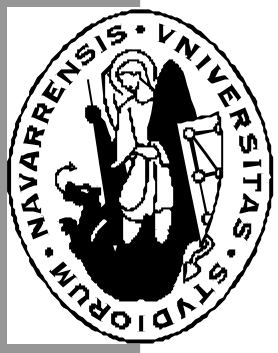


tecnun

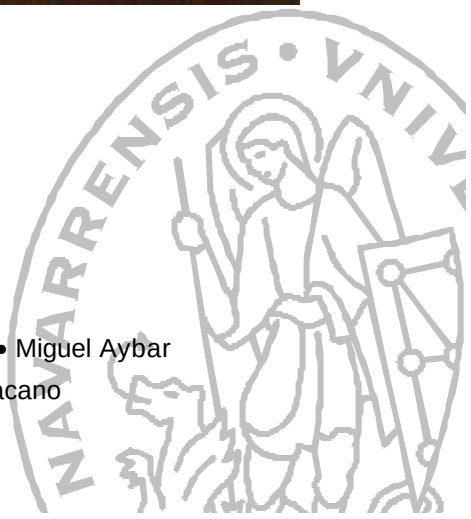
CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Escuela Superior de Ingenieros · Ingeniarien Goi Mailako Eskola

Aprenda Informática ...

Aprenda C++ Básico

como si estuviera en primero

San Sebastián, Febrero 2004

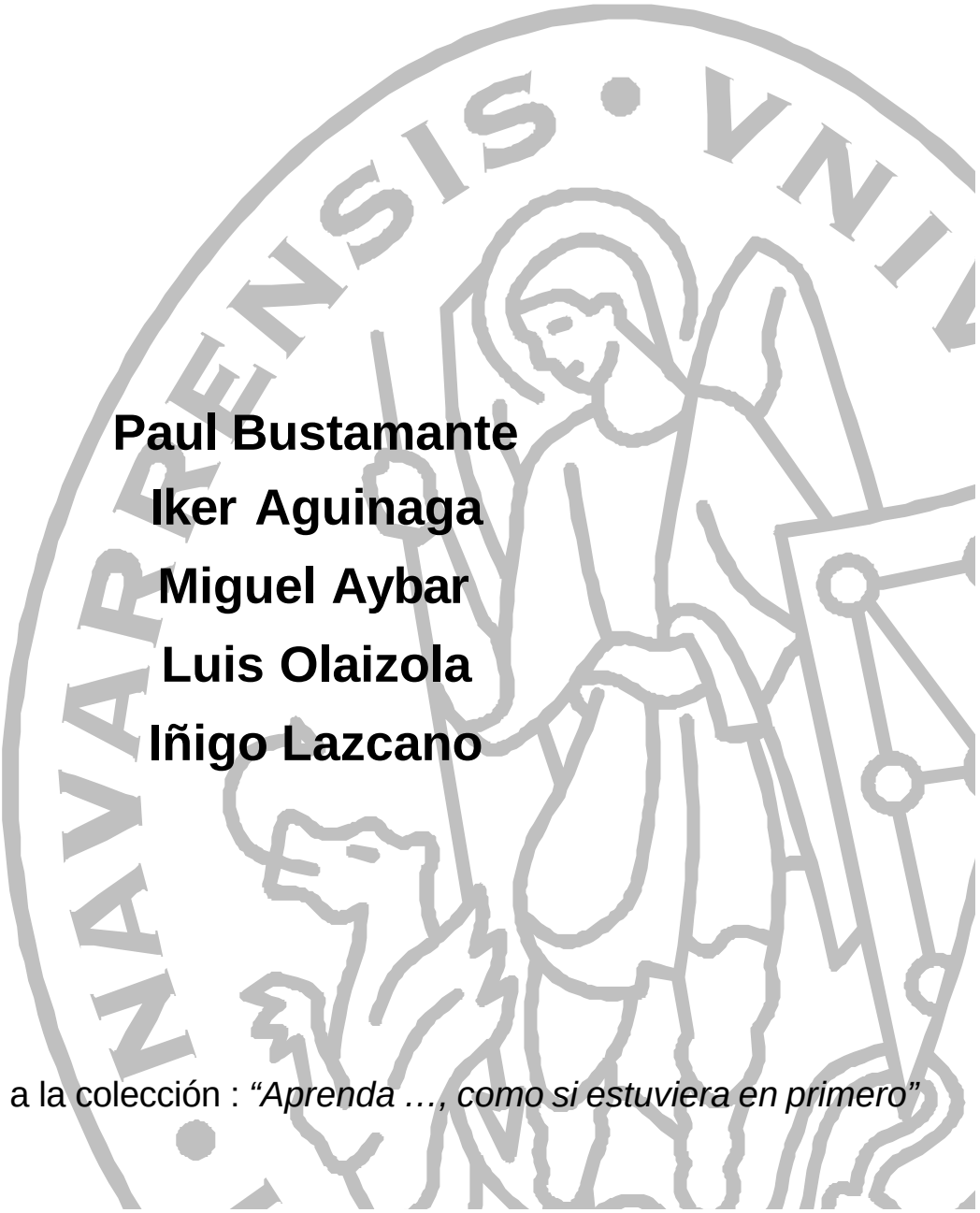


Paul Bustamante • Iker Aguinaga • Miguel Aybar
Luis Olaizola • Iñigo Lazacano

tecnun
CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Escuela Superior de Ingenieros · Ingeniarien Goi Mailako Eskola

Aprenda C++ Básico

como si estuviera en primero



Paul Bustamante
Iker Aguinaga
Miguel Aybar
Luis Olaizola
Iñigo Lazcano

Perteneciente a la colección : *“Aprenda …, como si estuviera en primero”*

Esta publicación tiene la única finalidad de facilitar el estudio y trabajo de los alumnos de la asignatura.

Ni el autor ni la Universidad de Navarra perciben cantidad alguna por su edición o reproducción.

ÍNDICE

1. INTRODUCCIÓN

En estos apuntes se describe de forma abreviada la sintaxis del lenguaje C++. No se trata de aprender a programar en C++, sino más bien de presentar los recursos o las posibilidades que el C++ pone a disposición de los programadores.

Conocer un vocabulario y una gramática no equivale a saber un idioma. Conocer un idioma implica además el hábito de combinar sus elementos de forma semiautomática para producir frases que expresen lo que uno quiere decir. Conocer las palabras, las sentencias y la sintaxis del C++ no equivalen a saber programar, pero son condición necesaria para estar en condiciones de empezar a hacerlo, o de entender cómo funcionan programas ya hechos. El proporcionar la base necesaria para aprender a programar en C++ es el objetivo de estas páginas.

El comité para el estándar ANSI C fue formado en 1983 con el objetivo de crear un lenguaje uniforme a partir del C original, desarrollado por Kernighan y Ritchie en 1972, en la ATT. Hasta entonces el estándar lo marcaba el libro escrito en 1978 por estos dos autores[1].

El lenguaje C++ se comenzó a desarrollar en 1980. Su autor fue B. Stroustrup, también de la ATT. Al comienzo era una extensión del lenguaje C que fue denominada ***C with classes***. Este nuevo lenguaje comenzó a ser utilizado fuera de la ATT en 1983. El nombre ***C++*** es también de ese año, y hace referencia al carácter del operador incremento de C (++). Ante la gran difusión y éxito que iba obteniendo en el mundo de los programadores, la ATT comenzó a estandarizarlo internamente en 1987. En 1989 se formó un comité ANSI (seguido algún tiempo después por un comité ISO) para estandarizarlo a nivel americano e internacional.

En la actualidad, el C++ es un lenguaje versátil, potente y general. Su éxito entre los programadores profesionales le ha llevado a ocupar el primer puesto como herramienta de desarrollo de aplicaciones. El C++ mantiene las ventajas del C en cuanto a riqueza de operadores y expresiones, flexibilidad, concisión y eficiencia. Además, ha eliminado algunas de las dificultades y limitaciones del C original. La evolución de C++ ha continuado con la aparición de ***Java***, un lenguaje creado simplificando algunas cosas de C++ y añadiendo otras, que se utiliza para realizar aplicaciones en Internet.

Hay que señalar que el C++ ha influido en algunos puntos muy importantes del ANSI C, como por ejemplo en la forma de declarar las funciones, en los punteros a void, etc. En efecto, aunque el C++ es posterior al C, sus primeras versiones son anteriores al ANSI C, y algunas de las mejoras de éste fueron tomadas del C++.

