



# Proyectos con C++

## Práctica final

Antonio Garrido



ugr

Universidad de Granada

Departamento de Ciencias de la Computación  
e Inteligencia Artificial

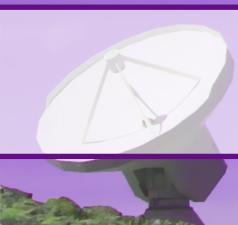
Copyright © 2015 Antonio Garrido

TODOS LOS DERECHOS RESERVADOS

*Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación*

UNIVERSIDAD DE GRANADA

*Febrero 2015*



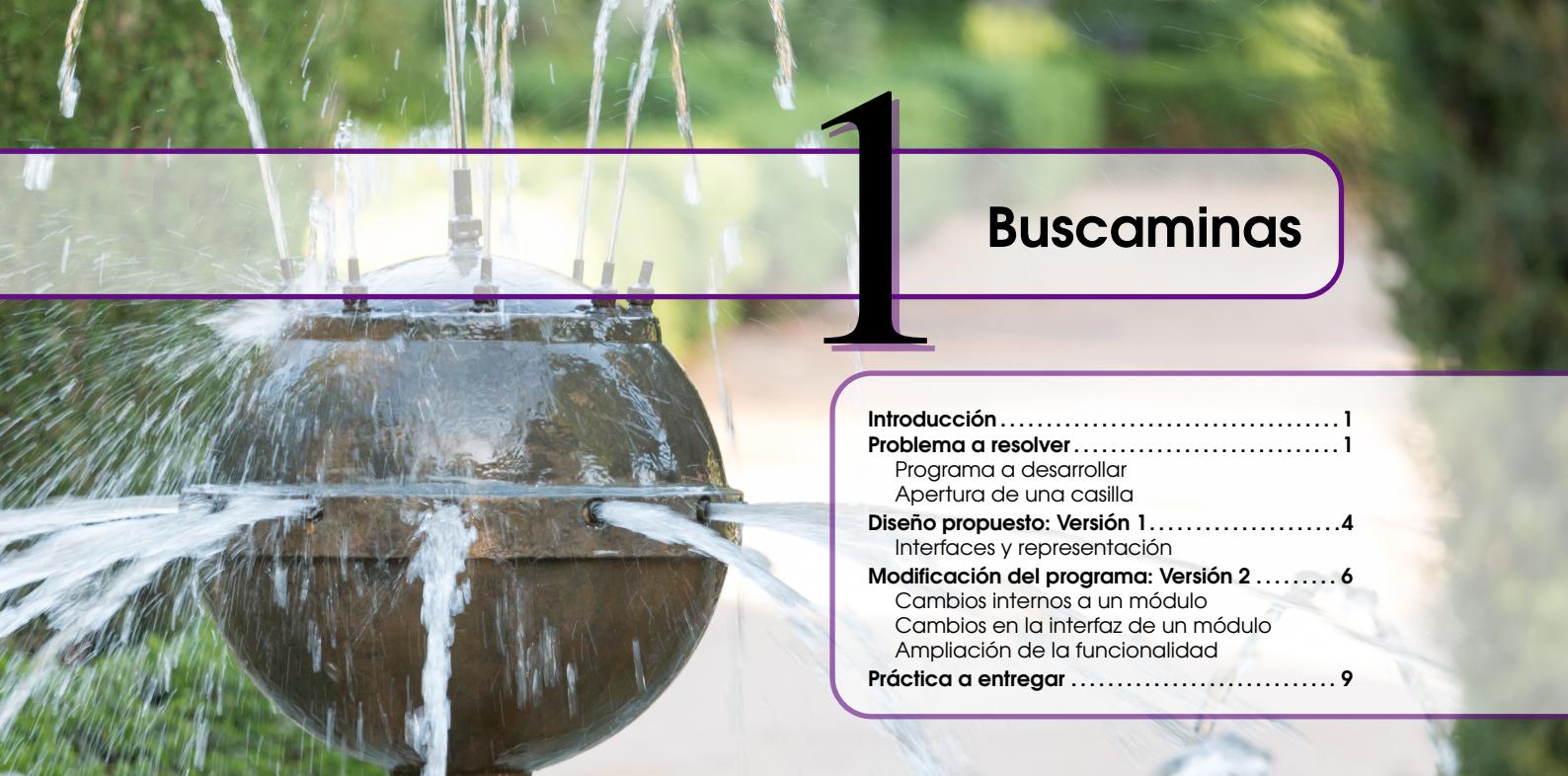
# Índice general

<b>1</b>	<b>Buscaminas</b>	1
1.1	Introducción	1
1.2	Problema a resolver	1
1.2.1	Programa a desarrollar	2
1.2.2	Apertura de una casilla	4
1.3	Diseño propuesto: Versión 1	4
1.3.1	Interfaces y representación	4
1.4	Modificación del programa: Versión 2	6
1.4.1	Cambios internos a un módulo	6
1.4.2	Cambios en la interfaz de un módulo	7
1.4.3	Ampliación de la funcionalidad	8
1.5	Práctica a entregar	9
<b>A</b>	<b>Generación de números aleatorios</b>	11
A.1	Introducción	11
A.2	El problema	11
A.2.1	Números pseudoaleatorios	12
A.3	Transformación del intervalo	13
A.3.1	Operación módulo	13
A.3.2	Normalizar a U(0,1)	13
<b>B</b>	<b>Tablas</b>	15
B.1	Tabla ASCII	15
B.2	Operadores C++	16
B.3	Palabras reservadas de C89, C99, C11, C++ y C++11	17
B.4	Manipuladores y funciones miembro para E/S formateada	17
B.4.1	Banderas y máscaras	18
B.4.2	Funciones miembro y manipuladores	19
	<b>Índice alfabético</b>	21



# 1

## Buscaminas



Introducción .....	1
Problema a resolver .....	1
Programa a desarrollar	
Apertura de una casilla	
Diseño propuesto: Versión 1 .....	4
Interfaces y representación	
Modificación del programa: Versión 2 .....	6
Cambios internos a un módulo	
Cambios en la interfaz de un módulo	
Ampliación de la funcionalidad	
Práctica a entregar .....	9

### 1.1 Introducción

En esta práctica se propone crear un programa para jugar al conocido juego *buscaminas*. Es un juego donde se usa una interfaz gráfica que facilita la visualización de las jugadas. Sin embargo, dado que estamos en un curso básico de C++ y queremos limitarnos al lenguaje, crearemos una versión para la consola. Lógicamente, este problema es un buen candidato para crear alguna solución con gráficos que permite, además, introducir nuevos y más avanzados diseños.

Los objetivos de este guión de prácticas son los siguientes:

1. Punteros y memoria dinámica:
  - Practicar con *cadenas-C*.
  - Practicar con una estructura basada en celdas enlazadas alojadas en memoria dinámica.
2. Diseño de clases avanzado:
  - Resolver un problema en el que creamos un nuevo tipo de dato que se alojará en la memoria dinámica.
  - Practicar con sobrecarga de operadores. En concreto, la sobrecarga del paréntesis junto con la devolución de referencias.
  - Practicar la sobrecarga de E/S.
3. Modificación de un programa. Generar una segunda versión de un mismo programa realizando modificaciones. El objetivo es entender mejor las consecuencias de una modificación, especialmente dependiendo de si cambiamos zonas ocultas o encapsuladas y código que forma parte de una interfaz. Para ello, la tarea de desarrollo se realizará en dos fases:
  - a) En la primera fase creamos un prototipo rápido que nos muestre el programa en funcionamiento.
  - b) En la segunda modificamos la primera versión. Estos cambios nos servirán para mejorar el programa, ampliar su funcionamiento, o simplemente para evaluar una implementación alternativa.

La solución del guón estará basada en el desarrollo de clases que se basan en tipos básicos del lenguaje, incluyendo punteros, *vectores-C*, *cadenas-C* y memoria dinámica. Por tanto, deberá evitar el uso de tipos de la *STL*.

### 1.2 Problema a resolver

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 habrá terminado sin éxito. Si no lo es, se abrirá la casilla y el resto de casillas del “entorno” (más adelante detallaremos esta idea).
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.

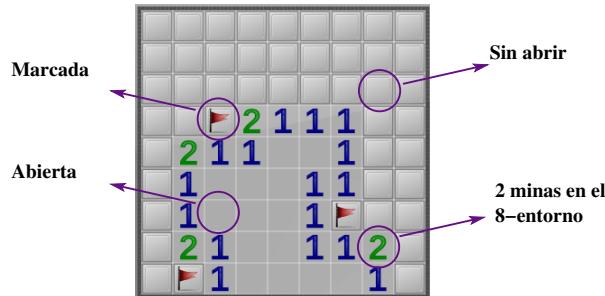
Para poder guiar al jugador de forma que pueda averiguar dónde se encuentran las minas, el juego ofrece pistas sobre dónde podrían estar las minas. En concreto, cuando se abre una casilla sin mina existen dos posibilidades:

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.

