro `assert`, que está disponible si incluimos el fichero de cabecera `cassert`. Tiene un único parámetro, que es una condición que debe cumplirse, es decir, tiene un parámetro de tipo booleano.

En caso de que dicha condición evalúe a `true`, no pasa nada, sigue el programa normalmente. Si por el contrario, su valor fuese `false`, el programa se interrumpe y se genera un error de ejecución. El mensaje generado incluye información sobre el fichero y línea de código que contiene la condición que ha fallado. Por tanto, es una herramienta que se ofrece para la depuración, para cuando el programador está probando sus programas y necesita saber si hay algún error en el momento de detectarlo. Por supuesto, un programa definitivo, que se ofrece al usuario para su explotación, no debería mostrar ninguno de estos mensajes⁵.

Ejercicio 6.18 — Ecuación de primer grado. Escriba un programa para resolver una ecuación de primer grado de la forma $ax + b = 0$ donde a y b son dos números enteros. Para ello, deberá crear una función que reciba esos dos valores y que devuelva la solución de la ecuación. Establezca una precondición e incluya una línea `assert` para comprobarla al entrar en la función. Ejecute el programa para ver el efecto de romper la condición, y finalmente, modifíquelo para que su ejecución sea siempre correcta.

⁵De hecho, el compilador ofrece la posibilidad de eliminar todas las aserciones de un programa añadiendo, simplemente, una opción de compilación.

6.5 Valores por defecto y sobrecarga

Para finalizar este tema es interesante añadir algunos contenidos adicionales sobre funciones. Aunque en principio puedan parecer cuestiones muy simples que sólo facilitan y simplifican la escritura de funciones, más adelante las usaremos de manera más intensiva e incluso serán fundamentales para desarrollar otras herramientas del lenguaje.

6.5.1 Parámetros con valores por defecto

En C++ podemos definir funciones que tengan parámetros con *valores por defecto*. Básicamente, se trata de parámetros que queremos que sean opcionales, es decir, que el usuario puede o no especificar en la llamada. En caso de que el correspondiente argumento no aparezca, el programa asumirá que le hemos dado cierto valor (por defecto).

Por ejemplo, podemos crear una función *ImprimirTabla* que muestra la tabla de multiplicar de cierto entero. La función es muy simple de programar, ya que sólo tiene un parámetro entero, que pasa por valor, y muestra la tabla en consola:

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

void ImprimirTabla(int n)
{
    for (int i=1; i<=10; ++i)
        cout << n << " x " << i << " = " << n*i << endl;
}

int main()
{
    int n;

    cout << "Indique la tabla que desea imprimir: ";
    cin >> n;
    ImprimirTabla(n);
}
```

Lo normal será que la tabla muestre los productos del 1 al 10. Pero, en un momento dado, podríamos necesitar más o menos productos (por ejemplo: del 1 al 7 o del 1 al 25). Para resolver este problema, podemos cambiar la función que imprime la tabla, proponiendo la siguiente:

```
void ImprimirTabla(int n, int maximo=10)
{
    for (int i=1; i<=maximo; ++i)
        cout << n << " x " << i << " = " << n*i << endl;
}
```

Observe que hemos modificado la cabecera de la función para que también acepte un parámetro que indica el número de entradas que tiene que imprimir el bucle. Así, podemos hacer una llamada *ImprimirTabla(5, 100)* que imprimiría la tabla del 5, pero con 100 entradas.

Ahora bien, hemos añadido en la cabecera que el segundo parámetro tiene el valor por defecto 10. Por tanto, si llamamos a *ImprimirTabla(5)*, donde no hemos especificado el segundo parámetro, imprimirá la tabla como si hubiéramos hecho *ImprimirTabla(5, 10)*. El valor por defecto es útil cuando, aun siendo lo habitual usar un valor determinado, se puede dar el caso en que necesitemos usar un valor distinto al habitual.

Un aspecto interesante con respecto a este diseño es que la modificación de la función no afecta de ninguna forma a cualquier programa anterior que haya hecho uso de la primera versión. Efectivamente, como sigue siendo válido llamar con un solo parámetro, y su comportamiento es idéntico a la primera versión, cualquier programa que hayamos desarrollado con anterioridad sigue siendo compilable y funcionará dando los mismos resultados.

De hecho, para hacer funcionar el *main* que hemos listado, donde pedimos un número y mostramos la tabla hasta el 10, no tenemos que hacer ningún cambio.

Finalmente, es necesario indicar que para evitar ambigüedades, los parámetros por defecto deben ser los últimos de la función. En caso de que demos menos parámetros, se supone que se usan los valores por defecto de los últimos.

6.5.2 Sobre carga de funciones

Cuando definimos distintas funciones, usamos distintos nombres para distinguirlas. Lógicamente, el compilador distingue estos nombres y enlaza cada llamada a la correspondiente definición de función. Ahora bien, el compilador caracteriza cada función no sólo con el nombre, sino también con el número y tipo de parámetros que tiene.

Por tanto, en nuestros programas será lícito añadir tantas funciones como queramos con el mismo nombre, siempre y cuando el número de parámetros y sus tipos permitan distinguir unas de otras. Diremos que estamos *sobre cargando* dichas funciones.

Ejercicio 6.19 — Sobre carga de la función de ordenar números. Considere el ejercicio 6.11, donde han usado un par de funciones de ordenación (para 2 y para 3 datos). Modifique el programa de manera que las dos funciones tengan el mismo nombre *Ordenar*, y compruebe que sigue funcionando correctamente.

Cuando sobre cargamos una función, el compilador es el encargado de resolver la correspondencia entre llamadas y definiciones. Aunque no vamos a entrar en detalles sobre el mecanismo de resolución de llamadas, es importante tener en cuenta que hay situaciones en las que el compilador no es capaz de escoger una, ya que no está clara la mejor opción.

Intuitivamente, si dos funciones tienen distintos parámetros, sabemos que se seleccionará aquella función cuyos tipos de datos coincidan con los de la llamada. Sin embargo, tenemos que tener en cuenta que podemos llamar a las funciones con tipos que no sean idénticos a los parámetros formales, es decir, que requieren de conversiones implícitas.

Por ejemplo, supongamos que creamos una función para calcular el cuadrado de un número real. Para tener varias versiones de precisión, lo hacemos para los tres tipos:

```
float Cuadrado(float a) { return a*a; }
double Cuadrado(double a) { return a*a; }
long double Cuadrado(long double a) { return a*a; }
```

Sin embargo, podemos realizar una llamada con un parámetro de tipo entero. En este caso, el compilador no sabría decidir qué función debe ejecutar ya que tiene sentido la conversión entre entero y cualquier de los tres tipos reales (**float**, **double** y **long double**), siendo válidas las tres llamadas.

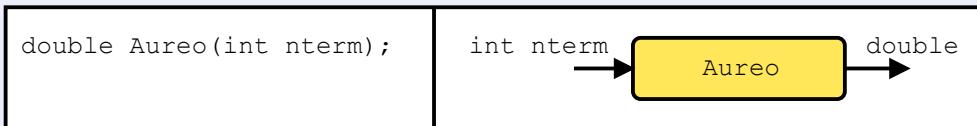
Ejercicio 6.20 — Ambigüedad en la llamada con entero. Escriba un programa que incluya las tres funciones *Cuadrado* anteriores y que pruebe a hacer una llamada -en el programa principal- pasándole un objeto de tipo entero.

Otro ejemplo interesante para ilustrar errores en la llamada a funciones sobrecargadas lo tenemos si creamos varias versiones con parámetros por referencia. Como hemos indicado anteriormente, ahora mismo no estamos considerando que haya conversiones cuando llamamos a funciones con este tipo de paso de parámetros, por lo que el error no sería de la misma naturaleza de los anteriores.

Ejercicio 6.21 — Error en la llamada a función sobrecargada. Escriba un programa que incluya dos funciones *Intercambiar* como la propuesta en el ejercicio 6.9, una para el tipo **int** y otra para el tipo **double**. Declare dos variables, de tipo **int** y **double**, en la función **main** y escriba una llamada a la función *Intercambiar* con ambos objetos. ¿Qué error obtiene? ¿Es igual que en el ejercicio anterior?

6.6 Ejercicios adicionales

Ejercicio 6.22 — Número áureo. Implemente una función que calcule el número áureo^a sabiendo que puede obtenerse como la fracción entre dos términos consecutivos de la serie de *Fibonacci*^b. La función recibirá como argumento el número de términos de la serie de Fibonacci que se deben calcular. La función devolverá el valor de la aproximación para ese número de términos.



^ahttp://es.wikipedia.org/wiki/Número_Áureo

^bhttp://es.wikipedia.org/wiki/Sucesión_de_Fibonacci

Ejercicio 6.23 — Cálculo de π . Diseñe e implemente dos funciones para calcular el valor de π de acuerdo a los siguientes métodos iterativos:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots \quad (\text{Fórmula de Leibniz})$$

$$\pi = \sum_{n=0}^{\infty} \frac{2(-1)^n 3^{(\frac{1}{2}-n)}}{2n+1} \quad (\text{Desarrollo en serie})$$

Haga un programa que ejecute estas funciones y que evalúe cómo de rápido converge cada método.

Ejercicio 6.24 — Días para el cumpleaños. Haga un programa que lea dos fechas: la fecha actual y su fecha de nacimiento. A continuación mostrará cuántos días faltan para su próximo cumpleaños. Deberá reutilizar las funciones para leer fechas y para calcular el día juliano del ejercicio 6.13. A continuación tiene varios ejemplos de ejecución:

Consola

```
Dígame la fecha actual: 2 8 2010
Dígame su fecha de nacimiento: 4 8 2000
Faltan 2 días para su cumpleaños
```

Consola

```
Dígame la fecha actual: 4 8 2010
Dígame su fecha de nacimiento: 4 8 2000
Hoy es su cumpleaños
```

Consola

```
Dígame la fecha actual: 10 8 2010
Dígame su fecha de nacimiento: 4 8 2000
Faltan 359 días para su cumpleaños
```

Ejercicio 6.25 — Números perfectos y amigos. Realice una aplicación que compruebe algunas propiedades curiosas de los números. En particular, dados dos números enteros debe comprobar:

- Si ambos números son *amigos*. Dos números a y b se dicen amigos si a es la suma de los divisores propios de b y b es la suma de los divisores propios de a . Los divisores propios de un número n son todos sus divisores, incluido el uno, salvo el propio n .
- Si cada uno de ellos se considera un número *perfecto*. Un número se dice perfecto si es amigo de sí mismo.

La aplicación principal podría ser similar a la siguiente:

```
int main() {
    Bienvenida();
    do {
        int a = LeeEnteroPositivo();
        int b = LeeEnteroPositivo();
        bool perfa = EsPerfecto(a);
        bool perfb = EsPerfecto(b);
        bool amigos = SonAmigos(a,b);
        MostrarResultadoPerfecto(perfa);
        MostrarResultadoPerfecto(perfb);
        MostrarResultadoAmigos(amigos);
    } while (Continuar());
    Despedida();
}
```

Para el desarrollo de *EsPerfecto* y *SonAmigos* debe aplicar un esquema de diseño descendente. Dicho esquema incluirá una función que calcule la suma de los divisores propios de un número n (divisores menores estrictos que n incluyendo al 1).

Ejercicio 6.26 — Números sociables. Una *sucesión alícuota* es una sucesión numérica en la que cada término es la suma de los divisores propios del término anterior. Por ejemplo, considere la siguiente sucesión:

87262 69410 67102 47954 23980 31460 46744 40916 32416 31466...

Los divisores propios de 87262 (el primer número) son:

1 2 7 14 23 46 161 271 322 542 1897 3794 6233 12466 43631

cuya suma es el siguiente término (69410). Si volvemos a repetir el cálculo con este número obtendríamos el siguiente término (67102) y así sucesivamente.

Se dice que un número es *sociable* si su sucesión alícuota es cíclica. Por ejemplo, la sucesión alícuota del número 1264460 es:

1264460 1547860 1727636 1305184 1264460...

que como podemos observar es circular (se repite a partir del 5º término cíclicamente).

Implemente una aplicación que determine si un número es o no sociable siguiendo el esquema de diseño descendente (y reutilizando parte del código de ejercicios anteriores si lo considera necesario).

Observe que pueden ocurrir situaciones como las siguientes:

- Si un número no es sociable su sucesión alícuota no es cíclica y, por tanto, infinita.
- Si los números con los que se hacen los cálculos son demasiado grandes se produzca un desbordamiento.
- Aun siendo sociable, puede ocurrir que antes de encontrar el ciclo de la sucesión, se produzca un desbordamiento.
- Que el número sea sociable, pero el ciclo sea tan largo que se necesite mucho tiempo para encontrar el ciclo.

El programa debe controlar estas situaciones y establecer algún criterio para evitar, al menos, ejecuciones de tiempo infinito.

7

Funciones con cadenas y vectores

Introducción.....	65
Ejercicios de funciones y vectores.....	65
Ejercicios de funciones y cadenas.....	67
Ejercicios adicionales.....	68

7.1 Introducción

Los temas de funciones, cadenas y vectores son en sí bastante amplios como para trabajar cada una de esas utilidades de forma independiente. Sin embargo, es conveniente incorporar el actual tema porque:

- El diseño de soluciones para problemas más complejos incluye tanto la selección y definición de estructuras de datos como el desarrollo de algoritmos. Disponer de los tipos **string** y **vector** ya nos ofrece muchas posibilidades para resolver problemas de cierta entidad. Es conveniente que el estudiante se enfrente a problemas que conducen a soluciones de mayor tamaño. Para ello, debe saber usar de forma combinada estas herramientas.
- Es recomendable estudiar los tres temas –funciones, cadenas y vectores– de forma independiente. Además, se ha diseñado un recorrido sobre éstos que permite abordarlos en un orden cualquiera. El objetivo de esta exposición es que el estudiante pueda trabajar con mayor intensidad con cuaquiera de ellos, sin la dificultad añadida de tener que conocer otros aspectos que puedan trabajarse de forma independiente.

En el tema que ahora presentamos, proponemos ejercicios para los que el estudiante tendrá que crear soluciones que pueden necesitar funciones, cadenas y/o vectores.

7.2 Ejercicios de funciones y vectores

La relación entre vectores y funciones no tiene, en principio, nada de especial. Como cualquier tipo simple, un vector se puede pasar a una función, por valor o por referencia, y puede ser devuelto por una función, con un comportamiento similar.

La novedad en el caso de los vectores es que ahora sabemos que el objeto que pasamos o devolvemos de una función puede llegar a ser muy grande. En el caso de los tipos simples, si pasamos un objeto que no se va a modificar –sólo es un dato de entrada– lo pasamos por valor, mientras que si se va a modificar, lo pasamos por referencia. En el caso de los vectores podemos aplicar el mismo criterio cuando decidimos si un dato debe pasar por valor o por referencia.

Sin embargo, sabemos que si lo pasamos por valor, la función deberá crear una copia del original para que los cambios al parámetro no le afecten. Esta copia es una operación costosa, por lo que es tentador pasarlo por referencia, para usar el original y ahorrarnos dicha copia. El problema es que el paso por referencia indica que se va a modificar el vector original, cuando no es nuestra intención. Una nueva posibilidad –que usaremos en este caso– será pasar el parámetro por *referencia constante*. Consiste en pasarlo por referencia, pero añadiendo la palabra **const** para que el compilador sepa que no se va a modificar el objeto. Un ejemplo de su uso es el siguiente:

```
#include <iostream>
#include <vector> // Para poder usar vectores
using namespace std;

void Escribir(const vector<int>& vec)
{
    for (int i=0; i<vec.size(); ++i)
        cout << vec[i] << endl;
}

int main()
{
    vector<int> v(10,1);
    Escribir(v);
}
```

Al usar el parámetro `vec` dentro de la función, estamos usando el original `v`, ya que lo hemos pasado por referencia, como indica la cabecera de la función. Sin embargo, el compilador sabe que `vec` sólo se usará para consultar, no para modificar, ya que hemos añadido la palabra `const`. De hecho, si intentamos modificar el vector dentro de la función, el compilador generará un error de compilación al haber intentado cambiar un objeto que no se va a modificar.

Ejercicio 7.1 — Media, desviación y varianza de un vector de números. Modifique el programa del ejercicio 4.4 de forma que modularice el código mediante las siguientes funciones:

- Leer los datos. No recibe ningún parámetro y devuelve un vector con los datos leídos desde la entrada estándar. Los datos son una secuencia de números que termina en un número negativo que actúa como centinela.
- Calcular la media. Recibe un vector de números reales y devuelve la media de dichos valores.
- Calcular la desviación. Recibe un vector de datos y un valor (por ejemplo, la media) y devuelve la desviación media de los datos a dicho valor.
- Calcular la varianza. Recibe un vector de datos y devuelve la varianza.

De la misma forma en que el paso por valor es costoso, también la devolución de un vector puede consumir muchos recursos¹. Por ejemplo, copiando el vector al punto de llamada para poder usar dicho resultado en alguna operación. Por ello, es posible usar el paso por referencia como un sistema para obtener el resultado evitando esa copia. La idea es pasar, por referencia, un vector donde se obtendrán los resultados.

Ejercicio 7.2 — Matrices de Vandermonde. Una matriz de *Vandermonde* cuadrada de tamaño n es una matriz que tiene la siguiente forma:

$$\begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \alpha_n^2 & \cdots & \alpha_n^{n-1} \end{pmatrix}$$

Haga un programa que lea desde la entrada estándar un valor n , y una serie de valores enteros $\alpha_1, \alpha_2, \dots, \alpha_n$ y que genere la correspondiente matriz de Vandermonde. Debe incluir, al menos, las siguientes funciones:

- Función que lee un vector de enteros desde la entrada estándar. Esta función será de tipo `void` y devolverá el vector mediante un paso por referencia.
- Función que recibe un vector de enteros y que devuelve la matriz de Vandermonde asociada. Esta función será de tipo `void` y devolverá la matriz mediante un paso por referencia.
- Función que presente adecuadamente una matriz de enteros en la salida estándar.

Ejercicio 7.3 — Factorización de un número. Implemente un programa que lea una serie de números enteros desde la entrada estándar y que muestre al usuario su factorización. El programa estará pidiendo números y mostrando su factorización hasta que el usuario deseé terminar. Dicho programa incluirá, al menos, las siguientes funciones:

- Función para calcular los primos menores que un número dado mediante el algoritmo de la *Criba de Eratóstenes* (que ya resolvió en el ejercicio 4.6).
- Función que, dado un valor entero, devuelva un vector con su descomposición en factores primos. Esta función hará uso del resultado de la criba para ir probando la divisibilidad del número.

A continuación se muestra un ejemplo de ejecución:

```
Consola
Escriba un valor entero: 1500
Su descomposición factorial es: 2 2 3 5 5 5
```

Observe que la intención es que la segunda función haga uso de los resultados de la criba por lo que tiene dos posibilidades:

- Crear el vector de números primos dentro de la segunda función. Cada vez que se descomponga un número se calculan de nuevo todos los primos menores que él.
- Crear el vector de primos fuera de la segunda función, de manera que sólo es necesario crearlo una vez. Por contra, no se sabe a priori cuál es el mayor primo que se va a necesitar.

Estudie y razonne cuáles son las ventajas e inconvenientes de cada una de las propuestas e implemente la que usted crea que es más eficiente.

¹Al menos en el estándar del 98. Realmente en el último estándar de C++, ya se ha mejorado esta situación para que se pueda realizar de forma más eficiente.

Ejercicio 7.4 — Polinomios. Implemente un programa que permita realizar las siguientes operaciones básicas sobre polinomios de cualquier grado:

- Evaluar un polinomio en un punto.
 - Derivar un polinomio.
 - Evaluar la derivada de un polinomio en un punto.
 - Calcular el valor máximo de un polinomio en un intervalo. Este algoritmo deberá evaluar el polinomio en varios puntos del intervalo y seleccionar aquel que lo maximice. Los puntos en los que va a realizar estarán dentro del intervalo y serán equidistantes entre sí. El número de puntos para muestrear será dado por el usuario del programa.
- Para trabajar con polinomios de la forma $a_nx^n + a_{n-1}x^{n-1} + \dots + a_2x^2 + a_1x + a_0$ almacenaremos sus coeficientes en vectores de números reales. Se piden, al menos, las siguientes funciones:
- Función para leer un polinomio desde la entrada estándar.
 - Función para escribir un polinomio en la salida estándar.
 - Función para evaluar un polinomio en un punto.
 - Función para, dado un polinomio, calcular y devolver su derivada.
 - Función para, dado un polinomio, un intervalo y un número de puntos a muestrear en ese intervalo, calcular y devolver su máximo de acuerdo al algoritmo indicado antes.
 - Función para, dado un polinomio, un intervalo y una cota de error máximo, calcular una raíz del polinomio en ese intervalo mediante el algoritmo de bisección (ya resuelto en el ejercicio 3.9).

El programa mostrará un menú de opciones al usuario para decidir en cada momento qué operación se desea realizar. La ejecución continuará hasta que el usuario decida finalizar seleccionando una de las opciones del menú.

Ejercicio 7.5 — Mediana de un vector. Considere de nuevo el ejercicio 4.16 y modularícelo adecuadamente. Deberá incluir funciones al menos para:

- Leer un vector de números enteros.
- Ordenar un vector de números enteros. Puede usar el algoritmo que considere oportuno.
- Calcular la mediana de un vector de números enteros.

Ejercicio 7.6 — Máximo de una matriz. Implemente una función que calcule el mayor elemento de una matriz de números reales.

Ejercicio 7.7 — Medias de las filas de una matriz. Haciendo uso de una de las funciones del ejercicio 7.1 realice una función que reciba como dato una matriz de reales y que devuelva un vector con las medias de cada fila.

Ejercicio 7.8 — Determinante de Vandermonde. En el ejercicio 7.2 se ha explicado qué forma tiene una matriz de Vandermonde. Realice un programa que lea una matriz cualquiera desde la entrada estándar y que compruebe si es o no de Vandermonde. En caso de serlo deberá calcular y mostrar su determinante. Para ello deberá implementar, al menos, las siguientes funciones:

- Función que lee una matriz de enteros desde la entrada estándar.
- Función que recibe como entrada una matriz y devuelve un dato indicando si es o no de Vandemonde.
- Función que recibe una matriz y, en caso de ser de Vandermonde devuelve su determinante. En caso de no ser de Vandermonde deberá devolver un dato que así lo indique.

El determinante de una matriz de Vandermonde se calcula como:

$$V_n = \prod_{1 \leq i < j \leq n} (\alpha_j - \alpha_i)$$

Por ejemplo, en una matriz de tamaño 4 generada a partir de los valores $\alpha_1, \alpha_2, \alpha_3, \alpha_4$, el determinante sería $(\alpha_4 - \alpha_3)(\alpha_4 - \alpha_2)(\alpha_4 - \alpha_1)(\alpha_3 - \alpha_2)(\alpha_3 - \alpha_1)(\alpha_2 - \alpha_1)$

7.3 Ejercicios de funciones y cadenas

Las consideraciones que podemos hacer sobre el uso de funciones que trabajan con cadenas son las mismas que las que se han hecho con los vectores:

- Una cadena puede ser pasada como parámetro por valor o por referencia.
- Una cadena puede ser devuelta (`return`) por una función.
- El paso por valor de cadenas es una operación que puede resultar ineficiente por lo que se recomienda el paso como `const &`.

Ejercicio 7.9 — Palíndromo. Implemente una función que reciba una cadena de caracteres como entrada y que, de forma eficiente, devuelva un dato indicando si es o no un palíndromo. La cadena de entrada podría contener espacios en blanco, signos de puntuación (salvo tildes) y mayúsculas y minúsculas. Tenga en cuenta que para que sea eficiente no debería seguir ninguna de estas estrategias:

- Calcular la cadena invertida en otra y luego compararlas. Se necesita memoria adicional.
- Usar la función `erase` para ir borrando los caracteres no alfabéticos. Este algoritmo es ineficiente.

Ejercicio 7.10 — Conversión a mayúsculas. Implemente una función que reciba como argumento una cadena y que la modifique cambiando todas las letras que contiene por sus correspondientes mayúsculas. El resto de caracteres se quedarán igual.

Ejercicio 7.11 — Búsqueda de una subcadena. Suponga que no dispone de la operación `find` para buscar cadenas. Implemente una función que realice esa misma tarea.

Ejercicio 7.12 — Separación de palabras. Implemente una función que reciba como entrada una cadena de caracteres y que devuelva un vector de cadenas tal que cada elemento de ese vector sea una palabra de la cadena de entrada. Se saltarán los caracteres no alfabéticos. Por ejemplo, si la cadena de entrada es “Hola, ¿como estas?” el vector de salida contendrá las cadenas “Hola”, “como” y “estas”. Observe que no se considerarán las letras con tilde.

Ejercicio 7.13 — Ordenación alfabética. Implemente un programa que lea nombres y apellidos de una serie de personas hasta que el usuario introduzca un carácter especial de fin de entrada de datos (por ejemplo ‘#’). El programa mostrará a continuación dos listados, uno con la ordenación ascendente por nombres y otro por apellidos. Vemos un ejemplo de ejecución:

```

Consola
Personas: Persona 1, nombre: Antonio
          Persona 1, apellidos: Perez Martin
          Persona 2, nombre: Javier
          Persona 2, apellidos: Gonzalez Rodriguez
          Persona 3, nombre: Elena
          Persona 3, apellidos: Arias Perez
          Persona 4, nombre: Ana
          Persona 4, apellidos: Rodriguez Garrido
          Persona 5, nombre: #

La ordenación por nombre es:
          Ana Rodriguez Garrido
          Antonio Perez Martin
          Elena Arias Perez
          Javier Gonzalez Rodriguez

La ordenación por apellidos es:
          Elena Arias Perez
          Javier Gonzalez Rodriguez
          Antonio Perez Martin
          Ana Rodriguez Garrido
  
```

Almacenará por separado los nombres y los apellidos en sendos vectores de cadenas. Tenga en cuenta que aunque sean vectores independientes, el elemento de la posición i del primer vector (nombre) contiene información asociada al elemento de la posición i del segundo vector (apellidos). Esto significa que si cambiamos de posición un elemento en el primer vector también hemos de cambiar su correspondiente del segundo vector.

Deberá modularizar convenientemente el código de este programa.

7.4 Ejercicios adicionales

Ejercicio 7.14 — Tres en raya. Se desea implementar el juego del tres en raya para dos personas. Es un juego por turnos en el que cada jugador coloca fichas de un determinado tipo en un tablero de tamaño 3x3 (por ejemplo: un jugador coloca *círculos* y otro coloca *equis*). El primer jugador que consiga colocar tres fichas de su tipo alineadas gana. Las fichas no pueden moverse una vez colocadas y sólo es posible poner nuevas fichas en las casillas que quedan vacías. Puede ocurrir que se completen todas las casillas del tablero sin que ninguno haya logrado el tres en raya, en ese caso habrá empate.

Seguiremos una estrategia de diseño descendente: comenzaremos escribiendo la función `main()` que implementa la lógica del juego:

- Limpiar el tablero
- Mostrar el tablero
- Inicializar turno
- Repetir mientras no acabe el juego:
 - Avanzar al siguiente turno
 - Preguntar al jugador del turno en curso dónde quiere colocar su ficha
 - Poner la ficha en el tablero
 - Mostrar tablero resultante
 - Comprobar el estado del juego: "gana alguien", "empate" o "seguimos jugando"
- Mostrar el ganador (o empate si lo hubiera).

El programa, además, permitirá que juguemos tantas partidas como deseemos. La función `main()` debe ser la siguiente:

```
int main() {
    vector<vector<char>> tab(3, vector<char>(3));
    int turno=0, estado, f, c;
    do {
        cout << "Comienza la partida" << endl;
        VaciarTablero(tab);
        VisualizarTablero(tab);
        do {
            turno = (turno+1)%2;           // Avanzar turno
            PedirJugada(turno,tab,f,c);   // Pregunta dónde poner ficha
            PonerFicha(turno,tab,f,c);    // Poner ficha en tablero
            VisualizarTablero(tab);       // Mostrar tablero
            estado = ComprobarEstado(tab); // Comprobar estado de partida
                                         // gana, empate, nada
        } while (estado==3);
        MostrarGanador(estado);         // Mostrar resultado
    } while (Seguir());
}
```

Aclaraciones sobre el programa:

- Se ha representado el tablero mediante una matriz de caracteres. Si una casilla está libre guardaremos un carácter blanco ' ', si la ocupa el primer jugador colocaremos un carácter 'O' y si la ocupa el segundo jugador colocaremos un carácter 'X'.
- La variable `turno` toma en cada iteración, de forma alternativa, los valores 0 y 1 para saber quién tiene el turno.
- `VaciarTablero`: Pone blancos en todas las posiciones del tablero.
- `VisualizarTablero`: Muestra el tablero en la salida estándar con un formato adecuado.
- `PedirJugada`: Le pregunta a un jugador en qué fila y columna desea poner una nueva ficha. Debe asegurarse de que es una jugada válida. No pone la ficha en el tablero.
- `PonerFicha`: Coloca la ficha de un jugador en el tablero.
- `ComprobarEstado`: Analiza el tablero y determina si hay tres en raya por parte de algún jugador, se ha completado el tablero o bien se puede seguir jugando. El resultado es un entero que codifica esas situaciones. Por ejemplo, puede devolver un valor 0 si gana el primer jugador, un valor 1 si gana el segundo, un valor 2 si hay empate y un valor 3 si podemos seguir jugando.
- `MostrarGanador`: Muestra en la salida estándar quién ha ganado la partida (o si se ha producido un empate). Debe implementar todas estas funciones y otras si las necesitas. Debe deducir la interfaz de las funciones a partir de las llamadas que puede ver en `main()`.

Ejercicio 7.15 — El ahorcado. Implemente el juego del ahorcado en C++. Este es un juego para dos personas (*jugador1* y *jugador2*) y la dinámica es la siguiente:

- El *jugador1* se inventa una palabra y el *jugador2* debe acertarla en un número máximo de intentos.
- En cada intento, el *jugador2* dice una letra y el *jugador1* le dice si está o no presente en la palabra que se inventó. Además, si está, debe mostrarle en qué posiciones está.
- No se permite que el *jugador2* repita letras.

Más concretamente, el programa debe realizar estas tareas:

- Al comienzo, el programa le pide al *jugador1* que le diga la palabra que se ha inventado. El *jugador2* no debería verla. El *jugador1* también podrá indicar el límite de intentos para adivinar.

Para almacenar la palabra usaremos un objeto de tipo **string**. Observe que debe tener cuidado con que el usuario escriba palabras válidas (sin caracteres que no sean alfabéticos). Tenga presente también que se distinguen mayúsculas y minúsculas.

Por ejemplo, si la palabra que se ha inventado es “ballena” y queremos tener un máximo de 4 intentos, el comienzo de la ejecución del programa podría ser este:

```

Consola

Hay dos jugadores:
1.- El que se inventa la palabra
2.- El que adivina la palabra

Jugador 1: escribe una palabra
Jugador 2: ¡No mires!
La palabra debe tener al menos una letra: ballena
Jugador 1: ¿De cuántos intentos disponemos? 4

```

- Comenzamos el juego mostrando al *jugador2* las letras que tiene la palabra que debe adivinar y otra información relevante (número de intentos que le quedan, letras que ha dicho, ...).
- Ahora el *jugador2* irá diciendo letras y el programa le irá indicando si están o no mostrando, en cada intento, información sobre la evolución del juego.
- Si el jugador acierta se mostrarán las letras en la posición correspondiente.
- Si el jugador falla se descontará un intento y se guardará la letra que ha dicho para que lo tenga presente.
- También se informará cuando el *jugador2* repita letras sin penalizar dichos intentos.

A continuación vemos una posible ejecución para averiguar la palabra “ballena” introducida previamente:

```

Consola

La palabra es: -----
Letras usadas:
Intentos : 4
Dime una letra: u
Mal, la letra no está

La palabra es: -----
Letras usadas: u
Intentos : 3
Dime una letra: a
Bien, la letra está en la palabra oculta

La palabra es: -a----a
Letras usadas: u
Intentos : 3
Dime una letra: a
¡ Esa letra ya la dijiste !

```

```

Consola

La palabra es: -a----a
Letras usadas: u
Intentos : 3
Dime una letra: n
Bien, la letra está en la palabra oculta

La palabra es: -a---na
Letras usadas: u
Intentos : 3
Dime una letra:

```

Debe modularizar convenientemente este programa.

Puede hacer más interesante este juego si consigue que, en lugar de que sea el usuario el que escribe la palabra a buscar, sea el propio ordenador el que se la invente. Para ello puede disponer de un listado de palabras almacenado en un vector de

cadenas de caracteres. El programa comenzará eligiendo una de ellas al azar: bastaría con inventarse un número entero entre 0 y el número de palabras almacenadas menos uno, que haría referencia a la posición del vector –la palabra– que vamos a elegir para jugar.

Dicho vector de palabras puede codificarse en el propio programa o bien se puede disponer de un fichero de texto plano con las palabras que se leerá (mediante la técnica de copiar y pegar) al comienzo de la ejecución.

Igualmente el número de intentos podría generarse aleatoriamente, o incluso generarse en función del tamaño de la palabra a buscar o el número de letras distintas que tiene.

Ejercicio 7.16 — Buscaminas. El juego del buscaminas comienza con un tablero de F filas y C columnas, donde se ocultan N minas. Inicialmente no se sabe nada sobre lo que hay debajo de cada casilla, pudiendo ser una casilla sin mina o con mina.

El problema consiste en localizar las minas sin detonar ninguna de ellas. En cada paso, el jugador escoge entre:

1. Destapar una casilla. Si es una mina, habrá detonado y el juego ha terminado sin éxito. Si no lo es, se abrirá la casilla y el resto de casillas del entorno.
2. Marcar una casilla como posición probable de una mina. Es decir, el jugador determina que esta casilla nunca se abrirá, ya que piensa que tiene una mina.

Si el juego continúa y llega el momento en que todas las casillas se han abierto (excepto las que contienen una mina), no quedarán casillas por abrir y, por tanto, se habrá ganado el juego.

Para poder guiar al jugador de forma que pueda averiguar dónde se encuentran las minas, el juego ofrece pistas sobre dónde podrían estar las minas. En concreto, cuando se abre una casilla sin mina existen dos posibilidades:

1. Está vacía, sin mina, y no hay ninguna mina al lado. En este caso, el juego explora automáticamente las 8 casillas que hay a su alrededor, abriendo aquellas que no tengan mina.
2. Está vacía, sin mina, pero tiene una o más minas al lado. En este caso el juego se limita a mostrar al jugador la casilla como vacía, pero mostrando un número que indica el número de minas que están en su entorno.

El jugador deberá analizar los números que se van mostrando para deducir dónde se sitúan las N minas que sabemos oculta el tablero.

La dinámica de juego

Se pretende obtener un programa sencillo que pueda usarse con la consola. Por tanto, las entradas y salidas deberán realizarse con `cin>>` y `cout<<`. Se propone la siguiente interfaz:

- Inicialmente comienza el juego pidiendo los parámetros de dificultad. Estos parámetros están compuestos por el número de filas, el número de columnas, y el número de minas ocultas. Observe que el número de minas no puede ser demasiado grande. Por ejemplo, podemos obligar a que siempre haya un número mínimo de 5 minas y un número máximo que corresponde al 50% de casillas en el tablero.
- En cada iteración, el juego muestra el tablero actual y pregunta por una acción a realizar. La acción puede ser:
 - Abrir una posición. El usuario escribe tres datos, el primero será una palabra “a” o “abrir” (con cualquier combinación mayúsculas/minúsculas), y los dos siguientes la fila y columna correspondientes. En este caso, el programa abre la casilla indicada modificando el tablero según corresponda. Observe que esta acción no hace nada si la casilla ya está abierta o tiene una marca.
 - Marcar/Desmarcar una posición. El usuario escribe tres datos, el primero una palabra “m” o “marcar” (con cualquier combinación mayúsculas/minúsculas), y los dos siguientes la fila y columna correspondiente. Si la casilla está sin marcar la pone como marcada, y en caso de que ya esté marcada elimina la marca.
- El juego termina con éxito cuando todas las casillas sin mina están abiertas. Por otro lado, el juego termina sin éxito si se realiza una acción de apertura sobre una casilla que contiene una mina.

Diseño propuesto

Existen distintas alternativas para almacenar el tablero del juego. En principio, la más sencilla es almacenar la información en un vector de vectores de enteros, y codificar con distintos enteros la situación de cada casilla, incluyendo información sobre el contenido, el estado (abierta o no) y si está marcada.

Para simplificar el problema, proponemos una forma de codificar dicha información, aunque pueden establecerse otras. En nuestra solución, proponemos que se guarde toda la información posible en el tablero, de forma que sea más sencillo y eficiente visualizarlo o procesarlo. En concreto, proponemos que el tablero contenga un entero con la siguiente codificación:

- Si contiene un número del 0 al 9, es una casilla oculta. Además, el entero indica el número de minas que hay alrededor (del 0 al 8), o indica que es una mina (el 9).
- Si contiene un número del 10 al 19, es una casilla que se ha abierto. Contiene la misma información que los números del 0 al 9, aunque como abiertas. Así, por ejemplo, el número 3 indica que está oculta y hay 3 minas alrededor, y el número 3+10 también indica que hay 3 minas alrededor, aunque es una casilla abierta.

- Si contiene un número del 20 al 29, es una casilla marcada. Contiene la misma información que los números del 0 al 9, aunque como marcada. Por ejemplo, el número 3+20 indica que hay 3 minas alrededor y que el usuario la tiene como marcada.

Con esta codificación, podemos deducir que una partida se ha perdido cuando hemos abierto una casilla 9.

Algunas de las funciones o módulos que puede contener la solución son:

- Generar el tablero.
- Imprimir el tablero. Esta función debería facilitar al usuario observar el estado actual del tablero para poder tomar una decisión.
- Solicitar una jugada. Deberá pedir una terna (acción, fila, columna) para procesar la jugada. Recuerde que la acción se especifica con una palabra, y que dicha palabra puede estar en minúscula o mayúscula.
- Comprobar si se ha resuelto el tablero. Será necesario comprobar que no quedan casillas por abrir y que las marcas son correctas.
- Abrir casilla. Corresponde al procesamiento de la acción de abrir sobre una determinada casilla (se explica más adelante).

Lógicamente, se puede decidir el conjunto de funciones que considere más conveniente, incluyendo éstas, parte de éstas, y otras que probablemente necesitará.

Generación del tablero

La generación del tablero consiste en crear una estructura bidimensional con tantas filas y columnas como se desee, y generar aleatoriamente la posición de las minas.

Se debe tener especial cuidado en la generación de minas ya que, al generar la posición de las minas de forma aleatoria, podría ocurrir que se intente colocar más de una vez una mina en una misma casilla. Lógicamente, sólo es posible tener una mina en una casilla, y por tanto, si ya existe una mina, deberá repetirse la generación en una casilla distinta.

Una vez generadas las minas, se deberá recorrer el tablero para asignar, a cada una de las restantes casillas, el valor entero que corresponde al número de minas en su entorno.

Algoritmo para “abrir” una casilla

La parte más complicada, desde el punto de vista algorítmico, es probablemente la apertura de una posición o casilla. Esta dificultad se debe a que abrir una casilla puede provocar la apertura de algunas de su entorno y éstas, a su vez, otras adicionales, de forma que se pueden encadenar un número muy alto de aperturas.

La idea es que la apertura de una casilla que no tiene mina en ella ni en su entorno (codificada como cero) dispara la apertura de cualquier casilla de su alrededor. Un algoritmo sencillo para resolver el problema a partir de una casilla (f_i, c_i) es el siguiente:

1. Añadir la casilla (f_i, c_i) al conjunto C de casillas por abrir.
2. Mientras queden casillas por procesar en C:
 - a) Extraer una casilla (f, c) del conjunto C de pendientes.
 - b) Si la casilla (f, c) tiene un número entre 1 y 8, modificar como abierta.
 - c) Si la casilla (f, c) tiene un número cero (vacía, oculta y sin minas vecinas), ponerla como abierta e insertar todas sus vecinas al conjunto C de pendientes.

Una vez que el conjunto C queda vacío, todas las casillas vacías conectadas con la original (f_i, c_i) se habrán visitado, es decir, se habrán abierto. Observe que cualquier casilla con un valor distinto de cero que se abra no propagará el efecto de apertura a sus vecinas. Además, las casillas que estén marcadas por el usuario (tienen un valor entre 20 y 29) ni se abrirán ni propagarán la apertura.

8

Funciones recursivas

Introducción.....	73
El caso general y el caso base	73
Funciones recursivas con varios puntos de salida.....	74
Múltiples casos base y/o generales.....	74
Ejercicios adicionales.....	74

8.1 Introducción

Este capítulo está dedicado a la resolución de problemas mediante el uso de funciones recursivas. Estas son funciones que se llaman a sí mismas como por ejemplo la siguiente, que calcula el cubo del valor absoluto de un número:

```
int cubo(int num) {
    int resultado;
    if (num>=0)
        resultado = num*num*num;
    else
        resultado = cubo(-num);
    return resultado;
}
```

Aquí se utiliza la recursividad¹ para cambiar el signo del número antes de realizar el producto. Si el número fuese positivo la función devolvería su cubo. Si fuese negativo devolvería el resultado de calcular cubo(-num), es decir, el resultado de calcular el cubo del valor absoluto del numero.

8.2 El caso general y el caso base

Una función recursiva se construye sobre la idea de que, si se intenta resolver un problema de una cierta complejidad, podemos asumir que la función ya es capaz de resolver un problema de iguales características pero de menor complejidad. Si aplicamos esta idea una y otra vez llegará un momento en el que la complejidad sea trivial. En ese punto no será necesario resolver el problema de forma recursiva puesto que sabremos resolverlo de forma más o menos inmediata. A este caso se le denomina *caso base*. El *caso general* sería el que aplica la definición recursiva.

Por ejemplo, considere el problema de calcular el máximo común divisor de dos números a y b mediante el algoritmo de Euclides. Este algoritmo nos dice que el MCD de dos números a y b coincide con el MCD de b y a % b. En el caso de que b sea cero el MCD es a. Expresado mediante una fórmula quedaría:

$$MCD(a,b) = \begin{cases} a & \text{si } b = 0 \\ MCD(b,a \% b) & \text{si } b > 0 \end{cases}$$

Aquí, el caso base (trivial) sería que b valga cero. El resultado sería a sin necesidad de calcular nada. El otro sería el caso general.

Ejercicio 8.1 — Suma de enteros. Considere el problema de calcular la sumatoria de los N primeros números enteros:

$$\text{suma}_N = \sum_{i=1}^N i$$

Piense en una definición recursiva de este problema e implemente una función recursiva que calcule la suma de los N primeros números enteros. N será un valor positivo leído desde la entrada estándar.

¹Lógicamente, si alguna vez tiene que resolver este problema, no lo implemente así.

8.3 Funciones recursivas con varios puntos de salida

Como norma general se recomienda que las funciones tengan un único `return` justo antes de finalizar, aunque vamos a admitir como excepción el caso de que se trate de funciones recursivas. En el ejemplo de la sección 8.1 para el cálculo del cubo vemos cómo se declara una variable `resultado` en donde se almacena el dato que devolverá la función al final. Esta variable se podía haber ahorrado si hubiésemos hecho esta implementación:

```
int cubo(int num) {
    if (num>=0)
        return num*num*num;
    else
        return cubo(-num);
}
```

Con la variable `resultado` la función consume algunos bytes más de memoria. En el caso de que no fuese recursiva no sería grave pero con las funciones recursivas debemos tener presente que cada llamada consume memoria de pila y esta es limitada, por lo que cualquier ahorro de memoria es apreciado.

Más adelante veremos que normalmente se admiten varios puntos de salida en funciones donde obligar a que haya un único `return` oscurece más que facilita la solución. Esto es especialmente cierto cuando las funciones son simples y relativamente cortas, donde es fácil ver las posiciones de los distintos `return` y por tanto entender fácilmente la solución. Además, como hemos visto, las soluciones pueden resultar más eficientes.

Ejercicio 8.2 — Cálculo recursivo. Suponiendo que el valor de `b` no es un número negativo, analice la siguiente función recursiva y explique qué calcula:

```
int func(int a, int b) {
    if (b==0)
        return 0;
    else // si (b>0)
        return a+func(a, b-1);
}
```

8.4 Múltiples casos base y/o generales

Las funciones recursivas pueden tener más de un caso general. Es decir, habrá situaciones en las que la solución al problema pase por hacer una llamada recursiva pero que esta pueda diferir dependiendo de alguna circunstancia.

Ejercicio 8.3 — Mejora del ejercicio “Cálculo recursivo”. Modifique la función del ejercicio 8.2 para tener en cuenta el caso de que `b` sea negativo.

Ejercicio 8.4 — Dígitos de un número. Implemente una función recursiva que reciba un número entero y que devuelva el número de dígitos que tiene. Para resolver este ejercicio recuerde que, según el estándar de C++, el resultado de la operación módulo (%) es un valor positivo si ambos operandos son positivos. Si alguno es negativo no se garantiza si el signo del resto será positivo o negativo. En otras palabras: deberá tener en cuenta el signo de los operandos en su algoritmo recursivo.

De igual forma, las funciones recursivas pueden tener más de un caso base. O lo que es lo mismo, distintas soluciones triviales dependiendo de las circunstancias.

Ejercicio 8.5 — Coeficiente binomial. El coeficiente binomial se puede definir recursivamente de esta forma:

$$\binom{n}{m} = \begin{cases} 0 & \text{si } n < m \\ n & \text{si } m = 1 \\ \binom{n-1}{m-1} + \binom{n-1}{m} & \text{en otro caso} \end{cases}$$

Haga una función recursiva que lo calcule.

8.5 Ejercicios adicionales

Ejercicio 8.6 — Inversión de un número. Implemente una función recursiva que permita invertir los dígitos de un número entero (`int`). La función recibe un dato de tipo `int` como entrada y devuelve otro dato `int` con el resultado. Por ejemplo, si recibe como entrada el número 2634, devolvería 4362.

Ejercicio 8.7 — Escribir secuencia de números. Implemente una función recursiva que escriba todos los números naturales desde el 1 hasta un cierto N (N será un parámetro de la función: un entero positivo).

- Haga una versión que los escriba en orden decreciente (del N al 1).
- Haga otra versión que los escriba en orden creciente (del 1 al N).

Ejercicio 8.8 — Potencia de 2. Implemente una función recursiva que reciba 2 números como entrada (a y b) y que devuelva un valor lógico que diga si a es una potencia de b o no. Por ejemplo:

- EsPotencia(16,2) devuelve true (16 es una potencia de 2).
- EsPotencia(76,5) devuelve false (76 no es una potencia de 5).
- EsPotencia(64,4) devuelve true (64 es una potencia de 4).

Ejercicio 8.9 — Inversión de un vector. Implemente una función recursiva que reciba como argumento un vector y que lo devuelva en orden inverso.

Ejercicio 8.10 — Mínimo de un vector. Implemente una función recursiva que reciba como argumento un vector de números y que devuelva el mínimo de ellos.

Ejercicio 8.11 — Igualdad de cadenas. Implemente una función recursiva que reciba dos cadenas de caracteres y que devuelva un valor entero de entre estos:

- Devolverá 0 si ambas cadenas son idénticas.
- Devolverá -1 si la primera es menor que la segunda.
- Devolverá 1 si la primera es mayor que la segunda.

Una cadena es menor que otra si su posición respecto a la otra, considerando el orden lexicográfico, es anterior. Por ejemplo:

- "Antonio" es menor que "Carlos".
- "Juan" es mayor que "Ana".
- "Jose" es igual que "Jose".
- "antonio" es mayor que "Carlos" (cuidado con las mayúsculas y minúsculas).
- "Juan" es menor que "Juanito".

Para aplicar el procedimiento recursivo, la comparación de las cadenas debe hacerse carácter a carácter.

Ejercicio 8.12 — Ordenación de la burbuja. Implemente de forma recursiva el método de ordenación de la burbuja para ordenar un vector de enteros. No se permite usar ningún tipo de bucle.

Ejercicio 8.13 — Máximo de las medias. Debe implementar una aplicación que lea una matriz de tamaño arbitrario por la entrada estándar y que calcule la mayor de las medias de cada fila de la matriz.

El algoritmo (la función `main()`) debe realizar estos pasos:

- Preguntar el número de filas de la matriz. Debes asegurarte de que es un valor mayor que cero.
- Preguntar el número de columnas de la matriz. Debe ser mayor que cero.
- Leer la matriz del tamaño indicado.
- Calcular la mayor de las medias de cada fila.
- Mostrar el valor resultante.

Por ejemplo, en la siguiente matriz:

2	5	4	7
4	7	1	2
3	2	1	9

la media de la primera fila es 4.5, la media de la segunda es 3.5 y la media de la tercera es 3.75. Por tanto, la mayor de las medias es 4.5.

Restricción: todas las funciones deben ser recursivas y **en ningún momento puede usar bucles**.

Ejercicio 8.14 — Laberinto. Imagine que está diseñando un juego en el que el personaje principal se mueve por un laberinto. Hemos de ayudarle implementando una función que le permita encontrar el camino de salida. Para ello, la función debe tener una representación del laberinto, es decir, el mapa del laberinto.

Se usará una matriz bidimensional como mapa: sus celdas contendrán valores 0 y 1. Si en una celda hay un 1 significa que es una pared mientras que si es un cero significa que está libre. A continuación puede ver un ejemplo de mapa:

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1
2	1	0	1	1	0	1	0	1	0	1
3	1	0	1	1	0	1	0	1	0	1
4	1	0	1	0	0	1	1	0	0	1
5	1	0	1	1	()	0	0	1	0	0
6	1	1	1	1	1	0	0	1	1	1
7	1	1	1	1	1	1	1	1	1	1

En este ejemplo se aprecia que el personaje está situado en la casilla (5,4). El camino de salida sería la secuencia de casillas:

$$(4,4) - (3,4) - (2,4) - (1,4) - (1,5) - (1,6) - (1,7) - (1,8) - (2,8) - (3,8) - (4,8) - (5,8) - (5,9)$$

Diseñe e implemente una función recursiva que muestre en la salida estándar la secuencia de casillas que debe visitar el personaje para salir del laberinto.

Observaciones:

- El tamaño de la matriz es arbitrario.
- La posición de partida del personaje es arbitraria, aunque debe ser una casilla con un valor cero.
- Se considera que una casilla es una salida del laberinto si está vacía (es un cero) y está en una fila o columna de los extremos de la matriz.
- Se puede suponer que hay exactamente una salida del laberinto, aunque sería fácil considerar la posibilidad de muchas o ninguna salida.
- No es posible andar en diagonal. Por ejemplo, no se puede mover desde la casilla (4,7) hasta la (3,6).

Ejercicio 8.15 — Anagramas. Se dice que una palabra es un anagrama si se obtiene de la transposición de letras de otra palabra. Por ejemplo:

- Monja, Jamón.
- Roma, amor, mora, ramo.

Diseñe e implemente una función recursiva que, dada una cadena de caracteres, calcule y devuelva un vector con las cadenas resultantes de hacer todas las posibles permutaciones de letras de la palabra recibida (tengan o no sentido según el diccionario, es decir, que las cadenas resultantes no tienen por qué ser palabras que existan).

A continuación implemente un programa que lea dos palabras y haciendo uso de la función recursiva nos diga si una es un anagrama de la otra o no. Para ello deberá añadir una nueva función recursiva que reciba un vector de cadenas y una cadena y nos diga si la cadena se encuentra o no entre las cadenas del vector.

No deberá considerar las tildes aunque sí se podrían considerar de forma sencilla las mayúsculas y minúsculas.

Ejercicio 8.16 — Abrir recursivamente en buscaminas. Considere el ejercicio 7.16 (página 71) que implementa el buscaminas. Diseñe e implemente una solución recursiva para el problema de “abrir” una casilla. Compare la complejidad del código obtenido con el que desarrolló haciendo uso de vectores y técnicas iterativas.

9

Estructuras y pares

Introducción.....	77
Estructuras en C++	77
Estructuras y funciones	
Anidamiento de estructuras	
Estructuras y otros datos compuestos	
El tipo par de la STL.....	79
Ejercicios adicionales.....	80



9.1 Introducción

En los temas anteriores se han propuesto problemas de cierta entidad, aunque se han limitado al desarrollo de soluciones que se basan en algoritmos que trabajan con tipos de datos cadena y vector. Estos tipos permiten manejar grandes cantidades de datos, aunque todos de un tipo común. Así, un vector puede tener miles de datos, pero todos de un tipo determinado.

Para complementar este tipo de estructuras de datos, en este capítulo se proponen problemas en los que se manejan datos compuestos, aunque en este caso heterogéneos. Es decir, estructuras de datos que pueden crearse como composición de varios datos de distintos tipos.

9.2 Estructuras en C++

El tipo de dato **struct** permite la creación de nuevos tipos de dato basándose en otros ya existentes, como por ejemplo los que proporciona el lenguaje de forma nativa (**int**, **double**, **bool**, ...).

Un ejemplo podría ser la definición de una estructura que permita almacenar las coordenadas de un punto en un espacio 2D. Este nuevo dato debe tener capacidad para almacenar dos números reales: la abcisa y la ordenada del punto. Podría ser como sigue:

```
struct Punto {  
    double x;  
    double y;  
};
```

Una vez definido el tipo se pueden declarar objetos de ese tipo con la sintaxis habitual para declaraciones. El siguiente código declara dos variables *p1* y *p2*, cada una de las cuales almacena internamente dos valores reales (las coordenadas):

```
Punto p1, p2;
```

El acceso a los componentes individuales que forman el nuevo tipo se hace mediante el operador punto. Por ejemplo, para leer y escribir un punto en la E/S estándar escribiremos esto:

```
Punto pto;  
cout << "Escriba las coordenadas: ";  
cin >> pto.x >> pto.y;  
cout << "El punto es: (" << pto.x << "," << pto.y << ")" << endl;
```

Ejercicio 9.1 — Puntos en un espacio 2D. Defina una estructura para almacenar las coordenadas de un punto en un espacio euclídeo 2D como la del ejemplo anterior. Para probarla, escriba un programa que lea desde la entrada estándar dos puntos, que muestre los valores de los puntos leídos y finalmente escriba la distancia euclídea entre ambos.

9.2.1 Estructuras y funciones

Al igual que ocurre con otros tipos compuestos –como **vector** y **string**– las estructuras también pueden:

- Pasarse por valor a una función.
- Pasarse por referencia a una función.
- Devolverse con **return** de una función.

De forma análoga, y dado que podrían contener en su interior gran cantidad de datos, es frecuente que los pasos por valor se eviten y se realicen pasos por referencia constante –mediante `const` &– por cuestiones de eficiencia.

Ejercicio 9.2 — Puntos en un espacio 2D con funciones. Retome el ejercicio 9.1 y modifíquelo para incluir, al menos, las siguientes funciones:

- Leer un punto desde entrada estándar. La función devolverá el punto mediante `return`.
- Escribir un punto en la salida estándar.
- Calcular la distancia euclídea entre 2 puntos.

Ejercicio 9.3 — El tipo fecha. Considere de nuevo el ejercicio 6.24 y realice un nuevo diseño del mismo. Para ello defina una estructura para almacenar fechas (día, mes y año) y, a continuación, implemente funciones para:

- Comprobar si una fecha es o no válida.
- Leer una fecha desde la entrada estándar y validarla. Si no es una fecha válida debe pedirla de nuevo.
- Calcular la diferencia en días entre dos fechas.

Haga un programa que lea la fecha actual y la de su nacimiento y que diga cuántos días faltan para su próximo cumpleaños.

9.2.2 Anidamiento de estructuras

Una estructura permite empaquetar en un dato múltiples datos. Estos pueden ser tipos simples pero también pueden ser, a su vez, otras estructuras. Por ejemplo, si deseamos crear un nuevo tipo de dato para almacenar círculos, podríamos hacerlo definiendo un círculo como el punto central junto con el radio. Suponiendo que ya disponemos de una estructura `Punto` como la vista anteriormente, un círculo se podría definir así:

```
struct Circulo {
    Punto centro;
    double radio;
};
```

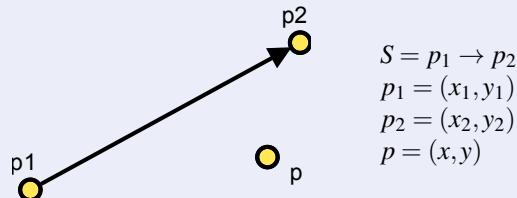
y, por ejemplo, podríamos acceder a la abcisa del centro del círculo usando de nuevo el operador punto:

```
Circulo cir;
cout << "Digame cuál es la abcisa del centro:";
cin >> cir.centro.x;
```

Ejercicio 9.4 — Segmentos de recta. Defina una estructura para almacenar segmentos de recta. Un segmento se define en base a una pareja de puntos. Implemente las siguientes funciones:

- Lectura de un segmento desde la entrada estándar.
- Escritura de un segmento en la salida estándar.
- Función que evalúa si un punto se encuentra a la derecha o a la izquierda de un segmento.
- Función que evalúa si dos segmentos se cruzan.

Suponiendo la situación mostrada en la figura y de acuerdo a esta notación:



Podemos saber si el punto p está a la derecha o a la izquierda del segmento S calculando el signo de la expresión $(x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$. Si el signo es negativo el punto está a la derecha y si es positivo a la izquierda.

Para averiguar si dos segmentos S_1 (definido por los puntos P_{11} y P_{12}) y S_2 (definido por P_{21} y P_{22}) se cruzan, debemos comprobar si P_{11} y P_{12} están en lados opuestos del segmento S_2 y, además, P_{21} y P_{22} están en lados opuestos de S_1 .

Haga un programa que lea dos segmentos desde la entrada estándar y que, a continuación, muestre en la salida estándar los siguientes datos:

- Datos que definen ambos segmentos.
- Si se cruzan o no entre sí.

9.2.3 Estructuras y otros datos compuestos

En ocasiones se necesita almacenar gran cantidad de datos de un tipo definido mediante `struct`. Para ello podremos usar, como es de esperar, el tipo `vector`, definiendo vectores de estructuras. Suponga que necesitamos almacenar los vértices que

forman un polígono con un número arbitrario de lados. Puesto que cada vértice se puede definir mediante un dato de tipo *Punto*, el polígono podría definirse así:

```
vector<Punto> poligono;
```

De esta forma, podemos mostrar los vértices con un código similar a este:

```
for (int i=0; i<poligono.size(); i++)
    cout << "Vértice " << i << ":" <<
        << poligono[i].x << "," << poligono[i].y << ")" << endl;
```

De igual forma, dentro de una estructura puede haber objetos de tipo compuesto como vectores y cadenas. Imagine que necesitamos definir un nuevo tipo para almacenar polígonos que, además de los vértices, incluya información adicional sobre el color de las aristas y el color interior. Los colores podrían representarse simplemente mediante una cadena de texto.