En estas ***Notas*** se van a presentar los fundamentos del lenguaje C++. Éste es a la vez un lenguaje procedural (orientado a algoritmos) y orientado a objetos. Como lenguaje procedural se asemeja al C y es compatible con él, aunque ya se ha dicho que presenta ciertas ventajas. Como lenguaje ***orientado a objetos*** se basa en una filosofía completamente diferente, que exige del programador un completo cambio de mentalidad. Las características propias de la ***Programación Orientada a Objetos*** (***Object Oriented Programming***, u ***OOP***) de C++ son modificaciones mayores que sí que cambian radicalmente su naturaleza.

[1] B. Kernighan and D. Ritchie, *The C Programming Language*, Prenctice-Hall, 1978.

1.1. Concepto de "programa"

Un ***programa*** –en sentido informático– está constituido por un conjunto de *instrucciones* que se ejecutan –ordinariamente– de modo *secuencial*, es decir, cada una a continuación de la anterior. Recientemente, con objeto de disminuir los tiempos de ejecución de programas críticos por su tamaño o complejidad, se está haciendo un gran esfuerzo en desarrollar programas *paralelos*, esto es, programas que se pueden ejecutar *simultáneamente* en varios procesadores. La programación paralela es mucho más complicada que la secuencial y no se hará referencia a ella en este curso.

Análogamente a los *datos* que maneja, las *instrucciones* que un procesador digital es capaz de entender están constituidas por conjuntos de *unos* y *ceros*. A esto se llama *lenguaje de máquina* o *binario*, y es muy difícil de manejar. Por ello, desde casi los primeros años de los ordenadores, se comenzaron a desarrollar los llamados *lenguajes de alto nivel* (tales como el **Fortran**, el **Cobol**, etc.), que están mucho más cerca del lenguaje natural. Estos lenguajes están basados en el uso de *identificadores*, tanto para los *datos* como para las componentes elementales del programa, que en algunos lenguajes se llaman *rutinas* o *procedimientos*, y que en C++ se denominan ***funciones***. Además, cada lenguaje dispone de una *sintaxis* o conjunto de reglas con las que se indica de modo inequívoco las operaciones que se quiere realizar.

Los *lenguajes de alto nivel* son más o menos comprensibles para el usuario, pero no para el procesador. Para que éste pueda ejecutarlos es necesario traducirlos *a su propio lenguaje de máquina*. Esta es una tarea que realiza un programa especial llamado ***compilador.*** Esta tarea se suele descomponer en dos etapas, que se pueden realizar juntas o por separado. El programa de alto nivel se suele almacenar en uno o más ficheros llamados ***ficheros fuente***, que en casi todos los *sistemas operativos* se caracterizan por una terminación –también llamada ***extensión***– especial. Así, todos los ficheros fuente de C++ deben terminar por (***.cpp***); ejemplos de nombres de estos ficheros son *calculos.cpp*, *derivada.cpp*, etc. La primera tarea del compilador es realizar una traducción directa del programa a un lenguaje más próximo al del computador (llamado *ensamblador*), produciendo un ***fichero objeto*** con el mismo nombre que el fichero original, pero con la extensión (***.obj***). En una segunda etapa se realiza el proceso de *montaje* (*linkage*) del programa, consistente en producir un ***programa ejecutable*** en lenguaje de máquina, en el que están ya incorporados todos los otros módulos que aporta el sistema sin intervención explícita del programador (funciones de librería, recursos del sistema operativo, etc.). En un PC con sistema operativo ***MS-DOS*** el programa ejecutable se guarda en un fichero con extensión (***.exe***). Este fichero es cargado por el sistema operativo en la memoria RAM cuando el programa va a ser ejecutado.

Una de las ventajas más importantes de los lenguajes de alto nivel es la *portabilidad* de los ficheros fuente resultantes. Quiere esto decir que un programa desarrollado en un PC podrá ser ejecutado en un Macintosh o en una máquina UNIX, con mínimas modificaciones y una simple recompilación.

1.2. Concepto de "función"

1.2.1. CONCEPTOS GENERALES

Las aplicaciones informáticas que habitualmente se utilizan, incluso a nivel de informática personal, suelen contener decenas y aún cientos de miles de líneas de código fuente. A medida que los programas se van desarrollando y aumentan de tamaño, se convertirían rápidamente en sistemas poco manejables si no fuera por la *modularización*, que es el proceso consistente en dividir un

programa muy grande en una serie de módulos mucho más pequeños y manejables. A estos módulos se les ha solido denominar de distintas formas (*subprogramas*, *subrutinas*, *procedimientos*, *funciones*, etc.) según los distintos lenguajes. El lenguaje C++ hace uso del concepto de ***función*** (***function***). Sea cual sea la nomenclatura, la idea es sin embargo siempre la misma: dividir un programa grande en un conjunto de subprogramas o funciones más pequeñas que son llamadas por el programa principal; éstas a su vez llaman a otras funciones más específicas y así sucesivamente.

Las funciones de C++ están implementadas con un particular cuidado y riqueza, constituyendo uno de los aspectos más potentes del lenguaje. Es muy importante entender bien su funcionamiento y sus posibilidades.

De las ventajas de las funciones y de sus características se hablará más tarde, en el apartado 7.

1.2.2. LA FUNCIÓN MAIN( )

Todo programa C++, desde el más pequeño hasta el más complejo, tiene un *programa principal* que es con el que se comienza la ejecución del programa. Este programa principal es también una función, pero una función que está por encima de todas las demás. Esta función se llama **main()** y tiene la forma siguiente:

```
int main(int argc, char *argv[])
{
     sentencia_1
     sentencia_2
     ...
}
```

Aunque el uso habitual es más simple:

```
void main(void)
{
...
}
```

Como ya se ha dicho, más adelante se ampliará la información en torno a las funciones.

1.3. Tokens

Existen seis clases de *componentes sintácticos* o *tokens* en el vocabulario del lenguaje C++: ***palabras clave***, ***identificadores***, ***constantes***, ***cadenas de caracteres***, ***operadores*** y ***separadores***. Los *separadores* –uno o varios espacios en blanco, tabuladores, caracteres de nueva línea (denominados "espacios en blanco" en conjunto), y también los *comentarios* escritos por el programador– se emplean para separar los demás *tokens*; por lo demás son ignorados por el compilador. El compilador descompone el texto fuente o programa en cada uno de sus ***tokens***, y a partir de esta descomposición genera el código objeto correspondiente. El compilador ignora también los sangrados al comienzo de las líneas.

1.3.1. PALABRAS CLAVE DEL C++

En C++, como en cualquier otro lenguaje, existen una serie de palabras clave (*keywords*) que el usuario no puede utilizar como identificadores (nombres de variables y/o de funciones). Estas

palabras sirven para indicar al computador que realice una tarea muy determinada (desde evaluar una comparación, hasta definir el tipo de una variable) y tienen un especial significado para el compilador. El C++ es un lenguaje muy conciso, con muchas menos palabras clave que otros lenguajes. A continuación se presenta la lista de las 48 palabras clave de C++, para las que más adelante se dará detalle de su significado (algunos compiladores añaden otras palabras clave, propias de cada uno de ellos. Es importante evitarlas como identificadores):

```
asm         continue    float       new         signed      try
auto        default     for         operator    sizeof      typedef
break       delete      friend      private     static      union
case        do          goto        protected   struct      unsigned
catch       double      if          public      switch      virtual
char        else        inline      register    template    void
class       enum        int         return      this        volatile
const       extern      long        short       throw       while
```

Además los identificadores que contienen un doble subrayado(--) están reservados para ser utilizados por implantaciones de C++ y bibliotecas estándar. Los usuarios deben evitarlos.

1.3.2. IDENTIFICADORES

Un ***identificador*** es un nombre con el que se hace referencia a una función o al contenido de una zona de la memoria (variable). Cada lenguaje tiene sus propias reglas respecto a las posibilidades de elección de nombres para las funciones y variables. En C++ estas reglas son las siguientes:

1. Un ***identificador*** se forma con una secuencia de ***letras*** (minúsculas de la ***a*** a la ***z***; mayúsculas de la ***A*** a la ***Z***; y ***dígitos*** del ***0*** al ***9***).
2. El carácter ***subrayado*** o ***underscore*** (_) se considera como una letra más.
3. Un identificador no puede contener espacios en blanco, ni otros caracteres distintos de los citados, como por ejemplo (`*,;.:-+,` etc.).
4. El primer carácter de un identificador debe ser siempre una letra o un (_), es decir, no puede ser un dígito.
5. Se hace distinción entre letras mayúsculas y minúsculas. Así, **Masa** es considerado como un identificador distinto de **masa** y de **MASA**.
6. C++ permite definir identificadores con un número ilimitado de caracteres.