En la figura 1.1 se presenta gráficamente el estado de una partida del juego. Se pueden observar casillas abiertas, ocultas, y posiciones donde se ha determinado que habrá una mina.



**Figura 1.1**  
Distintos estados de las casillas del buscaminas.

### 1.2.1 Programa a desarrollar

En este proyecto 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:

1. Inicialmente el juego comienza 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.
2. 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 la letra '`a`' que indica la acción 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 será la letra '`m`' que indica la acción y los dos siguientes la fila y columna correspondientes. Si la casilla está sin marcar la pone como marcada, y en caso de que ya esté marcada elimina la marca.
3. 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.

#### Ejemplo de ejecución

El programa comienza pidiendo los parámetros del juego: las dimensiones del tablero y el número de minas que tendrá. La consola tendrá el siguiente aspecto:

```
Consola
prompt% ./buscaminas
Dime el tamaño del tablero (filas y columnas): 10 10
Dime el número de minas: 10
```

Tras leer los parámetros, el programa presenta el tablero y pregunta por la acción a realizar. Lógicamente, el tablero tendrá todas las casillas ocultas. Por ejemplo, podemos “dibujar” el tablero como sigue:

Consola

	0	1	2	3	4	5	6	7	8	9
0	*	*	*	*	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*	*	*
8	*	*	*	*	*	*	*	*	*	*
9	*	*	*	*	*	*	*	*	*	*

Dime acción y posición (fil y col): a 3 3

donde puede ver que en la última línea hemos indicado que queremos realizar una acción de apertura de la casilla en la fila 3 y la columna 3. El resultado se presenta a continuación:

Consola

	0	1	2	3	4	5	6	7	8	9
0	*	*	*	*	1					
1	*	2	1	1	1					
2	*	1								
3	*	1								
4	*	1	1	1	1	2	1	1		
5	*	*	*	*	*	*	*	1		
6	*	*	*	*	*	*	1	1		
7	*	*	*	*	*	*	1			
8	*	*	*	*	*	*	1	1	1	
9	*	*	*	*	*	*	1	1	1	*

Dime acción y posición (fil y col):

Observe que en esta situación:

- Se han abierto muchas casillas, no sólo la indicada. Esto es consecuencia de que en esa casilla no había nada.
- Se han presentado casillas con números. Cada uno de ellos, como sabemos, indica el número de minas en el 8-entorno.
- Podemos deducir la posición de alguna de las minas. Por ejemplo, en la posición fila 0 y columna 3, 5-6 o 9-9. Podemos marcar alguna de esas casillas con la orden '*m*'. El resultado sería el siguiente:

Consola

Dime acción y posición (fil y col): m 5 6

	0	1	2	3	4	5	6	7	8	9
0	*	*	*	*	1					
1	*	2	1	1	1					
2	*	1								
3	*	1								
4	*	1	1	1	1	2	1	1		
5	*	*	*	*	*	*	?	1		
6	*	*	*	*	*	*	1	1		
7	*	*	*	*	*	*	1			
8	*	*	*	*	*	*	1	1	1	
9	*	*	*	*	*	*	1	1	1	*

Dime acción y posición (fil y col):

Observe que hemos presentado el campo de minas incluyendo un signo de interrogación. Este signo indica que la casilla está marcada, es decir, que sospechamos que hay una mina en esa posición.

Por otro lado, si realizamos una acción de apertura en una posición con mina, podemos obtener lo siguiente:

```
Dime acción y posición (fil y col): a 8 1
El campo de minas era el siguiente:
 0 1 2 3 4 5 6 7 8 9
-----
 0 | X| 1| 1| X| 1|   |   |   |   |
 1 | 2| 2| 1| 1| 1|   |   |   |   |
 2 | X| 1|   |   |   |   |   |   |   |
 3 | 1| 1|   |   |   |   |   |   |   |
 4 | 1| 1| 1| 1| 1| 2| 1| 1|   |
 5 | 1| X| 1| 1| 1| X| 2| X| 1|   |
 6 | 1| 1| 1| 1| 1| 2| 1| 1|   |
 7 | 2| 2| 1|   | 1| 1| 1|   |   |
 8 | X| X| 1|   | 1| X| 1|   | 1| 1|
 9 | 2| 2| 1|   | 1| 1| 1|   | 1| X|
-----
Lo siento, has perdido.
```

Observe que el tablero presenta su contenido incluyendo los números y las posiciones de las minas. Efectivamente, en la posición 8-1 había unamina.

### 1.2.2 Apertura de una casilla

La parte más interesante, desde el punto de vista algorítmico, es probablemente la apertura de una posición o casilla. La dificultad del algoritmo 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. Más concretamente, la apertura de una casilla que no tiene mina puede dar lugar a dos casos:

- Que la casilla tiene alguna mina en su entorno. Por ejemplo, en la figura 1.1 aparecen con un número. La apertura consiste simplemente en marcarla como abierta. Por tanto, a partir de ese momento el dibujo del campo de minas incluirá este número.
- Que la casilla no tiene mina en su entorno. Por ejemplo, en la figura 1.1 aparece indicada como *Abierta*. La apertura de una casilla como ésta consiste en marcarla como abierta y abrir las de su entorno.

Note que la apertura de una casilla sin minas en su entorno es un algoritmo que no se detiene con la apertura de dicha casilla, sino que provoca la apertura de su entorno. Lógicamente, estas nuevas aperturas podrán encadenarse. De ahí que una simple acción haya abierto tantas casillas en el ejemplo anterior.

## 1.3 Diseño propuesto: Versión 1

El programa que resuelve el problema puede diseñarse creando una clase *CampoMinas* que encapsule el la representación y funcionamiento del juego, junto con otro módulo de alto nivel *BuscaMinas* que corresponde a la aplicación de consola. Sin embargo, dado que queremos encapsular el problema de la representación del tablero, vamos a proponer la creación de 3 módulos:

1. Módulo *Tablero*. Básicamente, consiste en una matriz que representa el estado del tablero.
2. Módulo *CampoMinas*. Encapsula el funcionamiento del juego. Lógicamente, contiene un objeto de tipo *Tablero*, e implementa las operaciones propias del juego.
3. Módulo *Aplicación*. Este módulo implementará el código necesario para interactuar con el usuario y operar sobre un objeto de tipo *CampoMinas*.

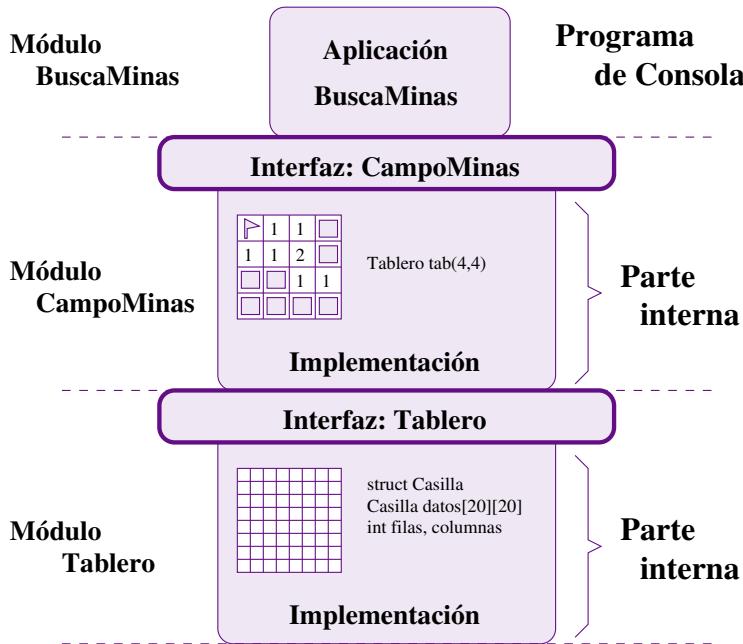
En la figura 1.2 puede observar la estructura que proponemos. Observe que hemos incluido algún apunte sobre los detalles de implementación.

### 1.3.1 Interfaces y representación

La mejor forma de entender los detalles del diseño propuesto es especificar con más detalle cuáles son las representaciones y las interfaces que se proponen. Concretamente, daremos algunas indicaciones para implementar dos clases: *Tablero* y *CampoMinas*.