```
struct Poligono {
    vector<Punto> vertices;
    string color_aristas;
    string color_interior;
};
```

Ejercicio 9.5 — Baraja de cartas. Se está trabajando en una aplicación que implemente un juego de cartas haciendo uso de la baraja española. Debe implementar estructuras para representar:

- Una carta. En este caso la carta viene definida por un palo y un número.
- Una baraja. La baraja es un montón de cartas.

Realice un programa (con funciones) que:

- Cree una baraja española con todas sus cartas.
- Muestre la baraja en la salida estándar.
- Mezcle las cartas de la baraja aleatoriamente
- Muestre la baraja mezclada en la salida estándar.

Observe que aunque la baraja se puede definir como un simple vector, se pide que se implemente dentro de una estructura. De esta forma la declaración de objetos de tipo baraja de cartas es más legible y fácil de entender que si los definimos como si fuesen un vector de cartas simplemente.

Ejercicio 9.6 — Polígonos. Defina una estructura para almacenar polígonos. Recuerde que un polígono se puede definir como una secuencia de puntos. A continuación realice un programa modular que permita calcular el perímetro de dicho polígono.

Una vez hecho, añada la funcionalidad que nos permita conocer si existe al menos alguna arista que se cruza con otra. Para ello debería hacer uso de la información del ejercicio 9.4 en donde se explica cómo determinar si dos segmentos de recta se cortan o no.

Con esta nueva función, modifique el programa anterior para que indique con un error que el polígono no es válido.

9.3 El tipo par de la STL

Con cierta frecuencia se necesita disponer de estructuras compuestas por únicamente dos miembros que, además, no requieren ser tratadas con la funcionalidad especial de un nuevo tipo de dato que incluye nuevas operaciones, sino que son simplemente pares de valores que se unen en un único objeto. Para estas situaciones se dispone del tipo **pair**. Para poder usar este nuevo tipo hay que hacer incluir el fichero **utility**.

Por ejemplo, si queremos definir una carta de una forma muy simple podemos aprovechar el tipo **pair** de la STL para incluir un par formado por el palo y el número de la siguiente forma:

```
#include <utility>
...
pair<char,int> carta;
```

El acceso a los miembros de la pareja se hace mediante *first* y *second*:

```
carta.first = 'E'; // Espadas
carta.second = 4;
```

Al igual que con el resto de tipos de C++ es posible:

- Pasar pares como argumentos de funciones por valor y por referencia.
- Devolver pares con **return** en las funciones.
- Declarar vectores de pares.
- Usar tipos compuestos como parte de un par.

La asignación de valores a los pares se puede hacer miembro a miembro (con *first* y *second*) como hemos visto. Además, es posible usar una sintaxis alternativa especificando el tipo y los valores entre paréntesis. Por ejemplo, el siguiente trozo de código consigue el mismo efecto que el ejemplo anterior:

```
carta = pair<char,int>('E', 4);
```

Ejercicio 9.7 — Temperaturas de ciudades. Realice un programa que almacene, en un vector de pares, los nombres y las temperaturas medias de una serie de ciudades leídas desde la entrada estándar. Cuando finalice la lectura de datos mostrará, calculándolo a partir del vector almacenado, el nombre de la ciudad cuya temperatura es mayor que todas las demás.

Ejercicio 9.8 — Agenda de teléfonos. Realice un programa que almacene, en un vector de pares, los nombres y los teléfonos de una serie de personas. Dicho programa comenzará con la agenda de contactos vacía y dispondrá de un menú con las siguientes opciones:

- Añadir nueva entrada a la agenda.
- Buscar el teléfono de una persona dado su nombre.
- Mostrar un listado alfabético de la agenda.

Para realizar el programa debe tener en cuenta la siguiente restricción: el vector siempre se encuentra ordenado alfabéticamente por el nombre de la persona. Esto quiere decir que cuando se inserta un nuevo contacto, debe hacerse en la posición que le corresponde en el vector de acuerdo a esta ordenación. Aprovechando este hecho, la búsqueda de contactos se hará usando el algoritmo de búsqueda binaria.

9.4 Ejercicios adicionales

Ejercicio 9.9 — Biblioteca. Se está diseñando un programa para gestionar una biblioteca y para cada libro se debe almacenar la siguiente información:

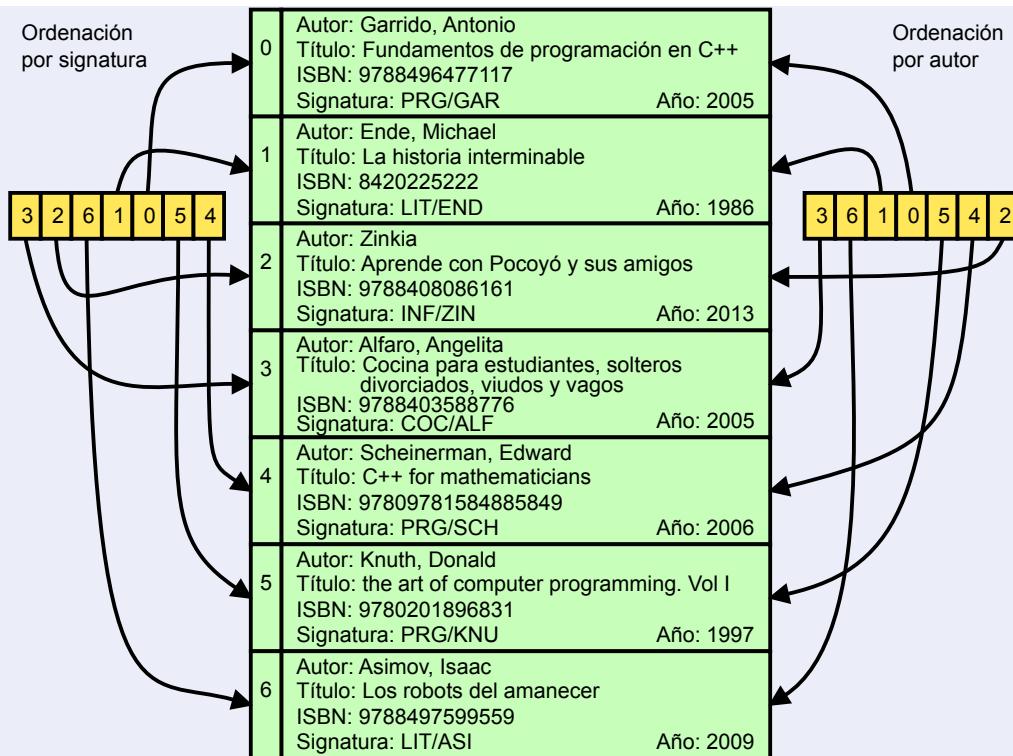
- Autores.
- Título.
- ISBN.
- Signatura.
- Año de publicación

Defina una estructura para almacenar datos de un libro. A continuación debe crear una aplicación que permita gestionar un conjunto de libros. Dicho conjunto se almacenará en un **vector**. Para que la gestión de libros sea más cómoda, el bibliotecario necesita obtener listados de los libros según distintos órdenes:

- Por autor.
- Por signatura.

Dado que el vector de libros puede contener gran cantidad de información, no es adecuado realizar una ordenación de ese vector cada vez que necesitemos hacer un listado. En lugar de eso, se van a crear dos funciones (una para ordenar por autor y otra por signatura). Cada una de estas funciones creará, a partir del vector de libros, un vector de *índices* que permite obtener una ordenación de la biblioteca por cada campo.

La figura que tiene a continuación muestra un ejemplo. En el centro está representado el vector que contiene la información de los libros con la numeración de cada posición del vector. Observe que este vector no está ordenado de ninguna forma. De esta forma, cuando se añade un nuevo a libro a la biblioteca simplemente se añade al final del vector: la inserción o el borrado de nuevos datos es muy rápida.



A la izquierda del vector se puede ver el vector índice que almacena las posiciones de cada libro en el vector de libros de forma que se consigue la ordenación por signatura. A la derecha vemos otro vector de índices, pero esta vez con la ordenación por autor. Estos vectores contienen las posiciones del vector original ordenadas de acuerdo al criterio elegido.

Ejercicio 9.10 — Estadísticas de texto. Debe escribir un programa que lea un texto desde la entrada estándar y que muestre algunos estadísticos sobre el mismo. En particular, se desea calcular:

- La frecuencia de aparición de cada carácter (histograma). Se mostrarán ordenados alfabéticamente.
- El carácter que más veces aparece (la moda).
- La mediana (si presentamos los valores del histograma ordenados por frecuencias: el elemento central).

Sólo se contabilizan caracteres del ASCII estándar, es decir, aquellos cuyo código ASCII está entre el 32 y el 127. Por ejemplo, si la entrada al programa fuese esta:

```
Consola

Escriba un texto (acabado en #):
Esto es un texto para probar la practica de conteo de caracteres. Como puedes observar, se ha evitado el uso de tildes y para finalizar la entrada de datos se usa el caracter almohadilla. Lo normal sera que el caracter que mas veces aparece sea el espacio en blanco#
```

El resultado sería este otro:

Consola

```
Este es el histograma asociado al texto:
( 32) = 48
# ( 35) = 1
, ( 44) = 1
. ( 46) = 2
C ( 67) = 1
E ( 69) = 1
L ( 76) = 1
a ( 97) = 35
b ( 98) = 3
c ( 99) = 13
d (100) = 10
e (101) = 33
f (102) = 1
h (104) = 2
i (105) = 7
l (108) = 13
m (109) = 4
n (110) = 7
o (111) = 16
p (112) = 7
q (113) = 2
r (114) = 17
s (115) = 16
t (116) = 12
u (117) = 6
v (118) = 3
x (120) = 1
y (121) = 1
z (122) = 1

La moda es ' ' y la mediana es 'm'.
```

Para resolver el problema debe definir un vector que contenga las letras que se van encontrando en el texto junto con su frecuencia. Dicho vector no almacenará datos sobre letras que no existan en el texto. Además, se implementará una función que reciba como entrada una letra y que modifique el vector para contabilizarla.

Veamos un ejemplo sobre cómo se desarrollaría el esquema de conteo con el texto del ejemplo previo.

- Inicialmente el vector contendrá cero datos puesto que aún no se ha leído ninguna letra.
- Tras leer la primera letra ('*E*' en el ejemplo), el vector contendrá la información de que la '*E*' aparece una vez: {'E'=1}.
- Tras leer la '*s*' se añadiría la información de que la letra '*s*' aparece una vez: {'E'=1, 's'=1}.
- {'E'=1, 's'=1, 't'=1}.
- {'E'=1, 'o'=1, 's'=1, 't'=1}.
- {' '=1, 'E'=1, 'o'=1, 's'=1, 't'=1}.
- {' '=1, 'E'=1, 'e'=1, 'o'=1, 's'=1, 't'=1}.
- Cuando se añade una letra que ya apareció con anterioridad, en lugar de añadir una nueva entrada en el vector, simplemente se incrementa la frecuencia de aparición de esa letra: {' '=1, 'E'=1, 'e'=1, 'o'=1, 's'=2, 't'=1}.
- ...

Como puede observar, la estructura siempre debe mantener ordenada la lista de letras, de esa forma se puede implementar una búsqueda binaria para averiguar de forma eficiente si una letra está o no presente.

Cuando finalice el conteo y muestre la información anterior, el programa entrará en un bucle que permitirá al usuario preguntar por la frecuencia de aparición de alguna letra concreta mientras este lo deseé. Por ejemplo, dado el texto anterior, si el usuario pregunta cuántas veces aparece la letra '*t*' el programa le dirá que 12.

Ejercicio 9.11 — Siete y media. Haciendo uso del tipo baraja de cartas definido en el ejercicio 9.5 implemente el juego de las "siete y media". Puede consultar las reglas, por ejemplo, en Wikipedia.

10

Flujos de Entrada/Salida

10.1 Introducción

Los flujos son el mecanismo que se utiliza en C++ para la E/S básica, es decir, para la comunicación con dispositivos externos como teclado, monitor o ficheros en unidades de almacenamiento externas. Son una abstracción que permite que los programas sean independientes de la forma de funcionar del hardware.

Conceptualmente, podemos concebir un flujo de datos como una tubería por la que fluyen los datos de uno en uno. Cada dato, en nuestro caso, sería un carácter (1 byte). Los flujos que ya se conocen son los siguientes:

- **cin**. Es el flujo que permite a los programas leer datos desde la entrada estándar.
- **cout**. Es el flujo que permite a los programas escribir datos en la salida estándar.

Además, C++ define otros dos flujos estándares también para salida de datos. Por defecto ambos están asociados a la consola por lo que, aparentemente, no presentan diferencias con **cout**. Son los siguientes:

- **cerr**. Es un flujo al que se envían datos de forma análoga a **cout**. La diferencia es que los mensajes enviados a este flujo tienen como objetivo informar sobre errores durante la ejecución de un programa mientras que los enviados a **cout** son mensajes de información al usuario de la aplicación.
- **clog**. Es otro flujo de salida para enviar mensajes de estado. Permite llevar una bitácora de lo que va ocurriendo en el programa.

Considere el siguiente código:

```
int a, b;

clog << "Pidiendo datos" << endl;
cout << "Escriba dos números y calcularé su división: ";
cin >> a >> b;

clog << "Verificando datos" << endl;

if (b==0) {
    cerr << "Error: división por cero" << endl;
    cout << "No se puede hacer la división" << endl;
} else {
    clog << "Mostrando resultado" << endl;
    cout << "El primero entre el segundo vale: " << a/b << endl;
}
```

Podemos apreciar que:

- Los mensajes enviados a **cout** son los que permiten la interacción con el usuario de la aplicación.
- Los mensajes enviados a **clog** sirven para ir anotando el progreso de la ejecución por si más adelante deseamos analizar qué ha ido ocurriendo.
- Los mensajes enviados a **cerr** son mensajes de error.

Por defecto, al estar asociados los tres flujos a la salida estándar (consola), el resultado puede ser confuso. A continuación vemos dos ejemplos de ejecución:

Introducción.....	83
Operaciones básicas de transferencia	
Ficheros	86
Apertura de ficheros	
Añadiendo datos a ficheros	
Cierre de ficheros	
Detección del fin del flujo	88
Control de errores en los flujos	89
Control de errores en ficheros	
Reutilización de flujos	90
Otras operaciones sobre flujos	90
Flujos y funciones	91
Copia de flujos	
Uso de flujos "genéricos"	
Algunas recomendaciones finales.....	92
Lectura adelantada y flujos	
Ventajas e inconvenientes de "no-op"	
Evitar la lectura parcial de objetos	
Ejercicios adicionales.....	95

Consola

```
Pidiendo datos
Escriba dos números y calcularé su división: 3 0
Verificando datos
Error: división por cero
No se puede hacer la división
```

Consola

```
Pidiendo datos
Escriba dos números y calcularé su división: 3 2
Verificando datos
Mostrando resultado
El primero entre el segundo vale: 1
```

Sin embargo, como se ve en el apéndice B, es posible separar las salidas para que no todas se muestren en la consola, sino que pueden enviarse a otro destino, como un fichero.

Aunque no vamos a usar habitualmente todos estos flujos estándar, es interesante que observe la conveniencia de cada uno de ellos. Por ejemplo, imagine que realizamos un programa que necesita ejecutarse durante días, realizando cálculos, obteniendo resultados, etc. El programa en ejecución puede usar:

- El flujo `cout` para ir obteniendo los resultados independientemente del resto de mensajes. Se podría enviar a un archivo que se analizará posteriormente.
- El flujo `cerr` para indicar algún error. Se podría enviar a la consola, pues si se produce un error, el usuario debería revisarlo cuanto antes.
- El flujo `clog` para indicar cómo va el proceso. Puede informar de lo que va ocurriendo conforme avanza, para que el usuario pueda consultar el estado en el momento que deseé.

10.1.1 Operaciones básicas de transferencia

Por ahora los operadores más usados para la transferencia de datos han sido estos:

- Operador `>>`. Lee datos desde un flujo. Para ello interpreta lo que se recibe desde el flujo y lo convierte según sea el tipo de dato del objeto en el que se debe almacenar. Además, descarta los separadores convenientemente (espacios en blanco, saltos de línea y tabuladores).
- Operador `<<`. Escribe datos en un flujo.

Ejercicio 10.1 — Separadores. Analice el siguiente código:

```
int a, b, c;
cout << "Escriba los datos: ";
cin >> a >> b >> c;
cout << a << " " << b << " " << c << endl;
```

Si ejecutar el código previo, indique cuál será el resultado de la siguiente ejecución:

Consola

```
Escriba los datos: 12      34.56     78
```

Ejercicio 10.2 — Más separadores. Analice el siguiente código:

```
char a, b, c;
cout << "Escriba los datos: ";
cin >> a >> b >> c;
cout << a << " " << b << " " << c << endl;
```

Si ejecutar el código previo, indique cuál será el resultado de las siguientes ejecuciones:

Consola

```
Escriba los datos: xyz
```

Consola

Escriba los datos: x y z

Observe que el operador de salida de datos `<<` realiza una transformación de la representación interna de los datos y la adapta para que sea comprensible para el usuario. Por ejemplo, si ejecuta el siguiente código:

```
char c= 'A';
int v= c;
cout << "Como carácter: " << c << endl;
cout << "Como entero: " << v << endl;
```

observará este resultado:

Consola

Como carácter: A
Como entero: 65

Tanto la variable `c` como la variable `v` almacenan el valor entero 65. Sin embargo, cuando se envían a la salida estándar, este valor es transformado o no según el tipo de dato de la variable.

Lectura de datos sin transformaciones

En situaciones en las que se necesita procesar todos los caracteres desde un flujo de entrada (incluyendo los separadores) se pueden usar estas operaciones:

- `getline()` (véase 5.3.1). Permite leer cadenas de caracteres para que sea posible la lectura de los separadores como parte de la cadena.
- `get()` (véase 5.3.3). Lee caracteres individuales del flujo. No se interpretan los separadores.

Ejercicio 10.3 — Lectura de caracteres individuales. Analice el siguiente código:

```
char a, b, c;
cout << "Escriba los datos: ";
a= cin.get();
b= cin.get();
c= cin.get();
cout << a << " " << b << " " << c << endl;
```

Sin ejecutar el código previo, indique cuál será el resultado de las siguientes ejecuciones:

Consola

Escriba los datos: xyz

Consola

Escriba los datos: x y z

Ejercicio 10.4 — Lectura de cadenas. A continuación puede ver dos versiones de un trozo de programa que lee una cadena de caracteres:

```
// Versión 1
string cad;
cout << "Escriba un texto: ";
cin >> cad;
cout << "Ha escrito: " << cad << endl;

// Versión 2
string cad;
cout << "Escriba un texto: ";
getline(cin,cad);
cout << "Ha escrito: " << cad << endl;
```

Si ejecutar el código previo, indique cuál será el resultado de ambas versiones para la siguiente ejecución:

Consola

Escriba un texto: Había una vez

Escritura de caracteres individuales

C++ provee también de una operación recíproca a `get()`:

- `put()`. Escribe caracteres individuales en un flujo.

Por ejemplo, el siguiente código escribe la palabra *Hola* en la salida estándar:

```
cout.put('H');
cout.put('o');
cout.put('l');
cout.put('a');
```

Note que aunque se ha escrito carácter a carácter, el efecto final es idéntico a escribir la palabra completa.

Ejercicio 10.5 — Contar separadores. Escriba un programa que, usando únicamente las operaciones `put` y `get`, lea una cadena de caracteres hasta que se encuentre un carácter '`@`' y la almacene en una variable de tipo `string`. Posteriormente el programa contará cuántos caracteres de los almacenados son separadores (espacios en blanco, tabuladores y saltos de línea). A continuación puede ver un ejemplo de ejecución:

```
Consola
Escriba un texto: Caminante, son tus huellas
el camino y nada más;
Caminante, no hay camino,
se hace camino al andar. @
Hay un total de 17 separadores.
```

10.2 Ficheros

Aunque mediante la redirección es posible leer y escribir en ficheros usando la E/S estándar (véase apéndice B), es frecuente que no queramos modificar el comportamiento de la E/S básica a la vez que transferir datos a o desde ficheros. Es decir, la necesidad de usar a la vez y de forma independiente la E/S básica y el sistema de ficheros es indiscutible.

Para acceder a los ficheros seguiremos usando el mecanismo de flujos. Al incluir el archivo de cabecera `fstream`, disponemos de nuevos tipos de datos para declarar objetos de tipo "flujo asociado a fichero":

- **`ifstream`**. Este tipo permite declarar objetos de tipo flujo asociados a ficheros para realizar entrada de datos (lectura desde ficheros).
- **`ofstream`**. Este tipo permite declarar objetos de tipo flujo asociados a ficheros para realizar salida de datos (escritura en ficheros).

El esquema que se sigue para trabajar con los ficheros es el siguiente:

1. Abrir fichero. Es necesario indicarle al Sistema Operativo cuál es el fichero sobre el que vamos a trabajar y será este el que lo prepare para ello.
2. Transferir datos. Una vez abierto el fichero podemos proceder a la transferencia de datos. Al ser objetos de tipo flujo, podemos usar las operaciones que ya conocemos para transferir datos con los flujos estándares: `<<`, `>>`, `getline`, `get` y `put`.
3. Cerrar fichero. Al finalizar la transferencia se debe cerrar el fichero y desligar el flujo de él. Tras cerrar un fichero no es posible transferir datos.

10.2.1 Apertura de ficheros

Para abrir un fichero usaremos la operación `open()` del flujo. Esto es válido para cualquiera de los flujos asociados a ficheros (`ifstream`, `ofstream`). Una vez abierto ya es posible realizar transferencias. Veamos un ejemplo:

```
ofstream f;           // Declaramos el objeto de tipo flujo de salida
f.open("fichero.txt"); // Apertura del fichero "fichero.txt"
f << "Escribimos algún texto" << endl;
f << "Y números: " << 254 << ", " << 567 << endl;
```

Como resultado de ejecutar este programa se crea un fichero de texto en el disco con el contenido:

```
Escribimos algún texto
Y números: 254, 567
```

Observe que el nombre del fichero se ha escrito como un literal de tipo cadena de caracteres *estilo-C*. Si deseamos abrir un fichero cuyo nombre tenemos almacenado en un `string`, debemos hacerlo como vemos en este otro ejemplo:

```
ofstream f;           // Declaramos el objeto de tipo flujo de salida
string nombre="fichero.txt"; // Tenemos el nombre en un string
f.open(nombre.c_str()); // Apertura del fichero
```

Ejercicio 10.6 — Números naturales. Escriba un programa que le pregunte al usuario el nombre de un fichero por la entrada estándar y que, a continuación, almacene en él los 100 primeros números naturales.

Ejercicio 10.7 — Media de tres números. Escriba un programa que lea desde un fichero 3 números y que muestre en la salida estándar su media. Antes de la ejecución debe crear un fichero en un editor de textos y almacenar en él los tres números.

Se puede abreviar la apertura de un fichero haciéndolo en la misma declaración del flujo. El resultado es el mismo que usando la operación `open()`. La sintaxis es esta:

```
ofstream f("fichero.txt");
```

Apertura en C++11

En el estándar C++11 sí es posible usar una cadena de tipo `string` como nombre de fichero en la apertura. Las siguientes llamadas a la operación `open()` son válidas:

```
string nombre="fichero.txt";
ifstream f1("fichero.txt");           // C++98 y C++11
ifstream f2(nombre.c_str());         // C++98 y C++11
ifstream f3(nombre);                // Solo C++11
```

10.2.2 Añadiendo datos a ficheros

Cuando se abre un fichero de salida de datos (`ofstream`), pueden ocurrir dos cosas:

1. Que el fichero no exista previamente. En este caso se crea vacío.
2. Que el fichero ya exista con anterioridad. En este caso se borra el contenido previo y se crea de nuevo vacío.

Si lo que deseamos es abrir un fichero para escribir en él sin perder lo que ya tenía podemos hacerlo así:

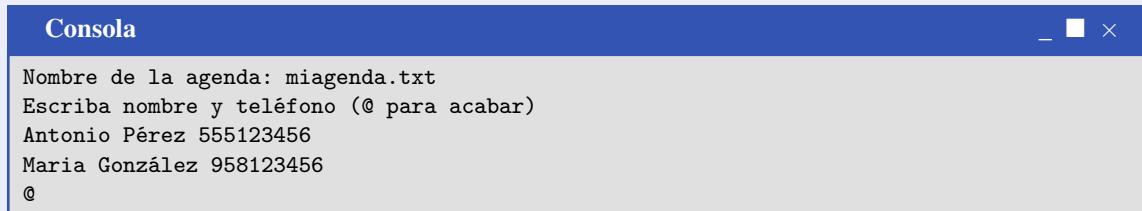
```
f.open("fichero.txt",ios::app);
```

Tras esto, si el fichero existía previamente mantiene su contenido y las operaciones de escritura sobre él añaden al final datos. Si el fichero no existía lo crea vacío.

Ejercicio 10.8 — Agenda telefónica. Implemente un programa que permita mantener una pequeña agenda de contactos. Al comenzar la ejecución el programa preguntará el nombre de nuestra agenda (que coincidirá con el nombre del fichero en el que vamos a almacenar los contactos). El programa leerá datos de contactos –nombre y teléfono– hasta que el usuario decida terminar (por ejemplo usando un carácter '`@`'). El programa almacenará los contactos en el fichero de texto.

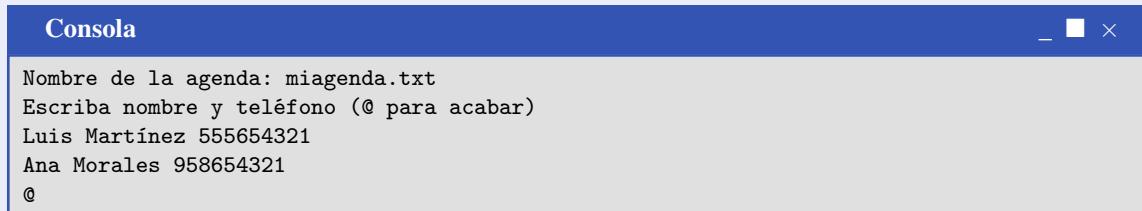
Debe tener en cuenta que, al ser una agenda, cada vez que ejecutamos el programa éste deberá tener cuidado de añadir contactos al final de la misma sin perder los que teníamos previamente. Para verificar que funciona bien deberá visualizar el contenido del fichero de texto en un editor de texto.

Un ejemplo de ejecución sería este:



```
Nombre de la agenda: miagenda.txt
Escriba nombre y teléfono (@ para acabar)
Antonio Pérez 555123456
Maria González 958123456
@
```

Si realizamos una nueva ejecución como esta:



```
Nombre de la agenda: miagenda.txt
Escriba nombre y teléfono (@ para acabar)
Luis Martínez 555654321
Ana Morales 958654321
@
```

El fichero "miagenda.txt" contendrá esta información:

```
Antonio Pérez 555123456
Maria González 958123456
Luis Martínez 555654321
Ana Morales 958654321
```

10.2.3 Cierre de ficheros

El cierre de un fichero se hará cuando se haya finalizado la transferencia de datos. Por distintos motivos, esta es una operación imprescindible. Si no se hace de forma explícita con la operación `close()`, los ficheros se cerrarán de forma automática al terminar la ejecución del bloque en el que se hayan declarado los objetos de tipo flujo. Siguiendo con el ejemplo anterior, el cierre del fichero se haría así:

```
f.close();
```

10.3 Detección del fin del flujo

Cuando se lee un dato desde la entrada estándar `cin`, el programa espera hasta que escribimos algo y pulsamos retorno de línea. En ese momento lee el dato, lo almacena y continúa la ejecución. Este comportamiento parece lógico pero, como estamos viendo, mediante el mecanismo de flujos también es posible leer datos desde ficheros (usando flujos asociados a ficheros o incluso usando la redirección de entrada de datos). En este escenario se podría dar la situación de que, tras haber leído todos los datos de un fichero, el programa intente hacer una nueva lectura. Puesto que el contenido del fichero es el que es y no hay posibilidad de añadir más mientras se está leyendo, el programa se detendría indefinidamente sin opción de avanzar su ejecución.

Por tanto, se necesita un mecanismo para comprobar si aún quedan datos por leer o, por el contrario, hemos llegado al final del flujo. Podemos considerar que todo flujo acaba en una marca especial denominada *EOF* (End-Of-File), es decir, tras el último byte útil que haya almacenado en el flujo, podemos suponer que existe una marca *EOF*. Esta marca no existe realmente, pero a efectos de nuestro programa podemos suponer que sí está puesto que cualquier petición de un carácter al flujo, éste responde con la marca *EOF*.

Suponga que en un fichero de texto “datos.txt” hemos almacenado el siguiente texto:

```
abc
```

Por lo que estamos diciendo, podemos suponer existe una marca especial de fin de fichero tras el contenido real del mismo, por lo que, virtualmente, para nuestro programa el contenido de este fichero sería:

```
abc<EOF>
```

De forma análoga a como se ha visto con anterioridad (véase 3.2.2 y 4.2.3), vamos a implementar un esquema de lectura adelantada en el siguiente ejemplo:

```
ifstream f("datos.txt");
int c;
c = f.get();
while (c!=EOF) {
    cout << (char)c << endl;
    c = f.get();
}
```

La ejecución de este programa nos presenta el siguiente resultado:

```
Consola
a
b
c
```

Observe que se ha declarado la variable `c` de tipo `int` en lugar de `char` como podría parecer razonable. El motivo es que la marca *EOF* es un valor que no puede coincidir con el de ningún otro carácter, es decir, es un valor fuera del rango [0,255] y por tanto, la función `get` debe devolver un dato que permita representar todos los caracteres y además la marca *EOF* que podría estar fuera de ese rango.

Ejercicio 10.9 — Contar vocales. Haga un programa que lea un texto desde un fichero y que al finalizar muestre el número total de vocales que ha leído. El contenido del fichero es arbitrario.

Además de esta marca, existe otra posibilidad para detectar el final de un flujo: la operación `eof()`. Esta operación devuelve `true` si se ha llegado a leer la marca *EOF* y `false` en caso contrario. Por ejemplo:

```
ifstream f("datos.txt");
int c;
c = f.get();
while (!f.eof()) {
    cout << (char)c << endl;
    c = f.get();
}
```

Ejercicio 10.10 — Media de datos. Haga un programa que lea números desde un fichero de texto y que muestre en la salida estándar la media de todos. El fichero puede contener un número arbitrario de números.

Deberá pensar cuál es la mejor forma de determinar si hemos llegado al final del fichero:

1. Comprobar si se ha leído *EOF*.
2. Comprobar si hemos llegado al final usando *f.eof()*.

Observe que en el ejemplo previo, tras leer la letra '*c*' del fichero, *f.eof()* devuelve **false**. Cuando se intenta leer algo tras la letra '*c*' (es decir, cuando se lee *EOF*), es cuando la operación *f.eof()* devolverá **true**.

Ejercicio 10.11 — Código erróneo. Ejecute el siguiente código y explique el resultado obtenido. Pruébelo con el fichero "datos.txt" usado en el ejemplo de esta sección.

```
ifstream f("datos.txt");
char c;
while (!f.eof()) {
    c = f.get();
    cout << c << endl;
}
```

10.4 Control de errores en los flujos

Al trabajar con flujos es posible que se produzcan errores. Como ya hemos podido comprobar en múltiples ocasiones, si hacemos la lectura de un dato de tipo entero y escribimos un carácter, la lectura falla. El efecto que hemos observado hasta ahora es que la variable que se debería haber leído tiene un valor sin determinar y que las sucesivas operaciones de lectura fallarán una tras otra.

Los flujos mantienen internamente información para gestionar estas situaciones. En particular disponen de variables lógicas que se activan o desactivan en función de si las operaciones tienen éxito o no. Aunque tenemos varias posibilidades, la más usada para comprobar si hay algún tipo de error es la operación *fail()*. Veamos un ejemplo:

```
int x;
cout << "Escriba un número: ";
cin >> x;
if (cin.fail())
    cout << "El número no se ha leído bien" << endl;
else
    cout << "El número es " << x << endl;
// Si hubo fallo no podemos hacer nuevas lecturas desde cin
```

De forma equivalente podemos usar esta otra sintaxis:

```
if (!cin)
```

Recuperación del estado de error

En una situación como la del ejemplo anterior, si a continuación tenemos más código de entrada de datos, éste no funcionará puesto que el flujo ha quedado inutilizado al fallar la lectura. Debemos tener la posibilidad de, una vez detectado un error, subsanarlo y permitir que siga la ejecución con normalidad. Para modificar el estado de un flujo de manera que se eliminen el indicador de error, se usa la operación *clear()*:

```
int x;
cout << "Escriba un número: ";
cin >> x;
if (cin.fail()) {
    cout << "El número no se ha leído bien" << endl;
    cin.clear();
} else
    cout << "El número es " << x << endl;
// Ahora podemos hacer nuevas lecturas desde cin tanto si hubo fallo como si no
```

Ejercicio 10.12 — Entero en intervalo. Escriba una función que lea y devuelva, desde la entrada estándar, un número entero en un intervalo [a,b]. Los valores a y b serán datos de entrada a la función. La función deberá comprobar también que el usuario haya escrito realmente un número además de que esté dentro del intervalo pedido.

10.4.1 Control de errores en ficheros

En el caso de los ficheros hay que tener especial cuidado con la gestión de errores en la apertura de los mismos. Tras la apertura es muy importante comprobar que ésta ha tenido éxito. Veamos un ejemplo:

```
ifstream f;
f.open("fichero.txt");
```

```

if (!f) { // Equivale a if (f.fail())
    cerr << "Error en apertura de fichero" << endl;
} else {
    cerr << "Éxito en apertura ..." << endl;
    ... // Lectura de datos desde el fichero
    f.close();
}

```

Observe que en el caso de que la apertura de un fichero falle, no es posible hacer la operación `close()`. Esta sólo debe hacerse sobre flujos que no tienen fallos.

Ejercicio 10.13 — Media con filtro. Reescriba el ejercicio 10.10 para considerar la posibilidad de que el fichero de datos contenga caracteres no numéricos. En caso de encontrar algo que no sea un número entero, el programa deberá descartarlo y seguir leyendo datos. Por ejemplo, si el fichero de datos tiene el siguiente contenido:

```

23 45
54.78 xxcc 10xx
a20

```

El resultado sería la media de 23, 45, 54, 78, 10 y 20 (que es 38.333333).

10.5 Reutilización de flujos

Como se ha dicho antes, una vez cerrado un fichero no es posible transferir datos. El siguiente código es incorrecto por este motivo:

```

ofstream f("fichero.txt");
f << "Texto a fichero" << endl;
f.close();
f << "Esto no se puede enviar" << endl;

```

Lo que sí se puede hacer tras cerrar un fichero es reutilizar el flujo que se ha liberado para transferir datos de nuevo hacia otro fichero o incluso hacia el mismo. Veamos un ejemplo:

```

ofstream f("fichero.txt");
f << "Texto a fichero" << endl;
f.close();
f.open("otrofichero.txt");
f << "Esto se guarda en otro fichero" << endl;
f.close();

```

Ejercicio 10.14 — Suma de datos. Realice un programa que lea datos numéricos de varios ficheros y que calcule la suma de todos ellos. Tanto el número de ficheros como el número de datos contenido en cada fichero es arbitrario. El programa solicitará el nombre de un fichero, a continuación lo leerá y pasará a pedir el nombre del siguiente fichero. Repetirá este ciclo hasta que el usuario indique que no hay más ficheros. Por ejemplo, suponiendo que tenemos estos tres ficheros:

- fichero1.txt que contiene los números 3 6 1
- fichero2.txt que contiene los números 9 5 78 45 -56 3
- fichero3.txt que contiene los números 0 44 -5 23

La ejecución podría ser similar a esta:

```

Consola
Fichero (@ para finalizar): fichero1.txt
Fichero (@ para finalizar): fichero2.txt
Fichero (@ para finalizar): fichero3.txt
Fichero (@ para finalizar): @
La suma es: 156

```

10.6 Otras operaciones sobre flujos

Los flujos disponen de muchas otras operaciones. En esta sección se explican algunas que son utilizadas con cierta frecuencia. Si necesitamos extraer caracteres de un flujo sin necesidad de almacenarlos ni procesarlos podemos usar `ignore()`:

```

ifstream f;
...
// Descarta el siguiente carácter de cin
cin.ignore();

```

```
// Descarta 10 caracteres de f (o menos si no hay 10)
f.ignore(10);

// Descarta o bien 20 caracteres o bien hasta encontrar la letra 'x' (lo primero que ocurra)
cin.ignore(20,'x'); // Descarta
```

Ejercicio 10.15 — Media del expediente. Realice un programa que lea desde la entrada estándar la lista de notas de un alumno. Cada entrada de esta línea consiste en una línea de texto con: identificador de la asignatura (4 caracteres), nota del expediente, lista de notas con las calificaciones de cada convocatoria. Siempre se usará un único espacio como separador en una línea y un retorno de línea para separar asignaturas entre sí. Por ejemplo:

```
ALG1 6.7 Jun12 3.4 Sep12 4.4 Jun13 6.7
PRG1 8.4 Feb12 3.1 Sep12 8.4
HIS2 7.2 Jun13 7.2
GEO3 5.5 Jun13 1.1 Sep13 3.2 Jun14 5.5
```

Haga un programa que lea estos datos y que muestre, al final, la nota media del expediente calculada como la nota media de las calificaciones finales de cada asignatura. Para este ejemplo el resultado sería 6.95.

Otras funciones útiles son las que permiten realizar una exploración previa del flujo sin extraer datos de él. En particular la operación `peek()` devuelve el siguiente carácter del flujo pero no lo extrae por lo que la siguiente operación de lectura que se haga volvería a leerlo. Por ejemplo, en el siguiente código podemos prever si lo que viene a continuación es un número o no y así usar el tipo de variable más adecuada para el almacenamiento:

```
if (isdigit(cin.peek())) {
    int numero;
    cin >> numero;
    cout << "El número es: " << numero << endl;
} else {
    char car;
    cin >> car;
    cout << "El carácter es: " << car << endl;
}
```

En lugar de `peek()` también podemos hacer uso de la operación `unget()` combinada con `get()` para obtener un efecto similar. Esta operación devuelve el último carácter leído al flujo:

```
int c=cin.get();
if (isdigit(c)) {
    int numero;
    cin.unget(); // Devolvemos el primer dígito
    cin >> numero;
    cout << "El número es: " << numero << endl;
} else {
    char car= c;
    cout << "El carácter es: " << car << endl;
}
```

Ejercicio 10.16 — Media de calificaciones. Realice un programa que lea las calificaciones de un curso para un alumno dado y que muestre su media. El programa irá leyendo el nombre de cada asignatura seguido de su calificación. Por ejemplo, dada la siguiente entrada:

```
Informática I 8.5 Geometría 9.3 Análisis Matemático 6.25 Historia de las matemáticas
8.75 Cálculo numérico 9.45
```

el programa dirá que la calificación media es de 8.45

Para realizar el ejercicio tenga en cuenta las siguientes apreciaciones:

- El nombre de las asignaturas puede tener una o más palabras.
- El nombre de las asignaturas no contendrá caracteres numéricos.
- La separación entre nombres de asignaturas y calificaciones puede ser cualquier separador válido (secuencias de uno o más espacios en blanco, tabuladores o retornos de línea).
- El número de asignaturas es arbitrario.

Puede realizar el ejercicio para que lea los datos desde la entrada estándar o desde un fichero de texto, en cuyo caso deberá solicitar el nombre del mismo al usuario.

10.7 Flujos y funciones

Los flujos son tipos de datos muy especiales, puesto que son gestores que nos sirven para manejar un recurso externo bastante complejo mediante una forma abstracta bastante simplificada. Es muy importante conocer exactamente cómo relacionar flujos y funciones.

10.7.1 Copia de flujos

Una particularidad de los flujos, es que no pueden copiarse. Es decir, si tenemos un objeto de tipo flujo, este no se puede copiar a otra variable del mismo tipo. Un flujo está asociado a un dispositivo que es único, si tuviésemos dos objetos en el programa asociados a un mismo dispositivo tendríamos problemas. Imagine que tenemos dos flujos leyendo a la vez de un fichero o dos flujos enviando datos a la vez a una impresora. El resultado sería impredecible o, al menos, muy difícil de gestionar. Por este motivo, cuando trabajamos con objetos de tipo flujo debemos tener presente que:

- No se puede asignar flujos entre sí.
- No se pueden pasar por valor a una función. Aunque sí se pueden pasar por referencia.
- No se pueden devolver con `return` de una función.

10.7.2 Uso de flujos “genéricos”

Suponga que necesita una función para debe recibir como parámetro un flujo para leer datos de él y devolver su suma. De acuerdo a lo indicado en la sección 10.7.1 esta debería escribirse así:

```
double Sumar(ifstream &f)
{
    double suma=0.0;
    double dato;
    f >> dato;
    while (!f.eof()) {
        suma += dato;
        f >> dato;
    }
    return suma;
}
```

Ahora imagine que le piden que realice la misma tarea pero sobre datos leídos desde la entrada estándar `cin`. Deberíamos copiar el código y sustituir `f` por `cin`. En esencia el código es el mismo. Sin embargo esto es un mal diseño ya que C++ permite que cuando los parámetros formales de las funciones son de tipo `istream` (o bien `ostream`) éstas puedan llamarse con parámetros actuales tanto de tipo `istream` como `ifstream` (lo mismo con `ostream` y `ofstream`). Es decir, si la cabecera de la función anterior fuese la siguiente:

```
double Sumar(istream &f)
```

podríamos realizar estas llamadas:

```
double s1;
cout << "Escriba datos en la entrada estándar: ";
s1 = Sumar(cin);
cout << "Suma desde cin: " << s1 << endl;

double s2;
ifstream f("datos.txt");
s2 = Sumar(f);
cout << "Suma desde fichero: " << s2 << endl;
```

Ejercicio 10.17 — Media de un flujo. Reescriba el ejercicio 10.13 para que al comienzo pregunte al usuario si desea leer datos desde la entrada estándar o desde un fichero. Deberá implementar una función que haga la lectura y el cálculo. En la función `main()` será donde se invoque a dicha función pasándole como argumento el flujo desde el que ha de leer. En caso de que la lectura se haga desde un fichero ¿podría hacerse la apertura dentro de la función?

10.8 Algunas recomendaciones finales

En las secciones anteriores se han revisado los aspectos más relevantes para entender cómo funcionan los flujos y tener una gama de herramientas básicas para poder resolver problemas con archivos de texto.

En esta sección añadimos una serie de recomendaciones finales que pueden resultar especialmente prácticas, ya que algunas herramientas son más habituales que otras y algunos esquemas o diseños son más fáciles de implementar que otros.

10.8.1 Lectura adelantada y flujos

Una peculiaridad de algunas operaciones sobre flujos es que devuelven como resultado el propio flujo además de haber realizado la operación de E/S correspondiente. Por un lado, esto nos facilita el encadenamiento de varias operaciones de E/S, pero por otra nos permite escribir el código de tratamiento de errores de una forma más compacta. Por ejemplo, el siguiente código:

```
int n;
cin >> n;
if (!cin)
    cerr << "Hubo un error en la lectura del número" << endl;
```

se puede reescribir así:

```
int n;
if (!(cin >> n))
    cerr << "Hubo un error en la lectura del número" << endl;
```

Con esta idea en mente es fácil ver código que procesa los datos de un flujo similar a éste:

```
int num, cont=0;
while (cin>>num)
    cont++;
cout << "Se han leído: " << cont << " números" << endl;
```

Observe que se sigue manteniendo el esquema de lectura adelantada pero se concentra en la condición del bucle `while` tanto la lectura como la comprobación de que el dato es válido. En este ejemplo se ha aprovechado también el hecho de que cuando se intenta leer de un flujo que no contiene datos, además de activar `eof()`, también se activa `fail()`. Por tanto, es muy frecuente procesar datos “hasta que se acaben” bien sea porque se llega al final del flujo o bien porque se ha producido un error en la lectura.

Ejercicio 10.18 — Parejas y múltiplos. Escriba un programa para procesar un archivo que contiene un número indeterminado de líneas, cada una de ellas conteniendo un par de números enteros positivos. El programa escribe aquellos pares que corresponden a dos valores enteros en los que el segundo es múltiplo del primero. Para resolverlo, haga que el programa lea dos nombres de archivo: el de entrada y el de salida.

Distinguiendo la causa de la parada

En ejemplos como el anterior vemos que se detiene el procesamiento cuando se alcanza el final del flujo o cuando se produce un error en la entrada. Podría ser interesante averiguar, después de la lectura, cuál de las dos situaciones se dió. Para ello se puede usar la operación `eof()`. Veamos un ejemplo:

```
int num, cont=0;
while (cin>>num)
    cont++;
cout << "Se han leído: " << cont << " números" << endl;
if (cin.eof())
    cout << "La lectura ha llegado al final" << endl;
else
    cout << "Se produjo un error en la lectura" << endl;
```

10.8.2 Ventajas e inconvenientes de “no-op”

Cuando un flujo de E/S falla entra en un modo de no operación. Las funciones de E/S no pueden funcionar hasta que se haya resuelto el fallo. Decimos que son operaciones *no-op* ya que el efecto de la llamada es que no hace nada. Como consecuencia, el programa continúa como si la operación se hubiera realizado, aunque nada ha cambiado. Por ejemplo, cuando un flujo de entrada está en modo de fallo e intenta leer un dato, la operación no se lleva a cabo y por tanto la variable que esté leyendo mantiene el valor anterior.

En principio podría pensar que es una forma de actuación incómoda ya que el programa puede continuar y llegar a un punto donde el error ya sea irrecuperable. Tal vez considere que un programa debería comprobar los errores de E/S y procesarlos en cuanto ocurren. Sin embargo, la implementación de las operaciones como *no-op* en caso de que haya un fallo nos permite escribir código más simple y eficaz. Por ejemplo, imagine que tenemos que leer 100 enteros desde la entrada estándar. Podemos hacer:

```
vector<int> v(100);

for (int i=0; i<100; ++i)
    cin >> v[i];

if (!cin)
    cerr << "Error de lectura" << endl;
```

Vemos que hemos separado claramente el código de lectura de la parte de comprobación de errores. Si quisieramos detectar el error en cuanto ocurre podríamos hacer el bucle como sigue:

```
for (int i=0; i<100 && cin; ++i)
    cin >> v[i];
```

Este ejemplo da lugar a un código menos legible al complicarse la condición de parada del bucle.

Note que en caso de tener un algoritmo más complicado, con múltiples lecturas desde un flujo, podemos posponer la comprobación de errores para la última parte, separando claramente el algoritmo de la gestión de errores.

No entienda este consejo como una invitación a la comprobación de errores sólo al final de las funciones. Simplemente es una facilidad más que nos permite escribir nuestro código de forma más simple y legible. No olvide que es importante comprobar los errores en el lugar adecuado. Por ejemplo, si el código anterior no estuviera escrito para 100 enteros, sino que tenemos que leer el tamaño de la entrada, puede ser importante comprobar que el número de datos es correcto antes de hacer que el vector tome el tamaño indicado. Por ejemplo, en el siguiente código:

```
int n;
cin >> n;

vector<int> v(n);
for (int i=0; i<n; ++i)
    cin >> v[i];

if (!cin)
    cerr << "Error de lectura" << endl;
```

Podemos obtener un fallo grave si el valor de n no es correcto o se lee erróneamente. En este código hemos supuesto que puede haber un error de lectura de uno de los n datos, pero asumimos que el valor de n se lee sin ningún problema.

Bucles de lectura de datos infinitos

Un error típico de los primeros programas es provocar un bucle infinito a causa de que un flujo ha quedado en estado de fallo. Es probable que ya se haya encontrado con esta situación a pesar de que los programas propuestos hasta ahora no han sido muy complejos. Por ejemplo, si ha escrito un programa simple que lee un dato hasta que es positivo, es posible que haya escrito algo como:

```
int valor;
do {
    cout << "Introduzca un valor positivo: ";
    cin >> valor;
} while (valor<=0);
```

que es un programa perfecto siempre que el dato que se encuentra en la entrada estándar sea un dato de tipo entero. Si en la entrada nos encontramos con una letra, el flujo entrará en estado de fallo y ninguna de las siguientes lecturas se podrá realizar. Como consecuencia, obtendrá un bucle infinito que no para de escribir el mensaje de petición del valor.

Ejercicio 10.19 — Lectura de enteros positivos. Modifique el código anterior para escribir un programa que lee un entero positivo y lo escribe en la salida estándar. El número se debe volver a solicitar mientras sea negativo. Tenga en cuenta que el programa termina escribiendo el número positivo o indicando que no se ha introducido un entero.

10.8.3 Evitar la lectura parcial de objetos

En múltiples ocasiones una operación de lectura se diseña para cargar el valor de un objeto compuesto. Por ejemplo: si leemos un número racional formado por un numerador y un denominador, tendremos que leer dos enteros, si leemos una fecha tenemos que leer día, mes y año, si leemos un número complejo habrá que leer dos números reales (partes real e imaginaria), etc. En estos casos es importante que las operaciones de lectura no se detengan con el objeto *medio modificado*.

Se debería evitar que una operación de lectura modifique un dato que no se ha leído por completo correctamente. Por ejemplo, imagine que tenemos el tipo *Fecha* compuesto por tres enteros:

```
struct Fecha {
    int dia;
    int mes;
    int anio;
};
```

y realizamos una función de lectura que carga una fecha suponiendo que se formatea como tres enteros consecutivos. En concreto, el código podría ser el siguiente (observe que no estamos incluyendo código para comprobar si la fecha es válida):

```
void Cargar (Fecha& f)
{
    cin >> f.dia;
    cin >> f.mes;
    cin >> f.anio;
}
```

Con esta función, podemos crear un programa que llama a la función para cargar una fecha y que comprueba si ha habido un error de lectura de la siguiente forma:

```
int main()
{
    Fecha f;
    // ...
    Cargar(f);
    if (!cin)
        cerr << "Error leyendo fecha" << endl;
    //...
```

En este caso podría haberse cargado la fecha parcialmente. Por ejemplo, tal vez se carga el día, pero al intentar leer el mes y el año no es posible y falla la lectura. El resultado es que la lectura ha fallado y el objeto *f* ha quedado *medio modificado*. En general, evitaremos realizar estos cambios parciales. La mejor solución consiste en modificar el objeto o no según haya habido éxito o no:

- Si la lectura tiene éxito: el objeto (completo) tendrá un nuevo valor.
- Si la lectura falla en algún componente: el objeto mantendrá su antiguo valor.

Por ejemplo, en nuestro caso podemos escribir la función:

```
bool Cargar (Fecha& f)
{
    bool ok=false;
    int d,m,a;
    cin >> d >> m >> a;
    if (cin) {
        f.dia= d;
        f.mes= m;
        f.anio= a;
        ok=true;
    }
}
```

```

    }
    return ok;
}

```

De esta forma al finalizar la llamada a la función sabremos si la lectura ha tenido éxito o no y el objeto leído tendrá un valor coherente. Observe que la devolución del valor booleano es simplemente por conveniencia, ya que el código de llamada podría comprobar el estado de *cin*. Más adelante estudiará otras posibilidades que harán el código más útil.

Ejercicio 10.20 Considere el problema de leer un objeto de tipo *Punto*. Un punto es un par (x,y) que representa un punto en el espacio. Escriba una función que lea un punto desde la entrada estándar. Esta función supondrá que el usuario escribe el punto con el siguiente formato: carácter '(', número real, carácter ',', número real, carácter ')'.

10.9 Ejercicios adicionales

Ejercicio 10.21 — Mezcla ordenada. Se dispone de dos ficheros de texto que contienen, cada uno de ellos, una lista de números enteros ordenados de menor a mayor. Haga un programa que mezcle ambos ficheros en un tercer fichero de forma que se mantenga el orden ascendente de los números.

Ejercicio 10.22 — Contabilidad empresarial. Una empresa dispone de varias cuentas corrientes. Se dispone de un fichero de transacciones en donde se van anotando los pagos y los cobros. El formato de dicho fichero consiste en una serie de apuntes tal que cada apunte consiste en:

- Identificación de la cuenta. Será de la forma Cxxx donde xxx es un número entero mayor o igual que uno.
- Nombre del proveedor. Es el nombre del comercio al que se realiza la transacción. Pueden ser una o más palabras (sin números).
- Lista de cargos. Una serie de números reales con los pagos realizados a esa cuenta y a ese proveedor. También pueden ser valores negativos en caso de que se trate de una devolución.

Un ejemplo podría ser el siguiente:

```
C1 Panadería Crujiente 243.40 132.75 347.3 C4 Carnicería La Roja 450 155.5 C2 Ultramarinos a lo lejos 23.50 -5.50 54.32 12.98 32.76 C1 Frutería La fibrosa 34.32 -4.50 12.56
```

En este ejemplo vemos que se ha hecho uso de tres cuentas (C1, C2 y C4) y como vemos puede haber varios apuntes de una misma cuenta. El primer apunte se refiere a la cuenta C1 y al proveedor "Panadería Crujiente" que ha hecho una serie de cargos (243.50, 132.75 y 347.30).

Haga una función que reciba como entrada un flujo y un vector de saldos de cada cuenta disponible en la empresa. Tras leer los datos, la función actualizará el vector de saldos de acuerdo a los apuntes contables. La función asumirá que el número de cuentas es el número de elementos que tenga el vector de saldos y estarán numeradas entre 1 y el número de elementos del vector. Por ejemplo, si la empresa dispone de cinco cuentas (C1, C2, C3, C4, C5) con estos saldos [2500, 1234.54, 80.43, 4879.12, 1434.10], al finalizar la lectura del fichero anterior nos quedarían estos saldos: [1734.17, 1116.48, 80.43, 4273.62, 1434.10].

Implemente un programa que le pregunte al usuario el número de cuentas de la empresa y el saldo de cada una. A continuación le preguntará si desea leer datos desde la entrada estándar o desde un fichero y, llamando a la función previa, calculará los nuevos saldos y los mostrará. En caso de que se lean desde un fichero el programa deberá solicitar el nombre del mismo.

Observaciones:

1. Puede haber un número arbitrario de apuntes.
2. El número de cuentas de la empresa también es arbitrario aunque la función asumirá que hay tantas como elementos tenga el vector de saldos.
3. El nombre de la empresa puede contener un número arbitrario de palabras.

Ejercicio 10.23 — Mini agenda. Considere un programa que debe procesar los datos de una serie de personas (*DNI*, *Nombre*, *Apellidos*, *Email*, *Edad*). Para ello debe definir una estructura para guardar los datos de una persona. Debe escribir un programa que muestre un menú de opciones al usuario para realizar las siguientes operaciones:

- Leer los datos de una persona desde la entrada estándar y almacenarlos al final del fichero.
- Leer el DNI de una persona desde la entrada estándar y buscarlo en el fichero. A continuación se mostrarán el resto de datos de esa persona en la salida estándar.
- Leer el DNI de una persona desde la entrada estándar y borrarla por completo del fichero.
- Mostrar un listado con todos los datos del fichero.

Ejercicio 10.24 — Expediente. Disponemos de un fichero como el del ejercicio 10.23 con datos de alumnos. Además, tenemos otra serie de ficheros, uno por asignatura, en los que se almacenan los DNI de los alumnos matriculados. Haga un programa que le pregunte al usuario el nombre del fichero con el listado de alumnos y el nombre del fichero de una asignatura y que cree un nuevo fichero que contenga el nombre de los alumnos matriculados en dicha asignatura.

A

Generación de números aleatorios

Introducción.....	97
Números pseudoaleatorios.....	97

A.1 Introducción

Muchos programas incluyen la generación automática de números aleatorios. Por ejemplo, un juego tiene que avanzar con eventos que simulen aleatoriedad a fin de obtener cierta variedad en el desarrollo de distintas partidas. Vamos a ver cómo podemos hacer que nuestros programas puedan generar dichos valores. Para ello presentamos las funciones que están disponibles en C++ “heredadas” desde el lenguaje C. En el último estándar de C++, puede encontrar nuevas y más potentes utilidades.

Básicamente, el problema consiste en que un programa tiene que ser capaz de generar una secuencia de números aleatorios tan larga como deseemos. Es decir, estamos interesados en obtener una serie de valores aleatorios:

$$x_0, x_1, x_2, \dots, x_i, \dots$$

Para obtener cada uno de estos valores, el lenguaje C++ ofrece la función `rand`, que devuelve un entero que está en el rango $[0, RAND_MAX]$, donde `RAND_MAX` es una constante predeterminada. El entero obtenido puede ser cualquiera de ese intervalo con igual probabilidad.

Lógicamente, si queremos un valor en otro intervalo podemos hacer una transformación lineal simple para cambiar esos valores. Lo más sencillo, y lo que recomendamos, es obtener un valor del rango $[0,1)$ con la siguiente expresión:

```
double aleatorio= rand() / (RAND_MAX+1.0);
```

y transformar dicho valor al rango deseado. Observe que hemos sumado 1.0, lo que permite que el resultado final no llegue a alcanzar el número uno. Además, con ese literal garantizamos que la división sea con valores de tipo double.

Para poder escribir una línea como la anterior, será necesario haber incluido el fichero de cabecera `cstdlib`.

A.2 Números pseudoaleatorios

Como sabemos, el ordenador es una máquina determinística, 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. Ahora bien, todos los números están determinados por la función $f(x)$, pero no sabemos cuánto vale el primer valor con el que comienza la secuencia. Toda la secuencia depende de él, de manera que si fijamos un valor concreto, siempre obtendremos la misma secuencia. Por ello, se denomina *semilla*.

Si nuestros programas usarán siempre la misma semilla, los números pseudoaleatorios que generaríamos serían siempre los mismos. Si queremos que distintas ejecuciones den lugar a distintas secuencias, es necesario cambiar de semilla en cada ejecución. Una forma muy simple de obtener distintas semillas es usar el valor del reloj del sistema. Note que si ejecutamos dos veces distintas un mismo programa, la semilla dependería del momento en que damos la orden de ejecución.

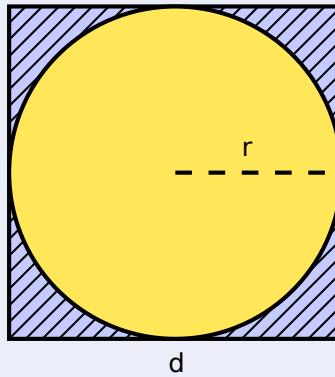
Para fijar una semilla, usamos la función `srand` de `cstdlib`, y para obtener un valor del reloj del sistema la función `time` del fichero de cabecera `ctime`. Un programa muy simple que genera tres valores aleatorios es el siguiente:

```
#include <cstdlib> // rand, srand
#include <ctime> // time
using namespace std;
...
int main()
{
    srand (time(0));

    double aleatorio1= rand() / (RAND_MAX+1.0);
    double aleatorio2= rand() / (RAND_MAX+1.0);
    double aleatorio3= rand() / (RAND_MAX+1.0);
}
```

Observe que la semilla sólo hay que fijarla al principio del programa (una única vez), y a partir de ese momento, se puede usar `rand` tantas veces como se desee para obtener nuevos valores pseudoaleatorios.

Ejercicio A.1 — Aproximando π . Supongamos un cuadrado de lado d con un círculo inscrito de radio $r = d/2$, tal como vemos en la figura. Sabemos que el área de un círculo viene dada por la expresión $S = \pi r^2$ mientras que la del cuadrado es $C = (2r)^2$.



Para aproximar el valor de π se propone realizar el siguiente experimento. Generar puntos aleatorios, uniformemente distribuidos, dentro de la superficie del cuadrado, y contar cuántos de ellos caen dentro del círculo y cuántos fuera. La probabilidad de que un punto caiga dentro del círculo, vendrá dada por la proporción entre el área del círculo y el área del cuadrado, es decir:

$$\frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

Como los puntos que lanzamos dentro del cuadrado se distribuyen de manera uniforme, la proporción de puntos que caen dentro del círculo debería mantenerse según la ecuación anterior. O sea, que si hemos dibujado N_{Tot} puntos y N_{Circ} de ellos están dentro del círculo, debe cumplirse que:

$$\frac{N_{Circ}}{N_{Tot}} = \frac{\pi}{4}$$

Este valor es una aproximación al número π que será más exacta cuanto mayor sea el número de puntos generados. Haga un programa que permita llevar a cabo esta simulación.

Ejercicio A.2 — Cifrado de texto. Una técnica muy simple para cifrar textos y hacerlos irreconocibles a simple vista es el cifrado César. Consiste en coger cada letra y cambiarla por otra que esté N posiciones a su derecha (o izquierda) con respecto a la ordenación alfabética. Por ejemplo, si consideramos $N=3$, cada vez que encontrremos la letra "d" la sustituiremos por la letra "g" (que está 3 posiciones a su derecha). En caso de encontrar una letra del final del alfabeto, consideraremos que este es cíclico. Por ejemplo, la letra "y" se sustituirá por la letra "b". De esta forma, dado un texto cifrado bastaría saber el valor de N para descifrarlo.

A modo de ejemplo, suponiendo un valor $N=3$, el texto "Hola" se transformaría en el texto "Krod". Descifrar el texto sería trivial conociendo N . Hay técnicas criptográficas muy sencillas que permiten descifrar este tipo de mensajes simplemente haciendo un análisis de la frecuencia de aparición de las letras (o de patrones de letras) y comparándola con la frecuencia teórica de uso de cada letra según el idioma en el que esté escrito el texto.

Una técnica más compleja podría ser cambiar el valor de N para cada letra que se está cifrando. Es decir, podríamos tener una serie de valores de N (uno por cada letra a cifrar). Por ejemplo, suponiendo los valores de N 4, 2, 5, 3 y el texto "Hola", el resultado de cifrarlo sería "Lqqd". Observe que en el texto cifrado se podrían incluso repetir letras sin que eso signifique que se correspondan con una misma letra en el texto original.

El problema que se nos plantea en este caso es que hay que disponer de tantos valores de N como letras tenga el texto a cifrar, y ese dato normalmente será desconocido a priori. En lugar de tener preestablecidos esos valores podemos considerar disponer de una secuencia de valores aleatorios que se van generando conforme se van necesitando. Además, debemos tener la posibilidad de volver a generar la misma secuencia en caso de que necesitemos descifrar el texto. Para ello podemos hacer uso de los generadores de números aleatorios sabiendo que, dada una semilla, un determinado método siempre va a generar la misma secuencia. Por tanto, bastaría conocer la semilla para descifrar un texto. Como consecuencia sería más difícil romper el código.

Implemente un programa que permita cifrar o descifrar un texto. Además del texto, el programa deberá preguntar al usuario la semilla que se ha usado para el cifrado.

B

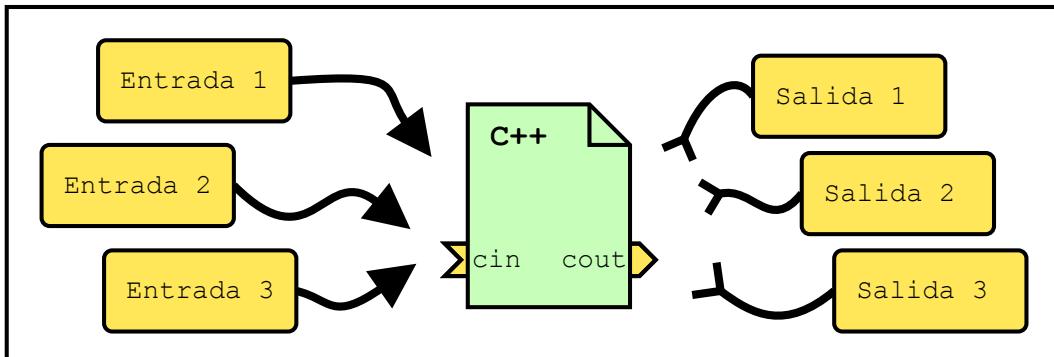
Redirección de Entrada/Salida

Introducción	101
Ejecuciones con muchos datos.....	103
Almacenamiento externo	104
Añadiendo en lugar de sustituyendo	
Redirección de E/S simultánea.....	104
Redirección de cerr	105
Encauzamiento	105

B.1 Introducción

Los problemas que hemos desarrollado han leído los datos desde la entrada estándar (`cin`) y han escrito los resultados en la salida estándar (`cout`). Hasta ahora, los datos de entrada han sido escasos, y hemos usado el teclado para introducirlos. Hemos asociado la idea de leer desde `cin` con la entrada de datos desde el teclado, aunque realmente no es así.

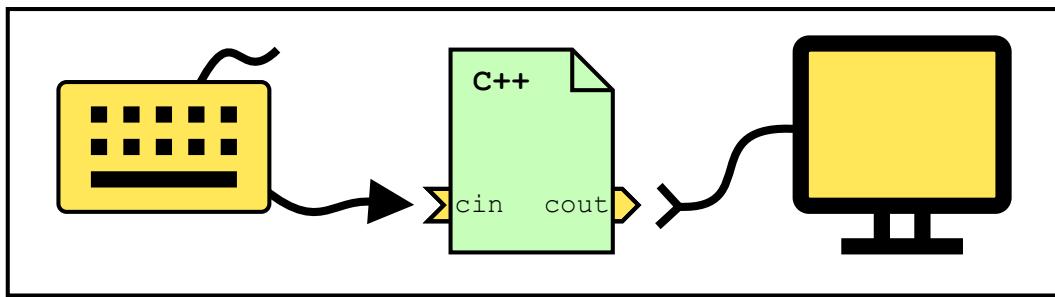
Cuando el programa comienza, se establece un canal de entrada de caracteres desde alguna fuente externa hasta la entrada estándar, así como un canal de salida de caracteres por la salida estándar. El programa no sabe nada acerca de la fuente o destino de los caracteres de E/S. Simplemente, cuando quiere leer un dato solicita caracteres a `cin` –para extraer el dato– o si quiere escribirlo los envía por `cout` –para generar el dato–. Cuando el sistema operativo ejecuta el programa, establece cuál es la fuente y cuál el destino de los datos. Es decir, “conecta una fuente de datos a `cin`” y “conecta `cout` a una salida de datos”:



Observe que el programa, por tanto, no es consciente de que existe un teclado. Simplemente sabe que cuando necesita un dato de entrada tiene que ir a `cin` y empezar a extraer los caracteres. En principio, parecía que se comportaba pacientemente mientras escribíamos lentamente los datos que nos solicitaba, pero lo cierto es que:

- Cuando el programa se detiene para que escribamos en el teclado, no lo hace porque tenemos que pulsar una serie de teclas de entrada, sino porque cuando intenta extraer el dato desde `cin`, no hay disponible ningún carácter.
- Cuando el programa sigue detenido mientras que pulsamos teclas, incluso la tecla *Retroceso*, mientras que editamos los datos de entrada, es porque sigue sin encontrar datos en `cin`. Mientras que no pulsemos *Enter*, no se confirmará la entrada, es decir, no se mandarán los caracteres introducidos por el flujo de entrada.

Puesto que la forma más común de uso de programas basados en consola es interactuar con el teclado y la pantalla, se establece por defecto el siguiente esquema de conexión para la E/S estándar (`cin` y `cout`):



Lógicamente, era de esperar que la lectura no estuviera asociada al teclado, pues seguramente el lector ya habrá realizado alguna entrada de datos mediante *copiar/pegar* en la consola. Lo que si es más interesante, es tener en cuenta que la secuencia de caracteres que se pasan al programa se pueden dar sin tener que esperar a que los solicite el programa y que las pausas que realiza el programa son consecuencia de que no hay datos en la entrada, y no de que tengamos que interactuar con el teclado. Por ejemplo, imagine que escribimos un programa que lee tres valores reales:

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

int main() {
    double d1, d2, d3;

    cout << "Primer dato: ";
    cin >> d1;
    cout << "Segundo dato: ";
    cin >> d2;
    cout << "Tercer dato: ";
    cin >> d3;

    cout << "La suma es: " << d1+d2+d3 << endl;
}
```

Si ejecutamos este programa, podemos interactuar para introducir los datos conforme los va pidiendo, es decir, pulsando la tecla *Enter* cada vez que escribimos un dato:

```
Consola
Primer dato: 2.5<enter>
Segundo dato: 3<enter>
Tercer dato: 3.5<enter>
La suma es: 9
```

Las tres veces que se ha detenido el programa para que nosotros escribamos el dato, se ha encontrado con que en `cin` no había ningún carácter disponible. Cuando escribimos un dato, y pulsamos la tecla *Enter*, la consola envía todos los caracteres por el flujo hacia el programa, que los consume conforme los necesita.

Este comportamiento nos indica que el resultado del programa depende de la secuencia de caracteres de entrada, que determina los tres números que tiene que sumar. No depende del momento en que se introducen, ya que eso simplemente afecta que el flujo de entrada esté vacío y el programa tenga que esperar.

Por ejemplo, imagine que en lugar de darle los datos de esa forma, los damos en una única línea. La ejecución sería como sigue:

```
Consola
Primer dato: 2.5 3 3.5<enter>
Segundo dato: Tercer dato: La suma es: 9
```

Observe que hemos introducido tres datos separados por espacios. Al pulsar *Enter*, todos los caracteres se encolan para entrar en el programa. Conforme el programa va pidiendo datos, y los va encontrando en `cin`, los va consumiendo. Así, cuando lee el primero, presenta el segundo mensaje y al solicitar un nuevo dato a `cin` no tiene que detenerse, pues ya se encuentra el valor 3 en espera. De forma similar ocurre para el último dato. Como puede comprobar, el resultado es el mismo, pues se han leído los tres datos sin ningún problema.

Ejercicio B.1 — Tres datos en una línea. Copie el programa de suma de tres números en su entorno y ejecútelo para comprobar el resultado expuesto. Compruebe el comportamiento si introduce los tres valores con múltiples espacios, saltos de línea y tabuladores entre ellos.

Por otro lado, dado que se pueden introducir los tres datos en una misma línea, cambie el programa para que aparezca un único mensaje “Introduzca tres datos.” al principio, y presente la suma de los tres.

B.2 Ejecuciones con muchos datos

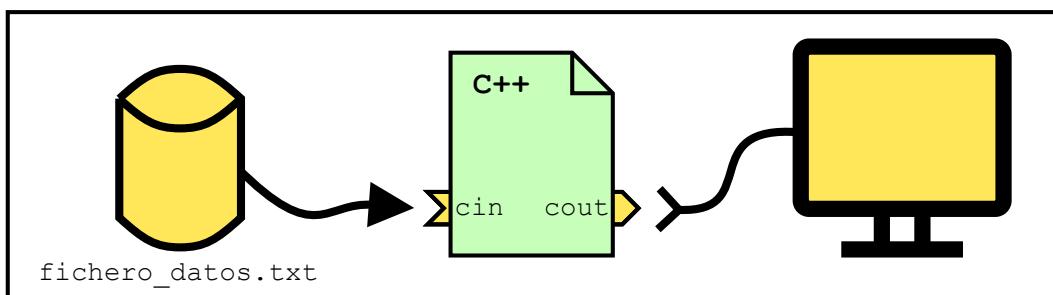
Cuando desarrollamos un programa y queremos ejecutarlo para probar su comportamiento con un número muy alto de datos de entrada, introducirlos una y otra vez por teclado en cada ejecución de prueba resulta demasiado tedioso. Para estos casos, lo mejor es tener los datos almacenados en un fichero y meterlos en el programa sin necesidad de reescribirlos. Como habrá imaginado, la solución más directa pasa por ejecutar el programa y cuando se detiene a la espera del primer dato, realizar una operación *copiar/pegar* para llevarlos desde el lugar donde los tengamos almacenados hasta la ventana de ejecución del programa. La consola recogerá los caracteres de esos datos como si los hubiéramos escrito con el teclado.

Otra alternativa es lanzar el programa indicando al sistema que conecte a `cin` el contenido de un fichero. Es decir, en lugar de lanzar el programa conectando el teclado como entrada y salida estándar, indicar que si el programa busca un dato en la entrada estándar, lo obtenga desde un fichero. Esta conexión se puede realizar fácilmente cuando lanzamos el programa desde el intérprete de órdenes¹. La sintaxis para lanzar un programa es:



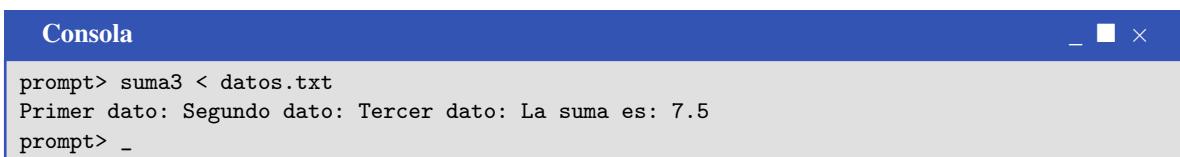
```
Consola
prompt> programa < fichero_datos.txt
```

donde vemos que hemos escrito el nombre del programa ejecutable, seguido por el carácter '`<`' y finalizando con el nombre del fichero de datos de entrada. De esta forma, ahora el esquema que muestra el flujo de datos de nuestro programa sería el siguiente:



Es importante tener en cuenta que este fichero de datos contiene los caracteres de entrada en formato ASCII. Por ejemplo, no será válido un archivo de tipo "doc" escrito con *Microsoft Word*, ya que estos archivos contienen múltiples datos como tamaño de márgenes, tipos de letra, etc. En nuestro caso no necesitamos un procesador de textos, sino un editor de texto plano. Por ejemplo, sería válido el editor de textos que integra el entorno de desarrollo que estemos usando o aplicaciones como el *block de notas* de Microsoft Windows o *kwrite* o *gedit* en entornos GNU/Linux.

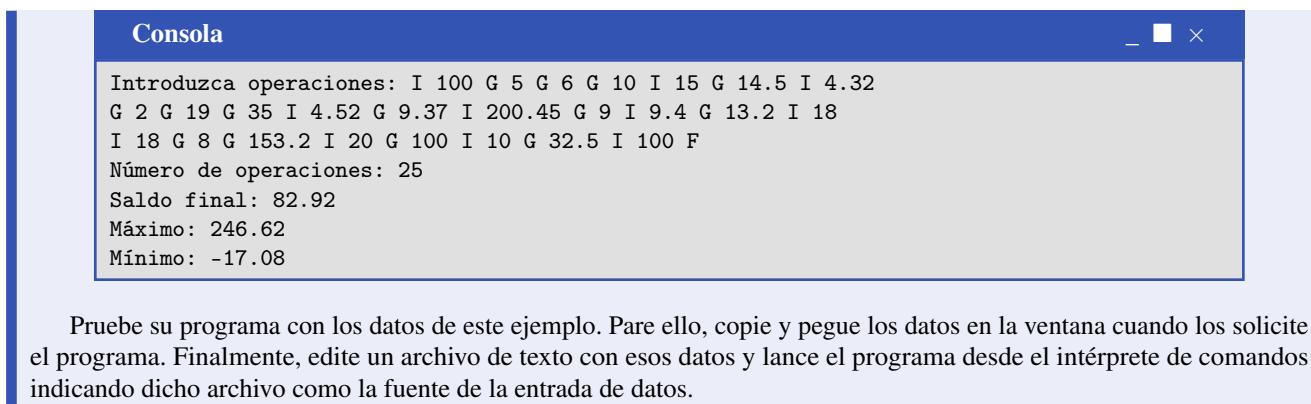
Para hacer una prueba vamos a usar el ejemplo previo de la suma de tres valores. Crearemos un fichero llamado "`datos.txt`" con un editor de texto en la misma carpeta donde tengamos nuestro ejecutable y le añadimos los 3 valores "2 2.5 3". Una vez salvado, abriremos un intérprete de comandos en nuestro sistema y cambiaremos el directorio activo a aquel en donde está nuestro programa ejecutable. Una vez allí ejecutaremos nuestro programa redirigiendo la entrada. La ejecución sería similar a la siguiente:



```
Consola
prompt> suma3 < datos.txt
Primer dato: Segundo dato: Tercer dato: La suma es: 7.5
prompt> _
```

Ejercicio B.2 — Cuenta bancaria. Se desea hacer un programa que calcule el máximo y mínimo saldo que se ha alcanzado a lo largo de la vida de una cuenta bancaria. Para ello asume que el saldo inicial es cero. El programa leerá un número indeterminado de pares -letra,valor- que indican ingresos (letra I) o gastos (letra G). El programa finaliza la lectura cuando lea una letra que no sea I/G, mostrando el valor máximo y mínimo alcanzados, así como el número de movimientos que se han realizado. Un ejemplo de ejecución es:

¹Los intérpretes de órdenes son programas que permiten dar instrucciones al sistema operativo usando para ello la consola. En sistemas de tipo Microsoft Windows existe un intérprete básico llamado `cmd.exe` (<http://es.wikipedia.org/wiki/Cmd.exe>). En sistemas de tipo GNU/Linux existen varios bastante extendidos como `bash` (<http://www.gnu.org/software/bash/manual/>) o `csh` (<http://linux.die.net/man/1/csh>).



Introduzca operaciones: I 100 G 5 G 6 G 10 I 15 G 14.5 I 4.32
G 2 G 19 G 35 I 4.52 G 9.37 I 200.45 G 9 I 9.4 G 13.2 I 18
I 18 G 8 G 153.2 I 20 G 100 I 10 G 32.5 I 100 F
Número de operaciones: 25
Saldo final: 82.92
Máximo: 246.62
Mínimo: -17.08

Pruébe su programa con los datos de este ejemplo. Pare ello, copie y pegue los datos en la ventana cuando los solicite el programa. Finalmente, edite un archivo de texto con esos datos y lance el programa desde el intérprete de comandos indicando dicho archivo como la fuente de la entrada de datos.

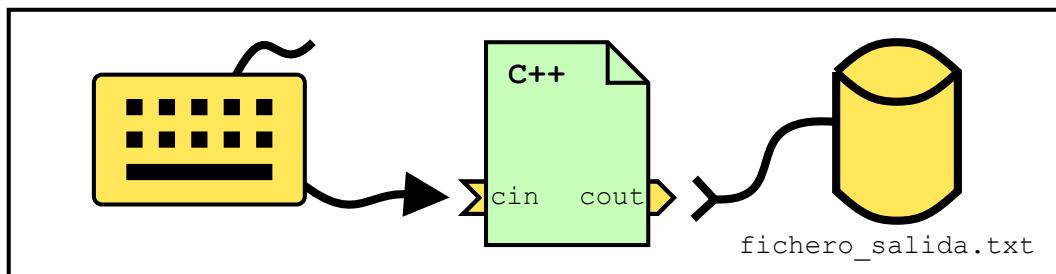
B.3 Almacenamiento externo

De forma análoga a como hemos redirigido la entrada de datos desde un fichero, también podemos redirigir la salida de datos a un fichero. El sentido de esto es tener un mecanismo sencillo para el almacenamiento persistente de datos sin necesidad de modificar el programa. Para ello usamos esta sintaxis:



```
prompt> programa > fichero_salida.txt
```

En este ejemplo el esquema de flujo de datos sería este:



Ejercicio B.3 Redirija la salida del programa del ejercicio B.2 para almacenar los datos en un fichero.

B.3.1 Añadiendo en lugar de sustituyendo

Observe que al redirigir la salida hacia un fichero, cada vez que ejecutamos el programa se borra el contenido previo de dicho fichero. Es decir, si el fichero al que enviamos los datos ya existía al comenzar la ejecución, es borrado y creado vacío antes de empezar a escribir en él.

Si deseamos conservar los datos que se habían almacenado previamente, usaremos la siguiente sintaxis:

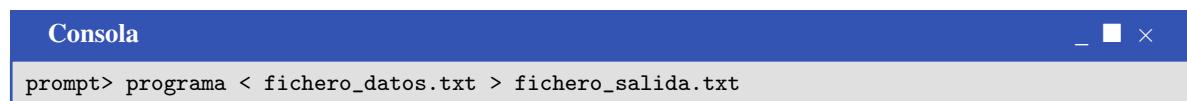


```
prompt> programa >> fichero_salida.txt
```

Observe que hemos duplicado el símbolo > para indicar que queremos añadir información al fichero y no perder lo que contenía.