   Ejemplos de identificadores válidos son los siguientes:

   ```
   tiempo, distancia1, caso_A, PI, velocidad_de_la_luz
   ```

   Por el contrario, los siguientes nombres no son válidos:

   ```
   1_valor, tiempo-total, %final
   ```

En general es muy aconsejable ***elegir los nombres*** de las funciones y las variables de forma que permitan conocer a simple vista qué tipo de variable o función representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación y –sobre todo– de corrección y mantenimiento de los programas. Es cierto que los nombres largos son más laboriosos de teclear, pero en general resulta rentable tomarse esa pequeña molestia.

1.3.3. CONSTANTES

Las variables pueden cambiar de valor a lo largo de la ejecución de un programa, o bien en ejecuciones distintas de un mismo programa. Además de variables, un programa utiliza también constantes, es decir, valores que siempre son los mismos. En C++ existen distintos tipos de constantes:

Constantes numéricas, constantes carácter, cadenas de caracteres y de tipo ***enumeración***. Todas ellas se tratarán más a fondo en un apartado posterior.

1.3.4. OPERADORES

Los ***operadores*** son signos especiales –a veces, conjuntos de dos caracteres– que indican determinadas operaciones a realizar con las variables y/o constantes sobre las que actúan en el programa. El lenguaje C++ es particularmente rico en distintos tipos de operadores: *aritméticos* (`+`, `-`, `*`, `/`, `%`), *de asignación* (`=`, `+=`, `-=`, `++`, `--`, `*=`, `/=`), *relacionales* (`==`, `<`, `>`, `<=`, `>=`, `!=`), *lógicos* (`&&`, `||`, `!`) y otros. También los ***operadores*** serán vistos con mucho más detalle en apartados posteriores.

1.3.5. SEPARADORES

Como ya se ha comentado, los *separadores* están constituidos por uno o varios espacios en blanco, tabuladores, y caracteres de nueva línea. Su papel es ayudar al compilador a descomponer el programa fuente en cada uno de sus ***tokens***. Es conveniente introducir espacios en blanco incluso cuando no son estrictamente necesarios, con objeto de mejorar la legibilidad de los programas.

1.3.6. COMENTARIOS

El lenguaje C++ permite que el programador introduzca ***comentarios*** en los ficheros fuente que contienen el código de su programa. La misión de los comentarios es servir de explicación o aclaración sobre cómo está hecho el programa, de forma que pueda ser entendido por una persona diferente (o por el propio programador algún tiempo después). El compilador ignora por completo los comentarios.

Los caracteres (`/*`) se emplean para iniciar un comentario introducido entre el código del programa; el comentario termina con los caracteres (`*/`). Todo texto introducido entre los símbolos de comienzo (`/*`) y final (`*/`) de comentario son siempre ignorados por el compilador. Por ejemplo:

```
/* Esto es un comentario simple. */

/* Esto es un comentario más largo,
   distribuido en varias líneas. El
   texto se suele alinear por la izquierda. */
```

Los comentarios pueden actuar también como *separadores* de otros tokens propios del lenguaje C++. Una fuente frecuente de errores –no especialmente difíciles de detectar– al programar en C++, es el olvidarse de cerrar un comentario que se ha abierto previamente.

Además se considera que son comentarios todo aquel texto que está desde dos barras consecutivas (//) hasta el fin de la línea. Las dos barras marcan el comienzo del comentario y el fin de la línea, el final. Si se desea poner comentarios de varias líneas, hay que colocar la doble barra al comienzo de cada línea. Los ejemplos anteriores se podrían escribir del siguiente modo:

```
// Esto es un comentario simple.

// Esto es un comentario más largo,
//    distribuido en varias líneas. El
//    texto se suele indentar por la izquierda.
```

1.4. Lenguaje C++

En las páginas anteriores ya han ido apareciendo algunas características importantes del lenguaje C++. En realidad, *el lenguaje C++ está constituido por tres elementos*: el ***compilador***, el ***preprocesador*** y la ***librería estándar***. A continuación se explica brevemente en qué consiste cada uno de estos elementos.

1.4.1. COMPILADOR

El compilador es el elemento más característico del lenguaje C++. Como ya se ha dicho anteriormente, su misión consiste en traducir a lenguaje de máquina el programa C++ contenido en uno o más ficheros fuente. El compilador es capaz de detectar ciertos errores durante el proceso de compilación, enviando al usuario el correspondiente mensaje de error.

1.4.2. PREPROCESADOR

El preprocesador es un componente característico de C++, que no existe en otros lenguajes de programación. El preprocesador actúa sobre el programa fuente, antes de que empiece la compilación propiamente dicha, para realizar ciertas operaciones. Una de estas operaciones es, por ejemplo, la sustitución de *constantes simbólicas*. Así, es posible que un programa haga uso repetidas veces del valor 3.141592654, correspondiente al número π. Es posible definir una constante simbólica llamada PI que se define como 3.141592654 al comienzo del programa y se introduce luego en el código cada vez que hace falta. En realidad PI no es una variable con un determinado valor: el preprocesador chequea todo el programa antes de comenzar la compilación y sustituye el texto PI por el texto 3.141592654 cada vez que lo encuentra. Las constantes simbólicas suelen escribirse completamente con mayúsculas, para distinguirlas de las variables.

El preprocesador realiza muchas otras funciones que se irán viendo a medida que se vaya explicando el lenguaje. Lo importante es recordar que actúa siempre por delante del compilador (de ahí su nombre), facilitando su tarea y la del programador.

1.4.3. LIBRERÍA ESTÁNDAR

Con objeto de mantener el lenguaje lo más sencillo posible, muchas sentencias que existen en otros lenguajes, no tienen su correspondiente contrapartida en C++. Por ejemplo, en C++ no hay sentencias para entrada y salida de datos. Es evidente, sin embargo, que esta es una funcionalidad que hay que cubrir de alguna manera. El lenguaje C++ lo hace por medio de ***funciones*** preprogramadas que se venden o se entregan junto con el compilador. Estas funciones *están agrupadas en un conjunto de librerías de código objeto*, que constituyen la llamada ***librería estándar*** del lenguaje. La llamada a dichas funciones se hace como a otras funciones cualesquiera, y *deben ser declaradas* antes de ser llamadas por el programa (más adelante se verá cómo se hace esto por medio de la directiva del preprocesador **#include**).

Se incluye una tabla con las librerías estándar en el apartado 15.

1.5. Ficheros

El código de cualquier programa escrito en C++ se almacena en uno o más ficheros, en el disco del ordenador. La magnitud del programa y su estructura interna determina o aconseja sobre el número de ficheros a utilizar. Como se verá más adelante, la división de un programa en varios ficheros es una forma de controlar su manejo y su modularidad. Cuando los programas son pequeños (hasta 50≈100 líneas de código), un solo fichero suele bastar. Para programas más grandes, y cuando se quiere mantener más independencia entre los distintos subprogramas, es conveniente repartir el código entre varios ficheros.

Recuérdese además que *cada vez que se introduce un cambio en el programa hay que volver a compilarlo*. La compilación se realiza a nivel de fichero, por lo que sólo los ficheros modificados deben ser compilados de nuevo. Si el programa está repartido entre varios ficheros pequeños esta operación se realiza mucho más rápidamente.

1.6. Lectura y escritura de datos

La lectura y escritura de datos se realiza por medio de operadores que se encuentran en una librería llamada ***iostream*** (input/output stream). Las declaraciones de los operadores de esta librería están en un fichero llamado ***iostream.h***. En C++ las entradas son leídas desde ***streams*** y las salidas son escritas en ***streams***. La palabra ***stream*** quiere decir algo así como *canal*, *flujo* o *corriente.*

2. TIPOS DE DATOS FUNDAMENTALES. VARIABLES

El C++, como cualquier otro lenguaje de programación, tiene posibilidad de trabajar con datos de distinta naturaleza: texto formado por caracteres alfanuméricos, números enteros, números reales con parte entera y parte fraccionaria, etc. Además, algunos de estos tipos de datos admiten distintos números de cifras (rango y/o precisión), posibilidad de ser sólo positivos o de ser positivos y negativos, etc. En este apartado se verán los *tipos fundamentales* de datos admitidos por el C++. Más adelante se verá que hay otros tipos de datos, *derivados* de los fundamentales. Los tipos de datos fundamentales del C++ se indican en la Tabla 2.1.

Tabla 2.1. Tipos de datos fundamentales (notación completa)

Datos enteros	char	signed char	unsigned char
	signed short int	signed int	signed long int
	unsigned short int	unsigned int	unsigned long int
Datos reales	float	double	long double

La palabra ***char*** hace referencia a que se trata de un carácter (una letra mayúscula o minúscula, un dígito, un carácter especial, ...). La palabra ***int*** indica que se trata de un número entero, mientras que ***float*** se refiere a un número real (también llamado de punto o coma flotante). Los números enteros pueden ser positivos o negativos (***signed***), o bien esencialmente no negativos (***unsigned***); los caracteres tienen un tratamiento muy similar a los enteros y admiten estos mismos cualificadores. En los datos enteros, las palabras ***short*** y ***long*** hacen referencia al número de cifras o rango de dichos números. En los datos reales las palabras ***double*** y ***long*** apuntan en esta misma dirección, aunque con un significado ligeramente diferente, como más adelante se verá.