#### Clase Tablero

El único objetivo que nos planteamos con esta clase es encapsular la estructura de datos que mantiene la matriz de casillas del juego. Para definirla, se propone crear una *matriz-C* fija de tamaño 20 por 20. Con esta matriz, podremos representar cualquier tablero que tenga hasta 20 casillas para cada dimensión.



**Figura 1.2**  
Tres módulos para implementar el juego *Buscaminas*.

Por otro lado, tenga en cuenta que el estado de una casilla puede definirse con tres valores booleanos: si tiene bomba, si está abierta y si está marcada. Para representarlo, crearemos una estructura con estos tres campos de forma que la matriz que define el tablero será una matriz de objetos de este tipo.

Como hemos visto en anteriores problemas, respecto a las operaciones que ofrece esta clase en la interfaz, se podrían incluir las siguientes:

- Consulta del número de filas del tablero.
- Consulta del número de columnas del tablero.
- Consulta del elemento, es decir, la estructura de tres valores almacenada en una posición determinada.
- Modificación del valor de una determinada posición, es decir, de la tripleta que define el estado.

Por lo tanto, el resultado es un contenedor de objetos que son tripletas de booleanos. La interfaz nos permite trabajar con la clase sin pensar en la forma en que se almacenan los objetos internamente.

### Clase CampoMinas

El objetivo de esta clase es encapsular la funcionalidad del juego, es decir, modificar el tablero conforme se ha descrito en la descripción del problema. Cuando el usuario juega una partida, interaccionará con esta clase para solicitar las acciones que corresponden al juego, y será esta clase la que resuelva la forma en que cambia el tablero y, por tanto, la partida. En esta clase se pueden incluir las siguientes operaciones:

- Un constructor con las dimensiones y el número de minas. Tendrá que inicializar el tablero y posicionar aleatoriamente las minas.
- Consulta del número de filas del tablero.
- Consulta del número de columnas del tablero.
- Consulta si alguna mina ha explosionado. Recorre el tablero y consulta si hay una casilla abierta y con bomba.
- Consulta si se ha ganado la partida. Recuerde que se ha ganado en caso de que no haya sido detonada ninguna mina y además todas las casillas sin mina están abiertas.
- Solicitar que una casilla se marque. En caso de que ya esté marcada, esta operación quitará la marca. Lógicamente, marcar una casilla abierta no tiene sentido. Podría devolver un booleano indicando si realmente se ha llevado a cabo la operación.
- Solicitar la apertura de una casilla. No tiene sentido la apertura de una casilla marcada o abierta. Podría devolver un booleano indicando si se ha conseguido abrir.
- Solicitar la impresión formateada del tablero. Esta operación `-PrettyPrint-` imprime el tablero en `cout`. Será una función sin parámetros.
- Mostrar el tablero formateado sin ocultar nada. Esta operación imprime el tablero en `cout`. Será una función sin parámetros. Corresponde al tablero que se muestra cuando termina la partida, por tanto, en caso de que el juego no haya terminado, mostrará un mensaje del tipo “*No hagas trampa.*”.

Es posible que encuentre operaciones auxiliares que podría facilitar la implementación de la clase. Estas operaciones se pueden añadir como métodos privados. Por ejemplo, puede crear una función que devuelve el número de bombas que hay alrededor de una casilla, o incluso una que devuelve si hay bomba y que acepta incluso posiciones fuera del tablero (donde

resolverá que no hay mina).

Finalmente, es interesante destacar que el algoritmo más interesante corresponde a la apertura de una casilla. Una implementación muy simple y eficaz es implementarla recursivamente: el problema de abrir una casilla incluye resolver el problema de abrir otra casilla. Tenga en cuenta que en esta implementación será necesario marcar la casilla abierta antes de las llamadas recursivas.

## Módulo BuscaMinas

Este módulo contiene la función `main`. En esta primera versión, se implementará de una forma muy simplificada, sin comprobar errores, para poder ejecutar el prototipo cuanto antes de manera que podamos resolver los errores de programación de las clases propuestas.

## 1.4 Modificación del programa: Versión 2

Una vez resuelto la primera versión, se proponen una serie de modificaciones para generar un nuevo programa. En la práctica, las modificaciones de un programa vendrán principalmente determinadas por la corrección de errores, mejora de la eficiencia o la incorporación de nueva funcionalidad. La mayoría de las propuestas de estas sección tienen esa intención, aunque lo más importante es que entienda que las modificaciones pueden ser de muchos tipos y que un buen programa debería facilitar dichos cambios.

Se aconseja realizar los cambios en el orden en que se van presentando en las siguientes secciones, aunque el resultado final sea el mismo.

### 1.4.1 Cambios internos a un módulo

La primera modificación que vamos a realizar al programa consiste en cambiar sólo la parte interna de un módulo, dejando la interfaz intacta. La consecuencia de este cambio es que el resto del programa debería seguir siendo el mismo, ya que los cambios no le afectan.

#### Cambio en la implementación de algoritmos

En este sección proponemos un cambio de una operación sin que afecte a la interfaz. Concretamente, se propone una nueva implementación para la operación de apertura de una casilla. Seguramente, con su experiencia, le resultará habitual la modificación del cuerpo de una función sin que afecte a la cabecera ni a la forma en que se usa desde el resto del programa.

Para que sea más ilustrativo, el cambio que proponemos es la implementación de un algoritmo totalmente nuevo. Recuerde que en la primera versión hemos propuesto un algoritmo recursivo que tenía, de hecho, una implementación muy corta. En este caso, proponemos crear un algoritmo iterativo.

Un algoritmo iterativo simple para resolver el problema a partir de una casilla  $(f, c)$  consiste en mantener una lista `Pend` de posiciones a abrir. Concretamente, tendremos que:

1. Añadir la casilla  $(f, c)$  al la lista `Pend` de casillas por abrir.
2. Mientras queden casillas por procesar en `Pend`:
  - a) Extraer una casilla del conjunto de pendientes `Pend`.
  - b) Si la casilla tiene un número entre 1 y 8, modificarla como abierta.
  - c) Si la casilla tiene un número cero (vacía, oculta y sin minas vecinas), ponerla como abierta e insertar todas sus vecinas no marcadas al conjunto de pendientes `Pend`.

Una vez que el conjunto `Pend` queda vacío, todas las casillas vacías conectadas con la original 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 ni se abrirán ni propagarán la apertura.

El problema está en cómo implementar la lista de pendientes. Puesto que por un lado no sabemos si habrá muchas o pocas y, por otro, el orden de visita de estas casillas no es importante, vamos a implementarla con una lista simple de celdas enlazadas<sup>1</sup>. Para ello, defina una estructura `CeldaPosicion` que contiene 3 campos: fila, columna y puntero a la siguiente. Tenga en cuenta que:

- Inicialmente la lista está vacía: `Pend` valdrá el puntero nulo.
- Insertar un elemento en la lista se puede realizar en la primera posición. Tendrá que reservar un objeto `CeldaPosicion` con `new` y añadirlo al comienzo.
- Extraer una casilla consiste en eliminar la primera celda de la lista y procesar la posición correspondiente.
- La estructura `CeldaPosicion` es una estructura interna de la implementación. Por tanto, no aparece en el fichero de cabecera. De hecho, el usuario de la clase no necesita saber que existe.

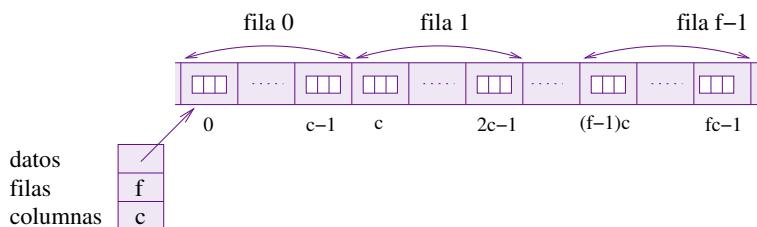
**Ejercicio 1.1 — Cambio de un algoritmo.** Modifique la función de apertura de una casilla. Mantenga la cabecera y la interfaz, modificando la función en base a un algoritmo iterativo que usa una lista de celdas enlazadas con las casillas pendientes de abrir.

<sup>1</sup>En la práctica, una implementación iterativa es probable que evitara las celdas enlazadas, ya que podemos fijar un máximo de casillas a abrir y podríamos almacenar las posiciones en un vector.