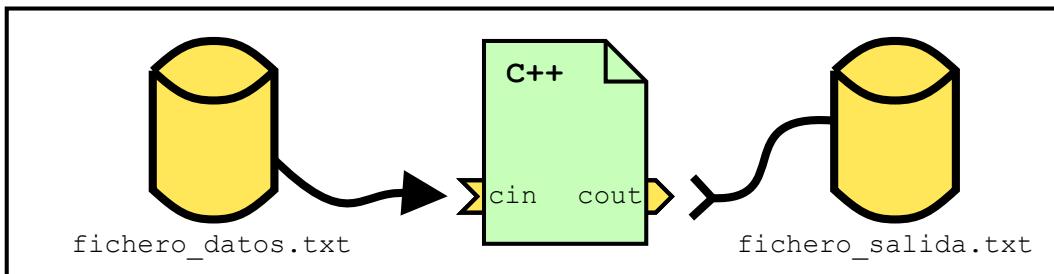
B.4 Redirección de E/S simultánea

La redirección simultánea de entrada y salida también es posible. De esta forma podemos automatizar por completo las ejecuciones y procesar datos de forma masiva almacenando los resultados obtenidos. Se hace combinando los casos expuestos anteriormente:



```
prompt> programa < fichero_datos.txt > fichero_salida.txt
```

En este ejemplo el esquema de flujo de datos sería este:



B.5 Redirección de cerr

Además del flujo estándar para salida de datos `cout` existe otro, también estándar, denominado `cerr` que está pensado para enviar mensajes de error y mantenerlos separados de los mensajes para comunicarse con el usuario de la aplicación. Este flujo se puede redirigir de forma independiente de `cout`. La sintaxis es la siguiente:

```

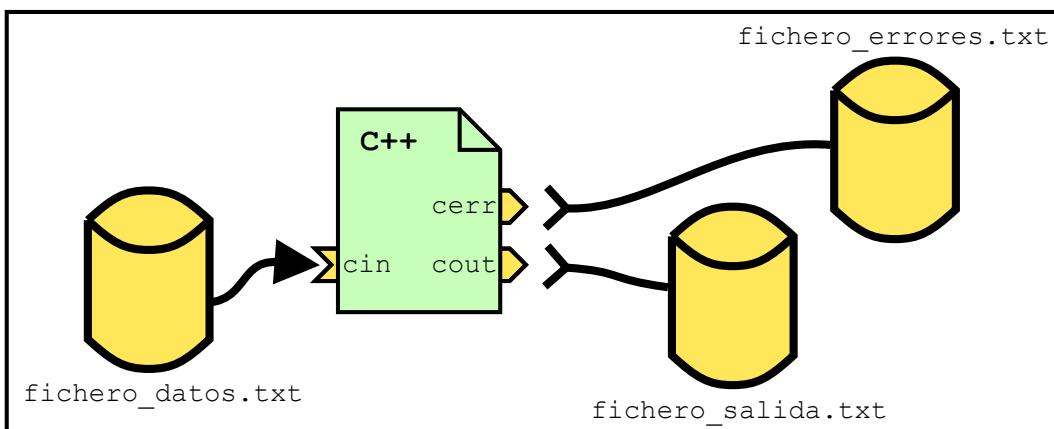
Consola
prompt> programa 2> fichero_errores.txt
  
```

El siguiente ejemplo muestra la redirección simultánea de varios flujos:

```

Consola
prompt> programa < fichero_datos.txt > fichero_salida.txt 2> fichero_errores.txt
  
```

En este ejemplo el esquema de flujo de datos sería este:



B.6 Encauzamiento

Considere una situación en la que los datos generados por un programa (*escritor*) se van a usar como entrada para otro (*lector*). Podríamos resolverlo usando lo que ya conocemos: almacenando los datos de salida de *escritor* en un fichero y luego usando ese fichero como entrada de *lector*:

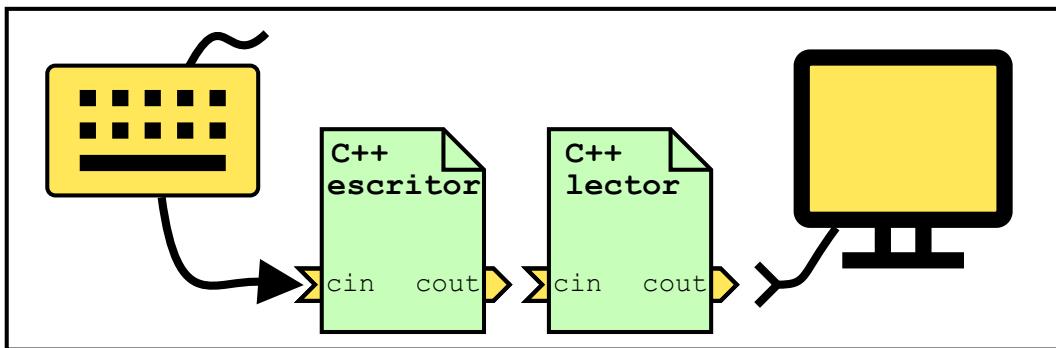
```

Consola
prompt> escritor > datos.txt
prompt> lector < datos.txt
  
```

Sin embargo, esto obliga a crear ese fichero aunque luego no se vuelva a necesitar. Disponemos de un mecanismo que soluciona este problema de una forma más efectiva y se denomina encauzamiento (o pipelining). Consiste en conectar la salida de un programa con la entrada de otro sin necesidad de generar ficheros intermedios:

```
prompt> escritor | lector
```

El flujo de datos en este ejemplo sería este:



Ejercicio B.4 Implemente un programa que lea una serie de números desde la entrada estándar y que muestre dicha secuencia en la salida estándar normalizada por la suma de todos ellos. La secuencia de entrada irá precedida por un número entero que dice cuántos números tiene la secuencia. Por ejemplo, si la secuencia de entrada es la siguiente:

4 3.3 7.8 -2.5 3.4

el resultado sería esta otra secuencia:

4 0.275 0.65 -0.208333 0.283333

Observe que la secuencia de salida va precedida por un número que indica el número de datos que se escriben a continuación.

A continuación escriba otro programa que lea una secuencia de datos y que calcule su media. Por ejemplo, si la secuencia de entrada es la siguiente:

4 0.275 0.65 -0.208333 0.283333

la salida de este programa sería esta otra:

1.416667

Para probar ambos programas genere desde un editor de texto un fichero con los datos de entrada al primer programa, use la salida de éste como entrada al segundo programa y finalmente escriba la media calculada en un fichero. Use la redirección y el encauzamiento.

Tablas

Tabla ASCII	107
Operadores C++	108
Palabras reservadas de C89, C99, C11, C++ y C++11	109

C.1 Tabla ASCII

En la figura C.1 se presenta la tabla de codificación ISO-8859-15 del alfabeto latino. Es similar a la ISO-8859-1. Esta codificación se distingue especialmente porque incluye el carácter correspondiente al euro.

Aunque es probable que la codificación ISO-8859-15 no sea la que esté usando en su sistema, la mayoría de los problemas que se resuelven en los cursos de programación asumen que es la codificación para los caracteres o secuencias de caracteres.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGC	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	í	¢	£	€	¥	Š	§	š	©	¤	«	¬	SHY	®	-
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	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 C.1
Tabla de codificación ISO-8859-15.

Observe que todos los caracteres especiales que existen en distintos idiomas y no en inglés están en la segunda parte de la tabla. Realmente, la tabla ASCII propiamente dicha es la primera mitad, mientras que la segunda es una extensión. Por ello, esta tabla no es la tabla ASCII, sino la tabla ASCII extendida ISO-8859-15.

En la práctica aceptaremos esta codificación para nuestras soluciones, ya que nos interesa especialmente el algoritmo a resolver sin entrar en detalles sobre los problemas de codificación. Note que podría tener problemas en la ejecución de algoritmos en español incluso si su sistema usa esta codificación. Por ejemplo, si quiere comprobar el orden de dos letras para ordenar una cadena, los caracteres con tilde están fuera del rango del alfabeto ''-'z''.

Si su sistema usa otras codificaciones como ISO-8859-1, windows-1252, o UTF-8, no tendrá problemas en ejecutar y comprobar el funcionamiento de sus algoritmos si evita el uso de la parte extendida, ya que en esas codificaciones los caracteres básicos se representan exactamente igual. Podríamos decir que realizamos soluciones que funcionan sin ningún problema solamente en inglés.

Más adelante es posible que tenga que resolver problemas para distintos lenguajes. Tendrá que consultar el tipo de codificación de su sistema y qué posibilidades tiene para resolverlo. Las soluciones pueden ir desde un tipo `char` con otra codificación hasta usar bibliotecas de funciones que resuelven sus problemas configurando el lenguaje usado.

C.2 Operadores C++

En la siguiente tabla se listan los operadores de C++. Los operadores situados en el mismo recuadro tienen la misma precedencia.

En general, para no tener ningún problema con la asociatividad, intente recordar que los operadores unarios y de asignación son asociativos por la derecha, mientras que los demás lo son por la izquierda.

Si revisa el estándar, podrá encontrar que realmente los unarios postfijo son de izquierda a derecha así como el operador condicional, aunque en la práctica no suelen crear incertidumbre.

Operadores de C++		
Operador	Nombre	Uso
::	Global	<code>::nombre</code>
::	Resolución de ámbito	<code>espacio_nombres::miembro</code>
::	Resolución de ámbito	<code>nombre_clase::miembro</code>
->	selección de miembro	<code>puntero->miembro</code>
.	Selección de miembro	<code>objeto.miembro</code>
[]	Índice de vector	<code>nombre_vector[expr]</code>
()	Llamada a función	<code>nombre_función(lista_expr)</code>
()	Construcción de valor	<code>tipo(lista_expr)</code>
++	Post-incremento	<code>valor_i++</code>
--	Post-decremento	<code>valor_i--</code>
<code>typeid</code>	Identificador de tipo	<code>typeid(type)</code>
<code>typeid</code>	... en tiempo de ejecución	<code>typeid(expr)</code>
<code>dynamic_cast</code>	conversión en ejecución	<code>dynamic_cast<tipo>(expr)</code>
	con verificación	
<code>static_cast</code>	conversión en compilación	<code>static_cast<tipo>(expr)</code>
	con verificación	
<code>reinterpret_cast</code>	conversión sin verificación	<code>reinterpret_cast<tipo>(expr)</code>
<code>const_cast</code>	conversión const	<code>const_cast<tipo>(expr)</code>
<code>sizeof</code>	Tamaño del tipo	<code>sizeof(tipo)</code>
<code>sizeof</code>	Tamaño del objeto	<code>sizeof(expr)</code>
++	Pre-incremento	<code>++valor_i</code>
--	Pre-decremento	<code>--valor_i</code>
~	Complemento	<code>~expr</code>
!	No	<code>!expr</code>
+	Más unario	<code>+expr</code>
-	Menos unario	<code>-expr</code>
&	Dirección de	<code>&valor_i</code>
*	Desreferencia	<code>*expr</code>
<code>new</code>	reserva	<code>new tipo</code>
<code>new</code>	reserva e iniciación	<code>new tipo(lista_expr)</code>
<code>new</code>	reserva (emplazamiento)	<code>new (lista_expr) tipo</code>
<code>new</code>	reserva (con inicialización)	<code>new (lista_expr)tipo(lista_expr)</code>
<code>delete</code>	destrucción (liberación)	<code>delete puntero</code>
<code>delete[]</code>	... de un vector	<code>delete[] puntero</code>
()	Conversión de tipo	<code>(tipo) expr</code>
<code>.*</code>	Selección de miembro	<code>objeto.*puntero_a_miembro</code>
<code>->*</code>	Selección de miembro	<code>puntero->*puntero_a_miembro</code>
*	Multiplicación	<code>expr*expr</code>
/	División	<code>expr/expr</code>
%	Módulo	<code>expr%expr</code>

continúa en la página siguiente

continúa de la página anterior

Operador	Nombre	Uso
+	Suma	<i>expr+expr</i>
-	Resta	<i>expr-expr</i>
<<	Desplazamiento a izquierda	<i>expr<<expr</i>
>>	Desplazamiento a derecha	<i>expr>>expr</i>
<	Menor	<i>expr<expr</i>
<=	Menor o igual	<i>expr<=expr</i>
>	Mayor	<i>expr>expr</i>
>=	Mayor o igual	<i>expr>=expr</i>
==	Igual	<i>expr==expr</i>
!=	No igual	<i>expr!=expr</i>
&	Y a nivel de bit	<i>expr&expr</i>
^	O exclusivo a nivel de bit	<i>expr^expr</i>
	O a nivel de bit	<i>expr expr</i>
&&	Y lógico	<i>expr&&expr</i>
	O lógico	<i>expr expr</i>
?:	expresión condicional	<i>expr?expr:expr</i>
=	Asignación	<i>valor_i=expr</i>
=	Multiplicación y asignación	<i>valor_i=expr</i>
/=	División y asignación	<i>valor_i/=expr</i>
%=	Resto y asignación	<i>valor_i%=expr</i>
+=	Suma y asignación	<i>valor_i+=expr</i>
-=	Resta y asignación	<i>valor_i-=expr</i>
<<=	Desplazar izq. y asignación	<i>valor_i<<=expr</i>
>>=	Desplazar der. y asignación	<i>valor_i>>=expr</i>
&=	Y y asignación	<i>valor_i&=expr</i>
^=	O exclusivo y asignación	<i>valor_i^=expr</i>
=	O y asignación	<i>valor_i =expr</i>
throw	Lanzamiento de excepción	<i>throw expr</i>
,	Coma	<i>expr ,expr</i>

Cuadro C.1
Operadores de C++

Estos operadores están incluidos en el estándar C++98. En C++11 aparecen tres más:

`sizeof..., noexcept, alignof`

C.3 Palabras reservadas de C89, C99, C11, C++ y C++11

Es interesante conocer las partes comunes del lenguaje C y C++. En esta sección presentamos la tabla C.2 con las palabras reservadas de las distintas versiones de ambos lenguajes. La parte más interesante se encuentra en los dos primeros bloques, donde se incluyen las palabras reservadas que estaban presentes en C cuando se creó C++. Dado que éste “heredó” gran parte de los contenidos de C, el segundo bloque de C++ se presenta como una adición a las del primero (comunes a ambos).

A pesar de ello, los últimos tres bloques presentan palabras que se han ido añadiendo en las sucesivas versiones de los lenguajes. Aunque todavía muchos programadores creen que C++ es un lenguaje ampliación de C, lo cierto es que los dos evolucionan de forma independiente.

Cuadro C.2

Palabras reservadas de C89, C99, C11, C++ y C++11.

Comunes a C(C89) y C++			
auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while
Adicionales de C++			
and	and_eq	asm	bitand
bitor	bool	catch	class
compl	const_cast	delete	dynamic_cast
explicit	export	false	friend
inline	mutable	namespace	new
not	not_eq	operator	or
or_eq	private	protected	public
reinterpret_cast	static_cast	template	this
throw	true	try	typeid
typename	using	virtual	wchar_t
xor	xor_eq		
Añadidas en C99			
_Bool	_Complex	_Imaginary	inline
restrict			
Añadidas en C++11			
alignas	alignof	char16_t	char32_t
constexpr	decltype	noexcept	nullptr
static_assert	thread_local		
Añadidas en C11			
_Alignas	_Alignof	_Atomic	_Generic
_Noreturn	_Static_assert	_Thread_local	

Bibliografía



Libros básicos (C++98)

- [1] Garrido, A. *Fundamentos de programación en C++*. Delta publicaciones. 2005.
- [2] Josuttis, N. *The C++ standard library. A tutorial and handbook*. Addison Wesley. 1999.
- [3] Stroustrup, B. *El lenguaje de programación C++ (ed. especial)*. Addison Wesley. 2002.
- [4] Deitel y Deitel, *C++ Cómo programar*. Sexta Edición. Prentice Hall. 2009.

Libros actualizados (C++11)

- [5] Garrido, A. *Fundamentos de programación con la STL*. Editorial Universidad de Granada. 2016.
- [6] B. Stroustrup, *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.
- [7] Nicolai Josuttis, *The C++ standard library. A tutorial and handbook (2nd edition)*. Addison-Wesley, 2012.
- [8] Deitel y Deitel, *C++ How to program (early objects version)*. Novena Edición. Prentice Hall. 2011.

Libros electrónicos (C++98)

- [9] B. Eckel, *Thinking in C++* (2^a edición). Prentice-Hall. 2000. Disponible en versión electrónica en <http://www.bruceeckel.com/>

Manuales de referencia

- [10] *C++ Reference*. Página web con una referencia bastante completa de los recursos disponibles en el lenguaje C++, incluyendo explicaciones de las aportaciones del último estándar. <http://en.cppreference.com/>
- [11] *C++ Reference*. Página web con una referencia bastante completa de los recursos disponibles en el lenguaje C++. <http://www.cplusplus.com/reference/>

Índice alfabético



A

anidamiento
 bucles, 22
 condicionales, 10
argumento, 48, 50
ASCII, 5
aserción, 59
at
 string, 38
vector, 30

B

búsqueda, 33
biblioteca, 3
bool, 13
bucle, 17
 anidamiento, 22
 condición compuesta, 23
contador, 21

C

cabecera
 archivos, 3
cadena de caracteres, 37
 declaración, 38
caracteres, 5
caso límite, 24
casting, 6
centinela, 18
cerr, 83
cin, 83
clog, 83
close, 88
concatenar, 41
condición compuesta, 11, 23
condicional, 9
 anidamiento, 10
constante, 3
contenedor
 vector como, 29
conversión entre booleano y entero, 13
conversión explícita, 6
conversión implícita, 52
cortocircuito
 evaluación en, 12
cout, 83

D

dinámico
 vector, 32
diseño
 patrones de, 17
diseño descendente, 54
do-while, 17

E

endl, 2
entero, 2
 como booleano, 13
eof, 88
erase, string y, 42
error de precisión, 4
error lógico, 4
estructura, 77
evaluación en corto, 12
expresiones, 1

F

fail, 89
fichero de cabecera
 cassert, 59
 ctype, 3, 5
 cmath, 3
 fstream, 86
find
 string y, 43
flujo, 83
 copia, 92
 flujo genérico, 92
for, 21
 bucle contador, 21
función, 47
función de biblioteca, 3
 assert, 59
 erase, 42
 exit, 58
 find, 43
 get, 40, 85
 getline, 38, 40, 85
 insert, 42
 isdigit, 5
 islower, 5
 pow, 3

put, 86
rand, 97
replace, 43
rfind, 43
sin, 3
sqrt, 3
strand, 98
substr, 42
time, 98
toupper, 5

G

gaussiana, 7
get, lectura de carácter, 40
getline, 38

H

hasta, bucles, 20

I

if doble (if/else), 9
if simple, 9
ifstream, 86
ignore, 90
incremento
 for e, 21
inicialización
 for e, 21
insert, string y, 42
intercambio, 51
ios::app, 87
istream, 92
iteración, 17

L

lectura
 acabada con centinela, 18
 adelantada, 19, 32
 de líneas, 39
lectura adelantada, 92

M

matriz, 29, 32
mediana, 34
menú, bucles do-while y, 18

mezcla

vectores y, 34

moldeado, 6

N

números aleatorios, 97

pseudoaleatorios, 97

O

ofstream, 86

open, 86

operación, 1

operador lógico, 11

ordenación, 33

ostream, 92

P

pair, 79

palabras reservadas, 109

parámetro, 48

parámetro actual, 50

parámetro formal, 48

paso por referencia, 50

paso por valor, 48

peek, 91

postcondición, 58

precondición, 58

prioridad, 2

pseudoaleatorio, 97

R

rand, 97

real, 2

recursividad, 73

caso base, 73

caso general, 73

puntos de salida, 74

redirección de E/S, 101

referencia constante, 65

replace, string y, 43

reservadas

palabras, 109

rfind

string y, 43

S

semilla, 97

size

string, 38

vector, 30

sobrecarga, 60

srand, 98

static_cast, 6

stream, 83

string, 37

+, 41

[], 39

at, 38

size, 38

struct, 77

substr, string y, 42

switch, 12

T

tabulador, 3

time, 98

tipo de dato, 1, 77, 79

top-down, 54

U

unget, 91

V

valor por defecto, 60

variable, 2

vector, 29

[], 31

at, 30

declaración, 30

mezcla, 34

size, 30

vector de vectores, 32

void, 49

W

while, 18