Esta nomenclatura puede simplificarse: las palabras ***signed*** e ***int*** son las opciones por defecto para los números enteros y pueden omitirse, resultando la Tabla 2.2, que indica la nomenclatura más habitual para los tipos fundamentales del C++.

Tabla 2.2. Tipos de datos fundamentales (notación abreviada).

Datos enteros	Char	signed char	unsigned char
	Short	int	long
	unsigned short	unsigned	unsigned long
Datos reales	Float	double	long double

A continuación se va a explicar cómo puede ser y cómo se almacena en C++ un dato de cada tipo fundamental.

Recuérdese que *en C++ es necesario declarar todas las variables que se vayan a utilizar*. Una variable no declarada produce un mensaje de error en la compilación. Cuando una variable es declarada se le reserva memoria de acuerdo con el *tipo* incluido en la declaración. Es posible *inicializar* –dar un valor inicial– las variables en el momento de la declaración; ya se verá que en ciertas ocasiones el compilador da un valor inicial por defecto, mientras que en otros casos no se realiza esta inicialización y la memoria asociada con la variable correspondiente contiene

basura informática (combinaciones sin sentido de unos y ceros, resultado de operaciones anteriores con esa zona de la memoria, para otros fines).

2.1. Caracteres (tipo *char*)

Las variables carácter (***tipo char***) contienen un único carácter y se almacenan en un ***byte*** de memoria (8 bits). En un bit se pueden almacenar dos valores (0 y 1); con dos bits se pueden almacenar $2^2 = 4$ valores (00, 01, 10, 11 en binario; 0, 1, 2, 3 en decimal). Con 8 bits se podrán almacenar $2^8 = 256$ valores diferentes (normalmente entre 0 y 255; con ciertos compiladores entre -128 y 127).

La declaración de variables tipo carácter puede tener la forma:

```
char nombre;
char nombre1, nombre2, nombre3;
```

Se puede declarar más de una variable de un tipo determinado en una sola sentencia. Se puede también inicializar la variable en la declaración. Por ejemplo, para definir la variable carácter **letra** y asignarle el valor **a**, se puede escribir:

```
char letra = 'a';
```

A partir de ese momento queda definida la variable **letra** con el valor correspondiente a la letra **a**. Recuérdese que el valor **'a'** utilizado para inicializar la variable **letra** es una constante carácter. En realidad, **letra** se guarda en un solo byte como un número entero, el correspondiente a la letra **a** en el código ASCII, que se muestra en la Tabla 2.3 para los caracteres estándar (existe un código ASCII extendido que utiliza los 256 valores y que contiene caracteres especiales y caracteres específicos de los alfabetos de diversos países, como por ejemplo las *vocales acentuadas* y la *letra ñ* para el castellano).

Tabla 2.3. Código ASCII estándar.

	0	1	2	3	4	5	6	7	8	9
0	nul	Soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	Vt	np	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	Us	sp	!	“	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	E	f	G	h	i	j	k	l	m
11	n	O	p	Q	r	s	t	u	v	w
12	x	Y	z	{	|	}	~	del		

La Tabla 2.3 se utiliza de la siguiente forma. La(s) primera(s) cifra(s) del número ASCII correspondiente a un carácter determinado figura en la primera columna de la Tabla, y la última cifra en la primera fila de la Tabla. Sabiendo la fila y la columna en la que está un determinado carácter puede componerse el número correspondiente. Por ejemplo, la letra A está en la fila 6 y la columna

5. Su número ASCII es por tanto el 65. El carácter % está en la fila 3 y la columna 7, por lo que su representación ASCII será el 37. Obsérvese que el código ASCII asocia números consecutivos con las letras mayúsculas y minúsculas ordenadas alfabéticamente. Esto simplifica notablemente ciertas operaciones de ordenación alfabética de nombres.

En la Tabla 2.3 aparecen muchos caracteres no imprimibles (todos aquellos que se expresan con 2 ó 3 letras). Por ejemplo, el **ht** es el tabulador horizontal, el **nl** es el carácter *nueva línea*, etc.

Volviendo al ejemplo de la variable **letra**, su contenido puede ser variado cuando se desee por medio de una sentencia que le asigne otro valor, por ejemplo:

```
letra = 'z';
```

También puede utilizarse una variable ***char*** para dar valor a otra variable de tipo ***char*:**

```
caracter = letra;       /* Ahora caracter es igual a 'z' */
```

Como una variable tipo ***char*** es un número entero pequeño (entre 0 y 255), se puede utilizar el contenido de una variable ***char*** de la misma forma que se utiliza un entero, por lo que están permitidas operaciones como:

```
letra = letra + 1;
letra_min = letra_may + ('a' - 'A');
```

En el primer ejemplo, si el contenido de **letra** era una **a**, al incrementarse en una unidad pasa a contener una **b**. El segundo ejemplo es interesante: puesto que la diferencia numérica entre las letras minúsculas y mayúsculas es siempre la misma (según el código ASCII), la segunda sentencia pasa una letra mayúscula a la correspondiente letra minúscula sumándole dicha diferencia numérica.

Recuérdese para concluir que las variables tipo ***char*** son y se almacenan como números enteros pequeños. Ya se verá más adelante que se pueden escribir como caracteres o como números según que formato de conversión se utilice en la llamada a la función de escritura.

También se puede reservar 1 byte de memoria mediante variables tipo ***bool***. Aunque realmente no sean un tipo de variable, C++ permite declarlarlas porque se les da un uso muy específico.

En una variable tipo ***bool*** se almacena un único valor en ese byte de memoria. Este valor puede ser 0 ó 1, donde el 0 corresponde a la sentencia *false* y el uno corresponde a la sentencia *true*.

Estas variables no se utilizan para almacenar datos, sino para controlar el flujo de ejecución, es decir, para dirigir las bifurcaciones y los bucles.

Una variable bool se declara e inicializa de la siguiente manera:

```
bool opc = 0;            //false=0

bool opt;

opt = true;        //true=1
```

2.2. Números enteros (tipo *int*)

De ordinario una variable tipo ***int*** se almacena en 4 bytes (32 bits).

Con 32 bits se pueden almacenar 2^{32} = 4294967296 números enteros diferentes: de 0 al 4294967296 para variables sin signo, y de -2147483648 al 2147483647 para variables que pueden ser positivas y negativas, que es la opción por defecto. Este es el ***rango*** de las variables tipo ***int***.

Una variable entera (tipo ***int***) se declara, o se declara y se inicializa en la forma:

```
unsigned int numero;
int nota = 10;
```

En este caso `numero` podrá estar entre 0 y 4294967296, mientras que `nota` deberá estar comprendida entre -2147483648 y 2147483647. Cuando a una variable ***int*** se le asigna en tiempo de ejecución un valor que queda fuera del rango permitido (situación de ***overflow*** o valor excesivo), se produce un error en el resultado de consecuencias tanto más imprevisibles cuanto que de ordinario *el programa no avisa al usuario de dicha circunstancia*.

2.3. Números enteros (tipo *long*)

Aunque lo habitual es que una variable tipo ***int*** ocupe 4 bytes, en algunos compiladores no se le asignan más que 2. Para evitar este problema se puede anteponer la palabra ***long***, que asegura que la variable alcance esos 4 bytes.

De esta manera el rango de la variable será el correspondiente al tipo ***int*** habitual.

2.4. Números enteros (tipo *short*)

Existe la posibilidad de utilizar enteros con un rango menor si se especifica como tipo ***short*** en su declaración:

```
short int numero_pequeño;
```

o, ya que la palabra clave ***int*** puede omitirse en este caso,

```
short numero_pequeño;
```

El rango de un entero ***short*** puede variar según el computador o el compilador que se utilice, pero de ordinario se utilizan 2 bytes (16 bits) para almacenarlos, por lo que se pueden representar 2^{16} = 65.536 números enteros diferentes. Si se utilizan números con signo, podrán representarse números entre –32.768 y 32.767. También se pueden declarar enteros ***short*** que sean siempre positivos con la palabra ***unsigned***:

```
unsigned short numero_positivo_mayor;
```

En algunos computadores una variable ***int*** ocupa 2 bytes (coincidiendo con ***short***) y en otros 4 bytes (coincidiendo con ***long***). Lo que se garantiza es que el rango de ***int*** no es menor que el de ***short*** ni mayor que el de ***long***.