### Modificar la representación del Tablero

En esta sección se propone un cambio en la representación para la clase *Tablero*. En lugar de usar una *matriz-C*, proponemos reservar memoria dinámica. De esta forma, tendremos un tipo que se adapta al tamaño real del problema. Si hay pocas casillas, no desperdiciaremos posiciones reservadas y si hay muchas, podemos ejecutar el programa al no estar limitados por un tamaño fijo.

La representación se puede hacer reservando un vector de tantas posiciones como casillas haya en el tablero. En la figura 1.3 se muestra una representación gráfica de la nueva estructura. Observe que tendremos que almacenar un puntero a la zona de memoria y dos enteros con el tamaño del tablero.



**Figura 1.3**  
Representación del tablero en memoria dinámica.

Como consecuencia de este cambio será obligatorio incluir la definición de los siguientes métodos:

- Constructor de copias.
- Destructor.
- Sobrecarga del operador de asignación.

No debe considerar estos métodos como una ampliación de la interfaz. Realmente, estos métodos también existían antes, aunque eran los que generaba automáticamente el compilador. Con la nueva estructura, los métodos generados son incorrectos, y por tanto tenemos que definirlos explícitamente para que el comportamiento de la clase sea exactamente el mismo que en la primera versión.

El cambio de la representación provoca la revisión de todas las funciones que la usan, es decir, que tienen acceso a la parte privada. Afortunadamente, las tenemos muy bien delimitadas, ya que el compilador nos garantiza que las únicas que pueden acceder son las funciones miembro de la clase. Vuelva a compilar el programa para comprobar que tiene el mismo comportamiento.

**Ejercicio 1.2 — Cambio de representación del tablero.** Cambie la representación de la clase *Tablero* para usar un vector de objetos *Casilla* en memoria dinámica.

### Modificar la representación de CampoMinas

Proponemos una modificación de la representación de la clase *CampoMinas*. Concretamente, deseamos mejorar la eficiencia de la función miembro que consulta si hay una mina detonada. Recuerde que la implementación de la primera versión recorre el tablero y busca una casilla abierta con bomba para determinar si ha habido explosión.

La modificación consiste en añadir un objeto de tipo **bool** que indique si ha habido explosión. Inicialmente, valdrá falso, y se activará cuando se abra una casilla con mina. Lógicamente, la función que consulta si ha habido explosión simplemente devolverá el valor de este objeto.

El resultado es un código mucho más eficiente, pues nos ahorraremos esos recorridos gastando solamente lo que ocupa un objeto de tipo **bool**. Tras realizar la modificación, el programa debería funcionar exactamente igual. Recuerde que estamos modificando la parte interna del módulo.

**Ejercicio 1.3 — Cambio de representación del campo de minas.** Cambie la representación de la clase *CampoMinas* para incluir un dato booleano que indique si ha habido una explosión. Modifique el código relacionado para aprovechar la mejora en tiempo.

### 1.4.2 Cambios en la interfaz de un módulo

Los cambios en la interfaz son mucho más problemáticos. Son cambios que deberíamos evitar. El problema es que un cambio de la interfaz no sólo afecta al módulo, sino a todos los que lo usaban. Por tanto, puede afectar a una gran parte del programa.

En esta sección vamos a aproponer algunos cambios, aunque realmente no vamos a mostrar la magnitud del problema, ya que en la práctica, un cambio de interfaz no sólo afecta a un programa, sino que puede afectar a muchos programas: todos aquellos que estén usando el módulo. Por ejemplo, imagine que ha creado un módulo y no sólo lo ha usado en varios programas, sino que también lo ha distribuido y otros paquetes software lo usan. Un cambio de la interfaz afectaría a software que ni sabemos que existe.

## Compatibilidad hacia atrás

Una forma de enfrentarse a cambios en la interfaz es garantizar la compatibilidad hacia atrás. La idea consiste en que un cambio no es problemático si los programas anteriores siguen siendo válidos. En realidad, no es tanto un cambio en la interfaz sino una ampliación.

Se propone cambiar la función que imprime el tablero formateado. Recuerde que se presentó como la función *PrettyPrint* sin parámetros. El cambio consiste en reescribir la función con un parámetro: un flujo de salida (tipo *ostream*). Si cambiamos la función de esta forma, los programas que la usaban dejarían de ser compilables. Para resolverlo, cambiamos la implementación manteniendo la compatibilidad. Para ello tenemos dos posibilidades:

- Mantenemos dos funciones sobrecargadas. Recuerde que podemos usar el mismo nombre para dos funciones que difieren en los parámetros.
- Cambiar la función añadiendo el parámetro e incluir un valor por defecto.

En la práctica, la segunda opción es la mejor solución. Tenga en cuenta que si tenemos dos funciones, serían casi iguales. De hecho, un cambio en una probablemente también habría que hacerlo en la otra.

**Ejercicio 1.4 — Cambio de la interfaz con parámetro por defecto.** Modifique la función *PrettyPrint* añadiendo un parámetro de tipo *ostream* cuyo valor por defecto será *cout*.

## Marcar como obsoleto

Un alternativa menos problemática al cambio de interfaz de un módulo es incluir nuevas posibilidades pero manteniendo la antigua interfaz. La idea consiste en que queremos una nueva interfaz pero no queremos que los programas que usan la antigua dejen de ser válidos. En lugar de cambiarla y provocar la necesidad de cambiar el código antiguo, se avisa de que el cambio se llevará a cabo en el futuro.

El resultado es que hay dos formas de hacer las cosas. El nuevo módulo se publica, indicando qué partes de la interfaz se consideran obsoletas y qué partes se recomienda usar. Se usa la palabra *deprecated*, que podríamos traducir como *desaprobado* o *obsoleto*. Cualquier software que se desarrolle, debería evitar la funcionalidad marcada como *deprecated* pues en futuras versiones podrían directamente eliminarse.

En nuestro problema, vamos a modificar la clase *Tablero*. En lugar de usar las funciones que asignan una casilla o consultan el valor de una casilla, vamos a implementar la sobrecarga del operador paréntesis con dos parámetros. La sobrecarga será doble: para objetos *const* y objetos no *const*. La función devuelve por referencia la casilla. Estas dos funciones, conviven con las anteriores (se añaden sin eliminar nada) dando lugar a una clase que ofrece varias formas de realizar la misma operación. En la documentación se podrían marcar las anteriores como *deprecated*.

**Ejercicio 1.5 — Cambio de la interfaz manteniendo la anterior.** Sobrecharge el operador de paréntesis para la clase *Tablero*. Para probar que funciona correctamente, modifique únicamente la función *Marcar* de la clase *CampoMinas*.

## 1.4.3 Ampliación de la funcionalidad

Modificar un módulo para ampliar la funcionalidad es una operación que no implica demasiados problemas. El único punto que podría ser delicado en esta ampliación es seleccionar una buena interfaz que garantice que será útil y perdurará en futuras revisiones.

### Ampliar la funcionalidad de un módulo

Para mejorar la funcionalidad de los módulos de nuestro programa vamos a incorporar las operaciones de E/S que nos permitan salvar el estado del juego en un momento determinado. Para ello, será necesario:

1. Incluir operaciones de E/S como texto para el tipo *Casilla*. Recuerde que es una estructura para la que podemos sobrecargar los operadores de E/S.
2. Incluir operaciones de E/S como texto para el tipo *Tablero*. Estas operaciones almacenan las dimensiones del tablero seguidas por el contenido de cada una de las casillas. Usará, por tanto, las operaciones propuestas en el punto anterior.
3. Incluir operaciones de E/S como texto para el tipo *CampoMinas*. En este caso, proponemos crear dos funciones *-Leer* y *Escribir*— que salvan el estado actual en un fichero. Este fichero contendrá como primera línea una marca *-cadena mágica-* que identifica el archivo. Por ejemplo, la cadena *#MP-BUSCAMINAS-V1* al principio del archivo.

**Ejercicio 1.6 — Ampliar la funcionalidad.** Amplíe los tipos *Casilla* y *Tablero* con la sobre carga de los operadores de E/S. Además, incluya dos funciones miembro *Leer* y *Escribir* en la clase *CampoMinas* para cargar y salvar, respectivamente, el estado actual del juego en un fichero. Éstas dos pueden devolver un booleano indicando si han tenido éxito.

### Ampliar la funcionalidad del programa

Finalmente, la segunda versión del programa incorporará algunas modificaciones del programa que implementa el juego en consola. Las modificaciones serán:

- El programa obtendrá los parámetros desde la línea de órdenes. Los parámetros consistirán en :

- Si tiene 3 parámetros, considerará que son las filas, columnas y número de minas.
- Si tiene 1 parámetro, se considerará que es el archivo que contiene el tablero. En este caso, el programa no necesita parámetros del juego, pues la carga del tablero le permite disponer de toda la información. Esta forma de ejecución nos permite dejar una partida a medias, salvar el tablero y posteriormente volver a retomarla.
- En otro caso, debe imprimir un mensaje con estas indicaciones y terminar.
- El programa debe comprobar que los parámetros que le hemos dado tienen sentido. Considerará sólo las partidas que tienen dimensiones de al menos 4 filas o 4 columnas y un número de minas positivo inferior a la mitad de las casillas del tablero.
- El programa incorporará una nueva acción '*s*' que corresponde a salvar el estado del tablero. La entrada consiste en esa acción seguida del nombre del archivo donde salvarlo.
- El programa debe incorporar el análisis de la entrada de la acción a realizar. Debe leer una acción como una línea de texto que deberá analizar para extraer tanto la acción correspondiente como sus parámetros. Para ello, tenga en cuenta que:
  - Puede asumir que una línea no tendrá más de 100 caracteres.
  - La acción de abrir puede especificarse con la letra '*a*' o con la palabra "*abrir*", en minúscula o mayúscula. De forma similar, para la acción de marcar –letra '*m*' o palabra "*marcar*"– y la acción salvar –letra '*s*' o palabra "*salvar*"–.
  - Todo el procesamiento de las cadenas deberá realizarse con *cadenas-C*<sup>2</sup>. Puede ser aconsejable encapsular el algoritmo de análisis de la acción en una función que recibe la cadena introducida y devuelve los parámetros correspondientes.
  - Si la acción no se ha podido decodificar, indicará que es una acción incorrecta y volverá a pedirla.

**Ejercicio 1.7 — Ampliación del programa.** Implemente las ampliaciones listadas en esta sección. Compruebe que el programa funciona correctamente introduciendo parámetros incorrectos o acciones no válidas.

## 1.5 Práctica a entregar

Debe crear una carpeta –por ejemplo, *buscaminas*– en la que incluya todos los archivos que componen la solución de este guión. Tenga en cuenta que debe crear dos versiones, por lo que deberá tener dos directorios –por ejemplo, *version1* y *version2*– que contendrán dos programas completamente independientes. En la figura 1.4 se presenta un esquema de cómo quedarán. Observe que no hay un directorio para bibliotecas, pues se generará el ejecutable directamente a partir de los archivos objeto.

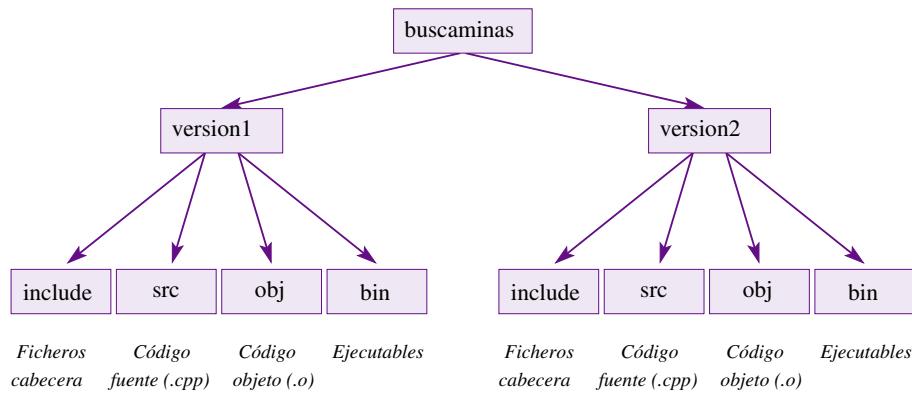


Figura 1.4  
Directorios del proyecto Buscaminas.

Aunque en este guión se han descrito las tareas de modificación como una secuencia de ejercicios, la segunda versión no contendrá ninguna referencia a ellos, sino la solución final. Tenga en cuenta que la secuenciación de tareas sólo se ha planteado para guiar al estudiante hasta la solución final.

Para desarrollar la solución, puede crear la primera versión y una vez terminada, copiarla exactamente en otro directorio. El archivo *Makefile* y toda la estructura que tenía será válido para realizar las modificaciones para la segunda versión. Además, no olvide que deberá incluir operaciones que permitan la “*limpieza*” de archivos intermedios generados.

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

<sup>2</sup>Puede consultar funciones de la biblioteca estándar de C como *strcmp*, *atoi*, *isspace* o *tolower*.



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

```
prompt% cd version1
prompt% make clean
prompt% cd ../version2
prompt% make clean
prompt% cd ../..
prompt% tar zcvf buscaminas.tgz buscaminas
```

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

# A

# Generación de números aleatorios

Introducción.....	11
El problema .....	11
Números pseudoaleatorios	
Transformación del intervalo .....	13
Operación módulo	
Normalizar a U(0,1)	

## A.1 Introducción

Muchos programas incluyen la generación automática de números aleatorios. Un ejemplo muy claro es un juego donde se tiene que avanzar con eventos que simulen aleatoriedad a fin de obtener cierta variedad en el desarrollo de distintas partidas.

Por ejemplo, imagine que deseamos crear un programa que avanza en un juego donde se lanza un dado. El programa simulará que obtenemos valores del conjunto  $\{1, 2, 3, 4, 5, 6\}$  hasta el final de la partida. Por ejemplo, podemos obtener:

1, 1, 6, 2, 4, 4, 3, 6, 2, 5, 5, 5, 1, ...

donde puede ver que se obtienen distintos valores, que se pueden repetir, con un orden indeterminado. Lógicamente, cuando empecemos una nueva partida, los valores que se obtienen deben ser distintos a los de la partida anterior.

En este apéndice vamos a ver cómo podemos hacer que nuestros programas puedan generar dichos valores. Para ello presentamos las funciones que están disponibles en C++ y que ya se podían usar en lenguaje C.

Existen otras utilidades disponibles sólo en lenguaje C++ –desde el estándar C++11– que resultan especialmente interesantes, no sólo porque puede realizar las mismas operaciones, sino porque ofrecen herramientas más avanzadas para crear fácilmente soluciones a problemas más complejos. Si bien no son difíciles de manejar, para entender bien estas nuevas herramientas convendría conocer algunos conceptos de teoría de la probabilidad. No es intención de este documento entrar en esa teoría, por lo que presentaremos las ideas más básicas de una forma muy intuitiva de forma que pueda resolver los problemas más simples y habituales sin gran dificultad.

## A.2 El problema

Fundamentalmente, el problema consiste en que un programa tiene que ser capaz de generar una secuencia de números aleatorios tan larga como deseemos. Es decir, estamos interesados en obtener una serie de valores aleatorios:

$x_0, x_1, x_2, \dots, x_i, \dots$

Para disponer de una utilidad básica que nos resuelva este problema basta con crear una función que nos devuelve cada uno de esos valores conforme la llamamos. En lenguaje C, y por tanto en C++, se ofrece la solución más simple: crear una función **rand** que devuelve estos valores. Esta función no tiene parámetros. Sólo devuelve un nuevo valor entero aleatorio. Por ejemplo, si deseamos obtener 1000 valores aleatorios podemos escribir:

```
#include <cstdlib> // rand()
//...
for (int i=0; i<1000; ++i)
    cout << rand() << ' ';
```

La función está diseñada para obtener un valor entero en el rango  $[0, RAND\_MAX]$ , donde *RAND\_MAX* es una constante predeterminada. Tenga en cuenta que:

- Es necesario incluir el archivo **cstdlib** para disponer de esta función.
- La constante *RAND\_MAX* depende de su sistema, ya que no está fijada en el estándar. Sólo sabemos que es entera, ya que el valor devuelto por **rand** es **int**.

- El entero obtenido puede ser cualquiera del intervalo  $[0, RAND\_MAX]$  con igual probabilidad. Lo que se conoce por una distribución uniforme, ya que cualquier valor de ese intervalo, incluyendo el 0 y  $RAND\_MAX$ , se obtiene con igual probabilidad.
- Es de esperar que el valor de  $RAND\_MAX$  sea bastante alto. De hecho es posible que sea el entero más grande. Por ejemplo, en un sistema con tamaño de palabra pequeño, 2 bytes, podría tener el valor 32767 o si tiene un sistema de 32 bits, es probable que valga 2147483647.

### A.2.1 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. El truco está, lógicamente, en que no necesitamos números aleatorios, sino que basta con que lo parezcan. Ahora bien, todos los números están determinados por la función  $f(x)$ , pero no sabemos cuánto vale el primer valor con el que comienza la secuencia. Toda la secuencia depende de él, de manera que si fijamos un valor concreto, siempre obtendremos la misma secuencia. Por ello, se denomina *semilla*.