2.5. Números reales (tipo *float*)

En muchas aplicaciones hacen falta variables reales, capaces de representar magnitudes que contengan *una parte entera y una parte fraccionaria*. Estas variables se llaman también de ***punto flotante***. De ordinario, en base 10 y con notación científica, estas variables se representan por medio de la ***mantisa***, que es un número mayor o igual que 0.1 y menor que 1.0, y un ***exponente*** que representa la potencia de 10 por la que hay que multiplicar la mantisa para obtener el número

considerado. Por ejemplo, π se representa como $0.3141592654 \cdot 10^{1}$. Tanto la mantisa como el exponente pueden ser positivos y negativos.

Los computadores trabajan en base 2. Por eso un número de tipo ***float*** se almacena en 4 bytes (32 bits), utilizando *24 bits para la mantisa* (1 para el signo y 23 para el valor) y *8 bits para el exponente* (1 para el signo y 7 para el valor). Es interesante ver qué clase de números de punto flotante pueden representarse de esta forma. En este caso hay que distinguir el ***rango*** de la ***precisión***. La *precisión* hace referencia al número de cifras con las que se representa la *mantisa*: con 23 bits el número más grande que se puede representar es,

$$2^{23} = 8.388.608$$

lo cual quiere decir que se pueden representar todos los números decimales de 6 cifras y la mayor parte –aunque no todos– de los de 7 cifras (por ejemplo, el número 9.213.456 no se puede representar con 23 bits). Por eso se dice que las variables tipo ***float*** tienen entre 6 y 7 cifras decimales equivalentes de precisión.

Respecto al *exponente de dos* por el que hay que multiplicar la *mantisa* en base 2, con 7 bits el número más grande que se puede representar es 128. El ***rango*** vendrá definido por la potencia,

$$2^{128} = 3.4028 \cdot 10^{38}$$

lo cual indica el número más grande representable de esta forma. El número más pequeño en valor absoluto será del orden de

$$2^{-127} = 5.8775 \cdot 10^{-39}$$

Las variables tipo ***float*** se declaran de la forma:

```
float numero_real;
```

Las variables tipo ***float*** pueden ser inicializadas en el momento de la declaración, de forma análoga a las variables tipo ***int***.

2.6. Números reales (tipo *double*)

Las variables tipo ***float*** tienen un *rango* y –sobre todo– una *precisión* muy limitada, insuficiente para la mayor parte de los cálculos técnicos y científicos. Este problema se soluciona con el tipo ***double***, que utiliza 8 bytes (64 bits) para almacenar una variable. Se utilizan *53 bits para la mantisa* (1 para el signo y 52 para el valor) y *11 para el exponente* (1 para el signo y 10 para el valor). La ***precisión*** es en este caso,

$$2^{52} = 4.503.599.627.370.496$$

lo cual representa entre 15 y 16 cifras decimales equivalentes. Con respecto al ***rango***, con un exponente de 10 bits el número más grande que se puede representar será del orden de 2 elevado a 2 elevado a 10 (que es 1024):

$$21024 = 1.7976 \cdot 10308$$

Las variables tipo ***double*** se declaran de forma análoga a las anteriores:

```
double real_grande;
```

Por último, existe la posibilidad de declarar una variable como ***long double***, aunque el C++ no garantiza un *rango* y una *precisión* mayores que las de ***double***. Eso depende del compilador y del tipo de computador. Estas variables se declaran en la forma:

```
long double real_pero_que_muy_grande;
```

cuyo *rango* y *precisión* no está normalizado.

2.7. Duración y visibilidad de las variables: Modos de almacenamiento

El ***tipo*** de una variable se refiere a la naturaleza de la información que contiene (ya se han visto los tipos *char*, *int*, *long*, *float*, *double*, etc.).

El ***modo de almacenamiento*** (storage class) es otra característica de las variables de C++ que determina cuándo se crea una variable, cuándo deja de existir y desde dónde se puede acceder a ella, es decir, desde dónde es visible.

Las variables pueden ser declaradas *en cualquier lugar de un bloque*[2]. Esto permite acercar la declaración de las variables al lugar en que se utilizan por primera vez.

Un caso importante son los bucles ***for***. La variable que sirve de contador al bucle puede declararse e inicializarse en la propia sentencia ***for.***

En C++ existen 4 ***modos de almacenamiento*** fundamentales: *auto*, *extern*, *static* y *register*. Seguidamente se exponen las características de cada uno de estos modos.

1. ***auto*** (automático). Es la opción por defecto para las variables que se declaran dentro de un bloque, incluido el que contiene el código de las funciones. La declaración debe estar siempre al comienzo del bloque. No es necesario poner la palabra *auto*. Cada variable *auto* es creada al comenzar a ejecutarse el bloque y deja de existir cuando el bloque se termina de ejecutar. Cada vez que se ejecuta el bloque, las variables *auto* se crean y se destruyen de nuevo. Las variables *auto* son variables *locales*, es decir, sólo son ***visibles*** en el bloque en el que están definidas y en otros bloques *anidados*[3] en él, aunque pueden ser ocultadas por una nueva declaración de una nueva variable con el mismo nombre en un bloque anidado. *No son inicializadas por defecto*, y –antes de que el programa les asigne un valor– pueden contener *basura informática*.

   A continuación se muestra un ejemplo de uso de variables de modo *auto*.

```
{
     int i=1, j=2;              /* se declaran e inicializan i y j */
     ...
     {
          float a=7., j=3.;     /* se declara una nueva variable j */
          ...
          j=j+a;                /* aqui j es float */
          ...                   /* la variable int j es invisible */
          ...                   /* la variable i=1 es visible */
     }
     ...                        /* fuera del bloque, a ya no existe */
     ...                        /* la variable j=2 existe y es entera */
}
```

2. ***extern***. Son variables globales, que se definen fuera de cualquier bloque o función, por ejemplo antes de definir la función **main()**. Estas variables *existen durante toda la ejecución*

[2] Un ***bloque*** es una unidad básica de agrupamiento de declaraciones e instrucciones encerrada entre llaves ({}).

[3] Se llama *bloque anidado* a un bloque contenido dentro de otro bloque.

del programa. Las variables *extern* son *visibles* por todas las funciones que están entre la definición y el fin del fichero. Para verlas desde otras funciones definidas anteriormente o desde otros ficheros, deben ser declaradas en ellos como variables *extern*. Por defecto, *son inicializadas a cero*.

Una variable *extern* es *definida o creada* (el momento en el que se le reserva memoria y se le asigna un valor) una sola vez, pero puede ser *declarada* (es decir, reconocida para poder ser utilizada) varias veces, con objeto de hacerla accesible desde diversas funciones o ficheros. También estas variables pueden ocultarse mediante la declaración de otra variable con el mismo nombre en el interior de un bloque. Aun así C++ dispone del operador (::) llamado operador de resolución de visibilidad. Este operador, antepuesto al nombre de una variable global que está oculta por una variable local con el mismo nombre, permite acceder al valor de la variable global.

La variables *extern* permiten transmitir valores entre distintas funciones, pero ésta es una práctica considerada como *peligrosa*. A continuación se presenta un ejemplo de uso de variables *extern*.

```
int i=1, j, k;                          /* se declaran antes de main() */

void main()
{
     int i=3;                       /* i=1 se hace invisible */
     int func1(int, int);
     ...                            /* j, k son visibles */
}

int func1(int i, int m)
{
     int k=3;                       /* k=0 se hace invisible */
     ...                            /* i=1 es invisible */
}
```

3. ***static***. Cuando ciertas variables son declaradas como *static* dentro de un bloque, estas variables conservan su valor entre distintas ejecuciones de ese bloque. Dicho de otra forma, las variables *static* se declaran dentro de un bloque como las *auto*, pero permanecen en memoria durante toda la ejecución del programa como las *extern*. Cuando se llama varias veces sucesivas a una función (o se ejecuta varias veces un bloque) que tiene declaradas variables *static*, los valores de dichas variables se conservan entre dichas llamadas. La inicialización sólo se realiza la primera vez. Por defecto, son inicializadas a cero.

   Las variables definidas como ***static extern*** son visibles sólo para las funciones y bloques comprendidos desde su definición hasta el fin del fichero. No son visibles desde otras funciones ni aunque se declaren como *extern*. Ésta es una forma de restringir la visibilidad de las variables.

   Por defecto, y por lo que respecta a su visibilidad, las ***funciones*** tienen modo *extern*. Una función puede también ser definida como *static*, y entonces sólo es visible para las funciones que están definidas después de dicha función y en el mismo fichero. Con estos modos se puede controlar la visibilidad de una función, es decir, desde qué otras funciones puede ser llamada.