Si nuestros programas usaran siempre la misma semilla, los números pseudoaleatorios que generaríamos serían siempre los mismos. Si queremos que distintas ejecuciones den lugar a distintas secuencias, es necesario cambiar de semilla en cada ejecución. Una forma muy simple de obtener distintas semillas es usar el valor del reloj del sistema. Note que si ejecutamos dos veces distintas un mismo programa, la semilla dependería del momento en que damos la orden de ejecución.

Para fijar una semilla, usamos la función **srand** de **cstdlib**, y para obtener un valor del reloj del sistema la función **time** del fichero de cabecera **ctime**. Un programa muy simple que genera tres valores aleatorios es el siguiente:

```
#include <cstdlib>    // rand, srand
#include <ctime>       // time
using namespace std;
int main()
{
    srand (time(0));

    int aleatorio1= rand();
    int aleatorio2= rand();
    int aleatorio3= rand();
}
```

Observe que la semilla sólo hay que fijarla al principio del programa (una única vez), y a partir de ese momento, se puede usar **rand** tantas veces como se desee para obtener nuevos valores pseudoaleatorios. Por ejemplo, si ejecuta el siguiente programa:

```
for (int i=0; i<10; ++i) {
    srand (time(0));
    cout << rand() << ' ';
}
cout << endl;
```

es posible que obtenga 10 valores iguales. O tal vez, si por casualidad el valor de **time** avanza durante el bucle, 2 grupos con los valores iguales. Por ejemplo, una ejecución en mi máquina ha dado lugar a:

```
1094219464 1094219464 1094219464 1094219464 1094219464
1094219464 1094219464 1094219464 1094219464 1094219464
```

El problema es que hemos situado el generador en un valor de semilla idéntico en cada iteración. Situamos el primer valor, generamos el valor aleatorio y en la siguiente iteración volvemos a situar de nuevo el generador en el primer valor. Recuerde que si **time** devuelve el valor en segundos, las 10 iteraciones del bucle probablemente situarán la misma semilla y el valor que se genera con **rand** será el mismo.

Realmente, **time** es muy poco aleatoria. Sin embargo, nosotros queremos un valor entero distinto cada vez que lancemos el programa, por lo que resolveremos el problema generando una primera semilla al comienzo y limitándonos a usar la función **rand** en el resto del programa. Note que la semilla dependerá del segundo en el que ejecutemos el programa, por lo que resultará en ejecuciones con secuencias aleatorias distintas.

## A.3 Transformación del intervalo

Normalmente no nos interesaría generar un número aleatorio en el intervalo  $[0, RAND\_MAX]$ , especialmente teniendo en cuenta que no conocemos el valor de esa constante. Por ejemplo, si queremos lanzar un dado queremos un valor entero que va del 1 al 6. Para resolver el problema debemos transformar el valor generado al intervalo deseado.

### A.3.1 Operación módulo

Si deseamos obtener un valor en un intervalo de enteros pequeño lo más tentador es realizar una operación de módulo con el operador `' % '`. Por ejemplo, para generar el valor de un dado podemos escribir:

```
int dado= rand()%6+1;
```

En general, si queremos obtener un valor en el intervalo de enteros  $[MIN, MAX]$ , ambos inclusive, podríamos calcularlo como:

```
aleatorio= rand()%(MAX-MIN+1)+MIN;
```

Es probable que si consulta código en la red, encuentre muchos ejemplos con líneas de este tipo. En general se utiliza este código frecuentemente por su simplicidad, aunque no resulta especialmente recomendable cuando las características de aleatoriedad del generador son especialmente importantes para la aplicación que se está desarrollando.

Efectivamente, con la operación módulo estamos haciendo depender el valor del número generado especialmente de los bits menos significativos del número generado. Por ejemplo, si realizamos una operación de módulo 100, realmente estamos cogiendo los dos últimos dígitos decimales de los enteros generados. Aunque el resultado final dependerá de la función  $f(x_i)$  que se esté usando como motor de generación, es probable que el comportamiento de los bits menos significativos sea menos aleatorio de lo esperado.

### A.3.2 Normalizar a $U(0,1)$

La variable aleatoria uniforme en el intervalo  $[0, 1]$  –que denotamos  $U(0, 1)$ – es especialmente importante en la teoría de la probabilidad. De hecho, si consulta distintos algoritmos de generación de números aleatorios para distintas distribuciones de probabilidad, encontrará que incluyen la generación de uno o varios valores de esta variable.

De forma simplificada, digamos que generar un valor de una variable  $U(0, 1)$  es obtener cualquier valor de ese intervalo, teniendo en cuenta que todos los números de ese intervalo tienen igual probabilidad. Con la función `rand` no tenemos más que transformar el valor dividiendo por el máximo `RAND_MAX`. En concreto, podemos usar la siguiente función:

```
inline double Uniforme01()
{
    return rand() / (RAND_MAX+1.0);
}
```

En este código debería observar que sumamos el valor `1.0`, que es de tipo `double`. Es interesante notar que la operación `RAND_MAX+1` es peligrosa. Esta expresión es entera, por lo que la división del valor de `rand` entre este número sería entera y casi seguro que daría el valor cero. Podría pensar que el siguiente código resuelve el problema:

```
inline double Uniforme01()
{
    return rand() / (double) (RAND_MAX+1); // Error
}
```

Sin embargo, no sólo es peligrosa por eso, sino que el valor de `RAND_MAX` podría ser el entero más grande por lo que al sumar uno se obtiene un entero incorrecto. Posiblemente, el entero más pequeño –el más negativo– del rango del tipo `int`.

Por otro lado, el valor que hemos obtenido es un número real del intervalo  $[0, 1]$ , sin llegar a alcanzar el valor 1.0. Con este número podemos obtener cualquier valor en el intervalo deseado sin más que realizar una transformación sencilla. Por ejemplo:

```
int dado= Uniforme01()*6+1;
```

Al usar esta expresión el valor aleatorio se encuentra en el intervalo  $[0, 6]$  al multiplicarlo por 6, y en el intervalo  $[1, 7]$  al sumar 1. Cuando asignamos a la variable `dado`, nos quedamos con el valor entero correspondiente. En general, podemos obtener un valor en un intervalo con la siguiente función:

```
inline int Uniforme(int minimo, int maximo)
{
    double u01= rand() / (RAND_MAX+1.0); // Uniforme01
    return minimo+ (maximo-minimo+1)*u01;
}
```



# B

## Tablas

Tabla ASCII .....	15
Operadores C++ .....	16
Palabras reservadas de C89, C99, C11, C++ y C++11.....	17
Manipuladores y funciones miembro para E/S formateada .....	17
Banderas y máscaras	
Funciones miembro y manipuladores	

### B.1 Tabla ASCII

En la figura B.1 se presenta la tabla de codificación ISO-8859-15 del alfabeto latino. Es similar a la ISO-8859-1 aunque difiere en 8 posiciones (`0xA4`, `0xA6`, `0xA8`, `0xB4`, `0xB8`, `0xBC`, `0xBD` y `0xBE`). Esta codificación se distingue especialmente porque incluye el carácter correspondiente al euro.

Aunque es probable que la codificación ISO-8859-15 no sea la que esté usando en su sistema, la mayoría de los problemas que se resuelven en los cursos de programación asumen que es la codificación para los caracteres o secuencias de caracteres.

	<b>x0</b>	<b>x1</b>	<b>x2</b>	<b>x3</b>	<b>x4</b>	<b>x5</b>	<b>x6</b>	<b>x7</b>	<b>x8</b>	<b>x9</b>	<b>xA</b>	<b>xB</b>	<b>xC</b>	<b>xD</b>	<b>xE</b>	<b>xF</b>
<b>0x</b>	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
<b>1x</b>	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
<b>2x</b>	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
<b>3x</b>	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
<b>4x</b>	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
<b>5x</b>	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
<b>6x</b>	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
<b>7x</b>	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
<b>8x</b>	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
<b>9x</b>	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGC	SCI	CSI	ST	OSC	PM	APC
<b>Ax</b>	NBSP	í	¢	£	€	¥	Š	§	š	©	¤	«	¬	SHY	®	-
<b>Bx</b>	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ý	ÿ
<b>Cx</b>	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
<b>Dx</b>	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	þ	ß
<b>Ex</b>	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
<b>Fx</b>	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

**Figura B.1**  
Tabla de codificación ISO-8859-15.

Observe que todos los caracteres especiales que existen en distintos idiomas y no en inglés están en la segunda parte de la tabla. Realmente, la tabla ASCII propiamente dicha es la primera mitad, mientras que la segunda es una extensión. Por ello, esta tabla no es la tabla ASCII, sino la tabla ASCII extendida ISO-8859-15.