4. ***register***. Este modo es una recomendación para el compilador, con objeto de que –si es posible– ciertas variables sean almacenadas en los registros de la CPU y los cálculos con ellas sean más rápidos. No existen los modos *auto* y *register* para las funciones.

2.8. Conversiones implícitas y explícitas de tipo (casting)

Las *conversiones implícitas de tipo* tienen lugar cuando en una expresión se mezclan variables de distintos tipos. Por ejemplo, para poder sumar dos variables hace falta que ambas sean del mismo tipo. Si una es *int* y otra *float*, la primera se convierte a *float* (es decir, la variable del tipo de menor rango se convierte al tipo de mayor rango), antes de realizar la operación. A esta conversión automática e implícita de tipo (el programador no necesita intervenir, aunque sí conocer sus reglas), se le denomina ***promoción***, pues la variable de menor rango es *promocionada* al rango de la otra.

Los rangos de las variables de mayor a menor se ordenan del siguiente modo:

long double > double > float > unsigned long > long > unsigned int > int > unsigned short > short > char

Otra clase de conversión implícita tiene lugar cuando el resultado de una expresión es asignado a una variable, pues dicho resultado se convierte al tipo de la variable (en este caso, ésta puede ser de menor rango que la expresión, por lo que esta conversión puede perder información y ser peligrosa).

En C++ existe también la posibilidad de realizar *conversiones explícitas de tipo* (llamadas ***casting***). El casting es pues una conversión de tipo, forzada por el programador. Para ello basta preceder la constante, variable o expresión que se desea convertir por el tipo al que se desea convertir, encerrado entre paréntesis. En el siguiente ejemplo,

```
k = (int) 1.7 + (int) masa;
```

la variable **masa** es convertida a tipo *int*, y la constante **1.7** (que es de tipo *double*) también.

El lenguaje C++ dispone de otra conversión explícita de tipo con una notación similar a la de las funciones y más sencilla que la del ***cast***. Se utiliza para ello el nombre del tipo al que se desea convertir seguido del valor a convertir entre paréntesis. Así, las siguientes expresiones son válidas en C++:

```
y = double(25);
double x = 5;
return int(x/y);
```

2.9. Typedef

Esta palabra reservada del lenguaje C++ sirve para la creación de nuevos nombres de tipos de datos. Mediante esta declaración es posible que el usuario defina una serie de tipos de variables propios, no incorporados en el lenguaje y que se forman a partir de tipos de datos ya existentes. Por ejemplo, la declaración:

```
typedef int ENTERO;
```

define un tipo de variable llamado **ENTERO** que corresponde a *int*.

El comando **typedef** ayuda a parametrizar un programa contra problemas de portabilidad. Generalmente se utiliza **typedef** para los tipos de datos que pueden ser dependientes de la instalación. También puede ayudar a documentar el programa, haciéndolo más legible.

3. CONSTANTES

Se entiende por ***constantes*** aquel tipo de información numérica o alfanumérica que no puede cambiar más que con una nueva compilación del programa. Como ya se ha dicho anteriormente, en el código de un programa en C++ pueden aparecer diversos tipos de constantes que se van a explicar a continuación.

3.1. Constantes numéricas

3.1.1. CONSTANTES ENTERAS

Una ***constante entera decimal*** está formada por una secuencia de dígitos del 0 al 9, constituyendo un número entero. Las constantes enteras decimales están sujetas a las mismas restricciones de rango que las variables tipo ***int*** y ***long***, pudiendo también ser ***unsigned***. El tipo de una constante se puede determinar automáticamente según su magnitud, o de modo explícito postponiendo ciertos caracteres, como en los ejemplos que siguen:

```
23484              constante tipo int
45815              constante tipo long (es mayor que 32767)
253u ó 253U        constante tipo unsigned int
739l ó 739L        constante tipo long
583ul ó 583UL      constante tipo unsigned long
```

En C++ se pueden definir también ***constantes enteras octales***, esto es, expresadas en base 8 con dígitos del 0 al 7. Se considera que una constante está expresada en base 8 si el primer dígito por la izquierda es un cero (0). Análogamente, una secuencia de dígitos (del 0 al 9) y de letras (A, B, C, D, E y F o minúsculas) precedida por `0x` o por `0X`, se interpreta como una ***constante entera hexadecimal***, esto es, en base 16. Por ejemplo:

```
011           constante octal (igual a 9 en base 10)
11            constante entera decimal (no es igual a 011)
0xA           constante haxadecimal (igual a 10 en base 10)
0xFF          constante hexadecimal (igual a 16²-1=255 en base 10)
```

Es probable que no haya necesidad de utilizar constantes octales y hexadecimales, pero conviene conocer su existencia y saber interpretarlas por si hiciera falta. La ventaja de los números expresados en base 8 y base 16 proviene de su estrecha relación con la base 2 (8 y 16 son potencias de 2), que es la forma en la que el ordenador almacena la información.

3.1.2. CONSTANTES DE PUNTO FLOTANTE

Como es natural, existen también ***constantes de punto flotante***, que pueden ser de tipo ***float***, ***double*** y ***long double***. Una constante de punto flotante se almacena de la misma forma que la variable correspondiente del mismo tipo. Por defecto –si no se indica otra cosa– las constantes de punto flotante son de tipo ***double***. Para indicar que una constante es de tipo ***float*** se le añade una **f** o una **F**; para indicar que es de tipo ***long double***, se le añade una **l** o una **L**. En cualquier caso, el punto decimal siempre debe estar presente si se trata de representar un número real.

Estas constantes se pueden expresar de varias formas. La más sencilla es un conjunto de dígitos del 0 al 9, incluyendo un punto decimal. Para constantes muy grandes o muy pequeñas puede utilizarse la *notación científica*; en este caso la constante tiene una parte entera, un punto decimal, una parte fraccionaria, una e o E, y un exponente entero (afectando a la base 10), con un signo

opcional. Se puede omitir la parte entera o la fraccionaria, pero no ambas a la vez. Las constantes de punto flotante son siempre positivas. Puede anteponerse un signo (-), pero no forma parte de la constante, sino que con ésta constituye una ***expresión***, como se verá más adelante. A continuación se presentan algunos ejemplos válidos:

```
1.23          constante tipo double (opción por defecto)
23.963f       constante tipo float
.00874        constante tipo double
23e2          constante tipo double (igual a 2300.0)
.874e-2       constante tipo double en notación científica (=.00874)
.874e-2f      constante tipo float en notación científica
```

seguidos de otros que no son correctos:

```
1,23          error: la coma no esta permitida
23963f        error: no hay punto decimal ni carácter e ó E
.e4           error: no hay ni parte entera ni fraccionaria
```

3.2. Constantes carácter

Una constante carácter es un carácter cualquiera encerrado entre apóstrofos (tal como 'x' o 't'). El valor de una constante carácter es el valor numérico asignado a ese carácter según el código ASCII (ver Tabla 2.3). Conviene indicar que en C++ no existen constantes tipo ***char***; lo que se llama aquí ***constantes carácter*** son en realidad constantes enteras (equivalente ASCII).

Hay que señalar que el valor ASCII de los números no coincide con el propio valor numérico. Por ejemplo, el valor ASCII de la constante carácter '7' es 55.

Ciertos caracteres no representables gráficamente, el apóstrofo (') y la barra invertida (\) y otros caracteres, se representan mediante la siguiente tabla de ***secuencias de escape***, con ayuda de la barra invertida (\)[4]

Nombre completo	Constante	en C	ASCII
sonido de alerta	BEL	\a	7
nueva línea	NL	\n	10
tabulador horizontal	HT	\t	9
tabulador vertical	VT	\v	11
retroceso	BS	\b	8
retorno de carro	CR	\r	13
salto de página	FF	\f	12
barra invertida	\	\\	92
apóstrofo	'	\'	39
comillas	"	\"	34
signo de interrogación	?	\?	63
carácter nulo	NULL	\0	0
número octal	000	\000	
número hexadecimal	hhh	\xhhh	

Los caracteres ASCII pueden ser también representados mediante el número octal correspondiente (hasta tres dígitos), encerrado entre apóstrofos y precedido por la barra invertida. Por ejemplo, '\07' y '\7' representan el número 7 del código ASCII ('\007' es la representación octal

[4] Una *secuencia de escape* está constituida por la barra invertida (\) seguida de otro carácter. La finalidad de la secuencia de escape es cambiar el significado habitual del carácter que sigue a la barra invertida.

del carácter '7'), que es el sonido de alerta. El C++ también admite secuencias de escape hexadecimales (sin límite de dígitos), por ejemplo '\x1a'.

3.3. Cadenas de caracteres

Una cadena de caracteres es una secuencia de caracteres delimitada por comillas ("), como por ejemplo: "Esto es una cadena de caracteres". Dentro de la cadena, pueden aparecer caracteres en blanco y se pueden emplear las mismas secuencias de escape válidas para las constantes carácter. Por ejemplo, las comillas (") deben estar precedidas por (\), para no ser interpretadas como fin de la cadena; también la propia barra invertida (\). Es muy importante señalar que el compilador sitúa siempre un byte nulo (\0) adicional al final de cada cadena de caracteres para señalar el final de esta. Así, la cadena "mesa" no ocupa 4 bytes, sino 5 bytes. A continuación se muestran algunos ejemplos de cadenas de caracteres:

```
"Informática I"
"'A'"
"        cadena con espacios en blanco        "
"Esto es una \"cadena de caracteres\".\n"
```

3.4. Constantes de tipo Enumeración

En C++ existen una clase especial de constantes, llamadas constantes ***enumeración***. Estas constantes se utilizan para definir los posibles valores de ciertos identificadores o variables que sólo deben poder tomar unos pocos valores. Por ejemplo, se puede pensar en una variable llamada **dia_de_la_semana** que sólo pueda tomar los 7 valores siguientes: **lunes**, **martes**, **miercoles**, **jueves**, **viernes**, **sabado** y **domingo**. Es muy fácil imaginar otros tipos de variables análogas, una de las cuales podría ser una variable booleana con sólo dos posibles valores: SI y NO, o TRUE y FALSE, u ON y OFF. El uso de este tipo de variables hace más claros y legibles los programas, a la par que disminuye la probabilidad de introducir errores.

En realidad, las constantes ***enumeración*** son los posibles valores de ciertas variables definidas como de ese tipo concreto. Considérese como ejemplo la siguiente declaración:

```
enum dia {lunes, martes, miercoles, jueves, viernes, sabado, domingo};
```

Esta declaración crea un nuevo ***tipo de variable*** –el tipo de variable ***dia***– que sólo puede tomar uno de los 7 valores encerrados entre las llaves.

```
enum dia  dia1, dia2;
```

y a estas variables se les pueden asignar valores en la forma

```
dia1 = martes;
```

o aparecer en diversos tipos de expresiones y de sentencias que se explicarán más adelante. A cada enumerador se le asigna un valor entero. Por omisión, estos valores parten de cero con un incremento unitario. Además, el programador puede controlar la asociación de esos valores como aparece a continuación,

```
enum dia {lunes=1, martes, miercoles, jueves, viernes, sabado, domingo};
```

asocia un valor 1 a **lunes**, 2 a **martes**, 3 a **miercoles**, etc., mientras que la declaración,

```
enum dia {lunes=1, martes, miercoles, jueves=7, viernes, sabado, domingo};
```

asocia un valor 1 a **lunes**, 2 a **martes**, 3 a **miercoles**, un 7 a **jueves**, un 8 a **viernes**, un 9 a **sabado** y un 10 a **domingo**.

Esta asociación no conlleva que las variables tipo ***enum*** se comporten como enteros, son un nuevo tipo de variables que necesitan un ***cast*** para que un valor entero les pueda ser asignado.

Se puede también hacer la definición del tipo ***enum*** y la declaración de las variables en una única sentencia, en la forma

```
enum palo {oros, copas, espadas, bastos}  carta1, carta2, carta3;
```

donde **carta1**, **carta2** y **carta3** son variables que sólo pueden tomar los valores ***oros***, ***copas***, ***espadas*** y ***bastos*** (equivalentes respectivamente a 0, 1, 2 y 3).

3.5. Cualificador const

En C++ el especificador ***const*** se puede utilizar con variables y con punteros. Las variables definidas como ***const*** no son lo mismo que las *constantes simbólicas*, aunque evidentemente hay una cierta similitud en las áreas de aplicación. Si una variable se define como ***const*** se tiene la garantía de que su valor no va a cambiar durante toda la ejecución del programa. Si en alguna sentencia del programa se intenta variar el valor de una variable definida como ***const***, el compilador produce un mensaje de error. Esta precaución permite detectar errores durante la compilación del programa, lo cual siempre es más sencillo que detectarlos en tiempo de ejecución.

Las variables de este tipo pueden ser inicializadas pero no pueden estar a la izquierda de una sentencia de asignación.

Las variables declaradas como ***const*** tienen importantes diferencias con las constantes simbólicas definidas con la directiva ***#define*** del preprocesador. Aunque ambas representan valores que no se puede modificar, las variables ***const*** están sometidas a las mismas reglas de visibilidad y duración que las demás variables del lenguaje.

```
void main(void)
{
    const int SIZE = 5;
    char  cs[SIZE] ;          /*al ser constante se puede utilizar la
                              variable SIZE para determinar el tamaño
                              del vector cs*/
}
```

En el caso de los punteros hay que distinguir entre dos formas de aplicar el cualificador ***const***:

1. un ***puntero variable*** apuntando a una ***variable constante***.
2. un ***puntero constante*** apuntando a una ***variable cualquiera***.

Un puntero a una variable ***const*** no puede modificar el valor de esa variable (si se intentase el compilador lo detectaría e imprimiría un mensaje de error), pero ese puntero no tiene por qué apuntar siempre a la misma variable.

En el caso de un puntero ***const***, éste apunta siempre a la misma dirección de memoria pero el valor de la variable almacenada en esa dirección puede cambiar sin ninguna dificultad.

Un puntero a variable ***const*** se declara anteponiendo la palabra ***const***:

```
const char *nombre1 "Ramón"   //no se puede modificar el valor de la
                              //variable
```

Por otra parte, un puntero ***const*** a variable cualquiera se declara interponiendo la palabra ***const*** entre el tipo y el nombre de la variable:

```
char* const nombre2 "Ramón"    // no se puede modificar la dirección a
                               // la que apunta el puntero, pero sí el
                               // valor.
```

4. OPERADORES, EXPRESIONES Y SENTENCIAS

4.1. Operadores

Un ***operador*** *es un carácter o grupo de caracteres que actúa sobre una, dos o más variables para realizar una determinada* ***operación*** *con un determinado* ***resultado***. Ejemplos típicos de operadores son la *suma* (+), la *diferencia* (-), el *producto* (*), etc. Los operadores pueden ser ***unarios***, ***binarios*** y ***ternarios***, según actúen sobre uno, dos o tres operandos, respectivamente. En C++ existen muchos operadores de diversos tipos (éste es uno de los puntos fuertes del lenguaje), que se verán a continuación.

4.1.1. OPERADORES ARITMÉTICOS

Los ***operadores aritméticos*** son los más sencillos de entender y de utilizar. Todos ellos son operadores binarios. En C++ se utilizan los cinco operadores siguientes:

- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Resto: %

Todos estos operadores se pueden aplicar a constantes, variables y expresiones. El resultado es el que se obtiene de aplicar la operación correspondiente entre los dos operandos.

El único operador que requiere una explicación adicional es el operador ***resto*** %. En realidad su nombre completo es ***resto de la división entera***. Este operador se aplica solamente a constantes, variables o expresiones de tipo ***int***. Aclarado esto, su significado es evidente: 23%4 es 3, puesto que el resto de dividir 23 por 4 es 3. Si ***a%b*** es cero, ***a*** es múltiplo de ***b***.

Como se verá más adelante, una ***expresión*** es un conjunto de variables y constantes –y también de otras expresiones más sencillas– relacionadas mediante distintos operadores. Un ejemplo de expresión en la que intervienen operadores aritméticos es el siguiente polinomio de grado 2 en la variable *x*:

```
5.0 + 3.0*x - x*x/2.0
```

Las expresiones pueden contener ***paréntesis*** (...) que agrupan a algunos de sus términos. Puede haber paréntesis contenidos dentro de otros paréntesis. El significado de los paréntesis coincide con el habitual en las expresiones matemáticas, con algunas características importantes que se verán más adelante. En ocasiones, la introducción de espacios en blanco mejora la legibilidad de las expresiones.

4.1.2. OPERADORES DE ASIGNACIÓN

Los ***operadores de asignación*** atribuyen a una variable –es decir, depositan en la zona de memoria correspondiente a dicha variable– el resultado de una expresión o el valor de otra variable (en realidad, una variable es un caso particular de una expresión).

El operador de asignación más utilizado es el ***operador de igualdad*** (=), que no debe ser confundido con la igualdad matemática. Su forma general es:

```
nombre_de_variable = expresion;
```

cuyo funcionamiento es como sigue: se evalúa **expresion** y el resultado se deposita en **nombre_de_variable**, sustituyendo cualquier otro valor que hubiera en esa posición de memoria anteriormente. Una posible utilización de este operador es como sigue:

```
variable = variable + 1;
```

Desde el punto de vista matemático este ejemplo no tiene sentido (¡Equivale a 0 = 1!), pero sí lo tiene considerando que en realidad ***el operador de asignación (=) representa una sustitución***; en efecto, se toma el valor de **variable** contenido en la memoria, se le suma una unidad y el valor resultante vuelve a depositarse en memoria en la zona correspondiente al identificador **variable**, sustituyendo al valor que había anteriormente. El resultado ha sido incrementar el valor de **variable** en una unidad.