En la práctica aceptaremos esta codificación para nuestras soluciones, ya que nos interesa especialmente el algoritmo a resolver sin entrar en detalles sobre los problemas de codificación. Note que podría tener problemas en la ejecución de

algoritmos en español incluso si su sistema usa esta codificación. Por ejemplo, si quiere comprobar el orden de dos letras para ordenar una cadena, los caracteres con tilde están fuera del rango del alfabeto '*a*'-'*z*'.

Si su sistema usa otras codificaciones como ISO-8859-1, windows-1252, o UTF-8, no tendrá problemas en ejecutar y comprobar el funcionamiento de sus algoritmos si evita el uso de la parte extendida, ya que en esas codificaciones los caracteres básicos se representan exactamente igual. Podríamos decir que realizamos soluciones que funcionan sin ningún problema solamente en inglés.

Más adelante es posible que tenga que resolver problemas para distintos lenguajes. Tendrá que consultar el tipo de codificación de su sistema y qué posibilidades tiene para resolverlo. Las soluciones pueden ir desde un tipo **char** con otra codificación hasta usar bibliotecas de funciones que resuelven sus problemas configurando el lenguaje usado.

## B.2 Operadores C++

En la siguiente tabla se listan los operadores de C++. Los operadores situados en el mismo recuadro tienen la misma precedencia.

En general, para no tener ningún problema con la asociatividad, intente recordar que los operadores unarios y de asignación son asociativos por la derecha, mientras que los demás lo son por la izquierda.

Si revisa el estándar, podrá encontrar que realmente los unarios postfijo son de izquierda a derecha así como el operador condicional, aunque en la práctica no suelen crear incertidumbre.

Operadores de C++		
Operador	Nombre	Uso
::	Global	:: <i>nombre</i>
::	Resolución de ámbito	<i>espacio_nombres</i> :: <i>miembro</i>
::	Resolución de ámbito	<i>nombre_clase</i> :: <i>miembro</i>
->	selección de miembro	<i>puntero</i> -> <i>miembro</i>
.	Selección de miembro	<i>objeto</i> . <i>miembro</i>
[]	Índice de vector	<i>nombre_vector</i> [ <i>expr</i> ]
()	Llamada a función	<i>nombre_función</i> ( <i>lista_expr</i> )
()	Construcción de valor	<i>tipo</i> ( <i>lista_expr</i> )
++	Post-incremento	<i>valor_i</i> ++
--	Post-decremento	<i>valor_i</i> --
typeid	Identificador de tipo	typeid( <i>type</i> )
typeid	... en tiempo de ejecución	typeid( <i>expr</i> )
dynamic_cast	conversión en ejecución	dynamic_cast< <i>tipo</i> >( <i>expr</i> )
	con verificación	
static_cast	conversión en compilación	static_cast< <i>tipo</i> >( <i>expr</i> )
	con verificación	
reinterpret_cast	conversión sin verificación	reinterpret_cast< <i>tipo</i> >( <i>expr</i> )
const_cast	conversión const	const_cast< <i>tipo</i> >( <i>expr</i> )
sizeof	Tamaño del tipo	sizeof( <i>tipo</i> )
sizeof	Tamaño del objeto	sizeof <i>expr</i>
++	Pre-incremento	++ <i>valor_i</i>
--	Pre-decremento	-- <i>valor_i</i>
~	Complemento	~ <i>expr</i>
!	No	! <i>expr</i>
+	Más unario	+ <i>expr</i>
-	Menos unario	- <i>expr</i>
&	Dirección de	& <i>valor_i</i>
*	Desreferencia	* <i>expr</i>
new	reserva	new <i>tipo</i>
new	reserva e iniciación	new <i>tipo</i> ( <i>lista_expr</i> )
new	reserva (emplazamiento)	new ( <i>lista_expr</i> ) <i>tipo</i>
new	reserva (con inicialización)	new ( <i>lista_expr</i> ) <i>tipo</i> ( <i>lista_expr</i> )
delete	destrucción (liberación)	delete <i>puntero</i>
delete []	... de un vector	delete [] <i>puntero</i>
()	Conversión de tipo	( <i>tipo</i> ) <i>expr</i>
.*	Selección de miembro	<i>objeto</i> . * <i>puntero_a_miembro</i>
->*	Selección de miembro	<i>puntero</i> ->* <i>puntero_a_miembro</i>
*	Multiplicación	<i>expr</i> * <i>expr</i>

continúa en la página siguiente

continúa de la página anterior		
Operador	Nombre	Uso
/	División	<i>expr</i> / <i>expr</i>
%	Módulo	<i>expr</i> % <i>expr</i>
+	Suma	<i>expr</i> + <i>expr</i>
-	Resta	<i>expr</i> - <i>expr</i>
<<	Desplazamiento a izquierda	<i>expr</i> << <i>expr</i>
>>	Desplazamiento a derecha	<i>expr</i> >> <i>expr</i>
<	Menor	<i>expr</i> < <i>expr</i>
<=	Menor o igual	<i>expr</i> <= <i>expr</i>
>	Mayor	<i>expr</i> > <i>expr</i>
>=	Mayor o igual	<i>expr</i> >= <i>expr</i>
==	Igual	<i>expr</i> == <i>expr</i>
!=	No igual	<i>expr</i> != <i>expr</i>
&	Y a nivel de bit	<i>expr</i> & <i>expr</i>
^	O exclusivo a nivel de bit	<i>expr</i> ^ <i>expr</i>
	O a nivel de bit	<i>expr</i>   <i>expr</i>
&&	Y lógico	<i>expr</i> && <i>expr</i>
	O lógico	<i>expr</i>    <i>expr</i>
?:	expresión condicional	<i>expr</i> ? <i>expr</i> : <i>expr</i>
=	Asignación	<i>valor_i</i> = <i>expr</i>
*=	Multiplicación y asignación	<i>valor_i</i> *= <i>expr</i>
/=	División y asignación	<i>valor_i</i> /= <i>expr</i>
%=	Resto y asignación	<i>valor_i</i> %= <i>expr</i>
+=	Suma y asignación	<i>valor_i</i> += <i>expr</i>
-=	Resta y asignación	<i>valor_i</i> -= <i>expr</i>
<=>	Desplazar izq. y asignación	<i>valor_i</i> <<= <i>expr</i>
>=>	Desplazar der. y asignación	<i>valor_i</i> >>= <i>expr</i>
&=	Y y asignación	<i>valor_i</i> &= <i>expr</i>
^=	O exclusivo y asignación	<i>valor_i</i> ^= <i>expr</i>
=	O y asignación	<i>valor_i</i>  = <i>expr</i>
throw	Lanzamiento de excepción	throw <i>expr</i>
,	Coma	<i>expr</i> , <i>expr</i>

**Tabla B.1**  
Operadores de C++

Estos operadores están incluidos en el estándar C++98. En C++11 aparecen tres más:

`sizeof..., noexcept, alignof`

## B.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 B.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.

## B.4 Manipuladores y funciones miembro para E/S formateada

En la siguiente figura B.2 se presentan los manipuladores y funciones miembro que ofrece C++ para formatear la E/S. Es recomendable estudiar la forma en que funcionan antes de usar esta tabla, ya que se presenta más como una tabla de referencia que como una lista de posibilidades.

Fundamentalmente, la E/S está controlada por un conjunto de banderas y valores almacenados en el flujo:

- Las banderas no son más que un valor booleano que indica si una característica está activa o no. Por ejemplo, hay una bandera para indicar si se presenta el signo a los números positivos. Por defecto sólo se presenta el signo a los negativos –con el carácter ‘–’– pero si activamos la bandera *showpos*, también se incluirá el carácter ‘+’ cuando el número es positivo.
- Se incluyen valores para controlar otros aspectos más complejos. Por ejemplo, el flujo almacena el carácter con el que tiene que llenar el espacio que queda cuando ajustamos un campo a la derecha.

**Tabla B.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	

#### B.4.1 Banderas y máscaras

Si observa la figura B.2 (página 20) descubrirá que no sólo hablamos de banderas, sino también de máscaras. Las máscaras no son más que un grupo de banderas. Son necesarias para poder realizar fácilmente operaciones sobre todas las banderas que componen la máscara.

Por ejemplo, la base usada al escribir un entero se controla con tres banderas: *oct*, *dec*, *hex*. Con tres banderas tenemos hasta 8 posibilidades dependiendo de cuáles activamos o no. Sin embargo, realmente sólo tienen sentido cuatro: primera activa, segunda activa, tercera activa o ninguna activa. No tiene sentido indicar que la salida se formatea en octal y decimal, por ejemplo.

Para poder manejar fácilmente un grupo de banderas en el que sólo tiene sentido que una esté activa se crean las máscaras. Estas máscaras son la unión de las distintas banderas. Cuando activamos una bandera de una máscara, realmente estamos desactivando también las otras.

A modo ilustrativo, presentamos el siguiente esquema de código para que entienda este mecanismo y le resulte más sencillo entender el mecanismo que usamos. En primer lugar definimos un entero que representa el lugar donde vamos a guardar todas las banderas.

```
// Un entero para almacenar hasta 32 banderas
unsigned int estado; // Está en algún lugar definido.
```

Este entero podría estar definido en algún lugar interno al flujo. Así, sólo gastará 4 bytes para controlar todas las banderas.

Para que el usuario –nosotros– podemos usar las banderas, se crean algunas constantes con nombres de banderas y máscaras. Por ejemplo, podemos crear los 3 nombres siguiente:

```
const int MascaraTriple= 0x7; // Una máscara que agrupa los 3 bits (situados al final del estado)
const int Bandera1= 0x1; // Una de las banderas incluidas en la máscara
const int Bandera2= 0x2; // Una de las banderas incluidas en la máscara
const int Bandera3= 0x4; // Una de las banderas incluidas en la máscara
```

Observe que para el usuario no es importante ni el sitio donde se guardan las banderas, ni el tipo que se usa ni los valores concretos. Realmente, sólo nos interesa que hay una máscara con un determinado nombre y 3 nombres para cada una de las banderas que tiene la máscara.

Para manejarlas, el que crea la biblioteca ofrece al usuario dos funciones:

```
void Reset (int mascara)
{
    estado= estado & ~mascara;
}
void SetF (int bandera, int mascara)
{
    estado= estado & ~mascara | bandera;
```

donde podríamos haber usado la primera para resolver la segunda.

Con estas funciones puede realizar fácilmente operaciones que activan y desactivan banderas. Por ejemplo:

```
Reset(MascaraTriple); // Deja desactivadas las tres banderas
//...
SetF(Bandera1,MascaraTriple); // Activa la bandera 1 y deja desactivadas 2,3
//...
SetF(Bandera3,MascaraTriple); // Activa la 3 y deja desactivadas 1,2
```

Note que las llamadas a *SetF* garantizan que la bandera que queremos activar será la única que quedará activada, independientemente del estado que teníamos antes de la llamada a la función.

Finalmente –“ya metíos en gastos”– imagine que añado una operación más *SetF* y mejoro la legibilidad de la función *Reset* anterior:

```
void Reset (int banderaOmaskara)
{
    estado= estado & ~banderaOmaskara;
}
void SetF (int bandera)
{
    estado= estado | bandera;
```

El resultado es algo que empieza a recordar a varias de las operaciones que hemos presentado en la figura B.2. Note que la función *SetF* nos servirá para banderas independientes, que no están en una máscara o que se pueden activar simultáneamente a otras. Sin embargo, se recomendará usar la función con dos parámetros cuando queramos activar sólo una bandera de una máscara.

## B.4.2 Funciones miembro y manipuladores

En general, se puede decir que el sistema de E/S define dos estrategias para poder enviar órdenes de formateo a un flujo de E/S:

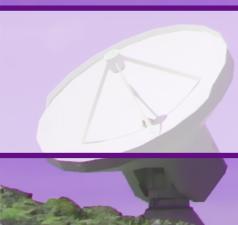
1. Funciones miembro. Se llama a un método del objeto con el operador punto. Por ejemplo, se puede llamar a **cout**. *fill(' ')*; para seleccionar el espacio como el carácter de relleno en **cout**.
2. Es un mecanismo que nos permite encadenar esas órdenes de formato los operadores habituales –<< y >>– de E/S para el flujo. Esta sintaxis facilita la escritura y mejora la legibilidad de estas órdenes.

En algunos casos, tenemos tanto la función miembro como el manipulador disponibles para modificar el formato de E/S (observe que en la figura B.2 sólo se especifica el manipulador en algunos casos). Si usa un manipulador y no se reconoce, es probable que le falte incluir el archivo de cabecera **iomanip**, donde se define.

En la práctica, seguramente lleva usando manipuladores desde que comenzó a programar ya que **endl** es un manipulador. En este caso, no necesita de la inclusión de **iomanip**, ya que no tiene parámetros.

	Bandera o función miembro	Descripción	Manipulador (si existe)	Notas
Activar/Desactivar banderas general	setf(f)	Activa bandera(s) f y devuelve estado previo	setiosflags(f)	Los identificadores de banderas y máscaras están en std::ios_base, aunque se puede usar std::ios, que es descendiente. Ej: std::ios::adjustfield El ancho de campo se puede usar para la entrada, indicando que sólo se pueden leer n-1 caracteres. Por ejemplo, para leer una cadena C El tipo que corresponde al grupo de banderas es ios::fmtflags. Por ejemplo, se puede hacer cout.setf(ios::fmtflags(0),ios::floatfield) Escriba #include <iomanip> para manipuladores con parámetros. Manipuladores de una bandera en una máscara desactivan el resto de ésta
	setf(f,mask)	Activa bandera f del grupo mask y devuelve estado previo de todas las banderas	resetiosflags(mask)+ +setiosflags(f)	
	unsetf(f)	Desactiva bandera(s) f	resetiosflags(f)	
	flags()	Devuelve todas las banderas		
	flags(f)	Selecciona banderas f como el nuevo estado y devuelve anterior		
Ancho de campo, relleno y ajuste (números, bool, cadena C, etc.)	copyfmt(str)	Copia las banderas desde flujo str		
	width()	Ancho de campo actual (por defecto 0, es decir, lo que se necesite)		
	width(n)	Fija ancho n y devuelve anterior (sólo afecta a E/S siguiente)	setw(n)	
	fill()	Devuelve carácter de relleno actual (por defecto: espacio)		
	fill(c)	Fija c como carácter de relleno	setfill(c)	
Números enteros	máscara adjustfield	left	left	
	right	Ajusta a la derecha	right	
	internal	Signo a la izquierda y valor a la derecha	internal	
	Ninguno	Ajusta a la derecha (por defecto)	resetiosflags(adjustfield)	
	showpos	Escribe el signo a los números positivos	< showpos noshowpos	
Números enteros	máscara basefield	uppercase	< uppercase nouppercase	
		showbase	< showbase noshowbase	
	oct	Escribe y lee octal	oct o setbase(8)	
	dec	Escribe y lee decimal (por defecto)	dec o setbase(10)	
	hex	Escribe y lee hexadecimal	hex o setbase(16)	
Punto flotante	Ninguno	Escribe decimal y lee de acuerdo a los caracteres iniciales	resetiosflags(basefield)	
	fixed	Notación fija (precisión indica el número de dígitos después de la coma)	fixed	
	scientific	Notación científica (precisión= núm. dígitos después de la coma)	scientific	
	Ninguno	La mejor de las dos (por defecto)	resetiosflags(floatfield)	
	precision()	Devuelve precisión actual de punto flotante (por defecto, 6)		
En istream y ostream	precision(p)	Selecciona precisión p y devuelve anterior	setprecision(p)	
	showpoint	Escribe ceros a la derecha del punto decimal		
	boolalpha	Si los bool se manejan textualmente (defecto 0/1). El texto depende de idioma	< boolalpha noboolalpha	
	flush()	Descarga búfer de salida	flush	
		Inserta salto de línea y descarga búfer	endl	
Otros		Inserta carácter fin de cadena	ends	
		Lee e ignora "espacios blancos"	ws	
	skipws	Salta "espacios blancos" iniciales en cada lectura (por defecto)	< skipws noskipws	
	unitbuf	Descarga búfer de salida tras cada escritura (por defecto para cerr y wcerr)	< unitbuf nounitbuf	

Figura B.2  
CheatSheet para E/S formateada.



## Índice alfabético

### A

ASCII, 15

### B

buscaminas, juego, 1

### F

función de biblioteca  
rand, 11  
srand, 12  
time, 12

### I

iomanip, 19  
ISO8859, 15

### M

manipulador de E/S, 19

### N

números aleatorios, 11  
pseudoaleatorios, 12

### O

operadores C++, 16

### P

palabras reservadas, 17  
pseudoaleatorio, 12

### R

rand, 11

reservadas  
palabras, 17

### S

semilla, 12  
srand, 12

### T

time, 12

### U

UTF-8, 16

### W

Windows-1252, 16