Así pues, una variable puede aparecer a la izquierda y a la derecha del operador (=). Sin embargo, a la izquierda de (=) no puede haber una expresión: tiene que ser necesariamente el nombre de una variable. Es incorrecto, por tanto, escribir algo así como:

```
a + b = c;
```

Existen otros diez operadores de asignación (`+=, -=, *=, /=, %=, <<=, >>=, &=, |= y ^=`). Estos operadores simplifican algunas operaciones recurrentes sobre una misma variable. Su forma general es:

```
variable op= expresion;
```

donde ***op*** representa cualquiera de los operadores. La expresión anterior es equivalente a:

```
variable = variable op expresion;
```

A continuación se presentan algunos ejemplos con estos operadores de asignación:

```
distancia += 1;       equivale a:     distancia = distancia + 1;
rango /= 2.0          equivale a:     rango = rango /2.0
x *= 3.0 * y - 1.0    equivale a:     x = x * (3.0 * y - 1.0)
```

4.1.3. OPERADORES INCREMENTALES

Los ***operadores incrementales*** (`++`) y (`--`) son operadores unarios que incrementan o disminuyen ***en una unidad*** el valor de la variable a la que afectan. Estos operadores pueden ir inmediatamente delante o detrás de la variable. Si preceden a la variable, ésta es incrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece. Si es la variable la que precede al operador, la variable es incrementada después de ser utilizada en la expresión. A continuación se presenta un ejemplo de estos operadores:

```
i = 2;
j = 2;
m = i++;      /* despues de ejecutarse esta sentencia m=2 e i=3 */
n = ++j;      /* despues de ejecutarse esta sentencia n=3 y j=3 */
```

Estos operadores son muy utilizados. Es importante entender muy bien por qué los resultados **m** y **n** del ejemplo anterior son diferentes.

4.1.4. OPERADORES RELACIONALES

Este es un apartado especialmente importante para todas aquellas personas sin experiencia en programación. Una característica imprescindible de cualquier lenguaje de programación es la de ***considerar alternativas***, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones. Los ***operadores relacionales*** permiten estudiar si se cumplen o no esas condiciones. Así pues, estos operadores producen un resultado u otro según se cumplan o no algunas condiciones que se verán a continuación.

En el lenguaje natural, existen varias palabras o formas de indicar si se cumple o no una determinada condición. En inglés estas formas son (***yes***, ***no***), (***on***, ***off***), (***true***, ***false***), etc. En Informática se ha hecho bastante general el utilizar la última de las formas citadas: (***true***, ***false***). Si una condición se cumple, el resultado es ***true***; en caso contrario, el resultado es ***false***.

En C++, un 0 representa la condición de ***false***, y cualquier número distinto de 0 equivale a la condición ***true***. Cuando el resultado de una expresión es ***true*** y hay que asignar un valor concreto distinto de cero, por defecto se toma un valor unidad. Los ***operadores relacionales*** de C++ son los siguientes:

- Igual que: `==`
- Menor que: `<`
- Mayor que: `>`
- Menor o igual que: `<=`
- Mayor o igual que: `>=`
- Distinto que: `!=`

Todos los ***operadores relacionales*** son operadores ***binarios*** (tienen dos operandos), y su forma general es la siguiente:

```
expresion1 op expresion2
```

donde ***op*** es uno de los operadores (`==, <, >, <=, >=, !=`). El funcionamiento de estos operadores es el siguiente: se evalúan **expresion1** y **expresion2**, y se comparan los valores resultantes. *Si la condición representada por el operador relacional se cumple, el resultado es 1; si la condición no se cumple, el resultado es 0.*

A continuación se incluyen algunos ejemplos de estos operadores aplicados a constantes:

```
(2==1)   /* resultado=0 porque la condición no se cumple */
(3<=3)   /* resultado=1 porque la condición se cumple */
(3<3)    /* resultado=0 porque la condición no se cumple */
(1!=1)   /* resultado=0 porque la condición no se cumple */
```

4.1.5. OPERADORES LÓGICOS

Los ***operadores lógicos*** son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc. El lenguaje C++ tiene dos operadores lógicos: el operador ***Y*** (&&) y el operador ***O*** (||). En inglés son los operadores ***and*** y ***or***. Su forma general es la siguiente:

```
expresion1 || expresion2
expresion1 && expresion2
```

El operador **&&** devuelve un 1 si tanto **expresion1** como **expresion2** son verdaderas (o iguales a 1), y 0 en caso contrario, es decir si una de las dos expresiones o las dos son falsas (iguales a 0); por otra parte, el operador **||** devuelve 1 si al menos una de las expresiones es cierta. Es importante tener en cuanta que los compiladores de C++ tratan de optimizar la ejecución de estas expresiones, lo cual puede tener a veces efectos no deseados. Por ejemplo: para que el resultado del operador **&&** sea verdadero, ambas expresiones tienen que ser verdaderas; si se evalúa **expresion1** y es falsa, ya no hace falta evaluar **expresion2**, y de hecho no se evalúa. Algo parecido pasa con el operador **||**: si **expresion1** es verdadera, ya no hace falta evaluar **expresion2**.

Los operadores **&&** y **||** se pueden combinar entre sí, agrupados entre paréntesis, dando a veces un código de más difícil interpretación. Por ejemplo:

```
(2==1) || (-1==-1)                          /* el resultado es 1 */
(2==2) && (3==-1)                           /* el resultado es 0 */
((2==2) && (3==3)) || (4==0)                /* el resultado es 1 */
((6==6) || (8==0)) && ((5==5) && (3==2))    /* el resultado es 0 */
```

4.1.6. OTROS OPERADORES

Además de los operadores vistos hasta ahora, el lenguaje C++ dispone de otros operadores. En esta sección se describen algunos ***operadores unarios*** adicionales.

– Operador ***menos*** (–).

  El efecto de este operador en una expresión es cambiar el signo de la variable o expresión que le sigue. Recuérdese que en C++ no hay constantes numéricas negativas. La forma general de este operador es:

```
- expresion
```

– Operador ***más*** (+).

  Este es un operador unario de C++, y que tiene como finalidad la de servir de complemento al operador (–) visto anteriormente. Se puede anteponer a una variable o expresión como operador unario, pero en realidad no hace nada.

– Operador ***sizeof***().

  Este operador de C++ puede parecer una función, pero en realidad es un operador. La finalidad del operador **sizeof()** es devolver el tamaño, en ***bytes***, del tipo de variable introducida entre los paréntesis.

```
var_1 = sizeof(double)          /* var_1 contiene el tamaño
                                   de una variable double   */
```

– Operador ***negación lógica*** (!).

  Este operador devuelve un cero (*false*) si se aplica a un valor distinto de cero (*true*), y devuelve un 1 (*true*) si se aplica a un valor cero (*false*). Su forma general es:

```
!expresion
```

– Operador ***coma*** (,).

  Los operandos de este operador son expresiones, y tiene la forma general:

```
expresion = expresion_1, expresion_2
```

En este caso, **expresion_1** se evalúa primero, y luego se evalúa **expresion_2**. El resultado global es el valor de la segunda expresión, es decir de **expresion_2**. Este es el operador de menos precedencia de todos los operadores de C++.

– Operadores ***dirección*** (&) e ***indirección*** (*).

Aunque estos operadores se introduzcan aquí de modo circunstancial, su importancia en el lenguaje C++ es absolutamente esencial, resultando uno de los puntos más fuertes –y quizás más difíciles de dominar– de este lenguaje. La forma general de estos operadores es la siguiente:

```
*expresion;
&variable;
```

El ***operador dirección* &** devuelve la dirección de memoria de la variable que le sigue. Por ejemplo:

```
variable_1 = &variable_2;
```

Después de ejecutarse esta instrucción **variable_1** contiene la dirección de memoria donde se guarda el contenido de **variable_2**. Las variables que almacenan direcciones de otras variables se denominan ***punteros*** (o apuntadores), deben ser declaradas como tales, y tienen su propia aritmética y modo de funcionar. Se verán con detalle un poco más adelante.

No se puede modificar la dirección de una variable, por lo que no están permitidas operaciones en las que el operador **&** figura a la izda del operador (=), al estilo de:

```
&variable_1 = nueva_direccion;          //error
```

El ***operador indirección* *** es el operador complementario del **&**. Aplicado a una expresión que represente una dirección de memoria (***puntero***) permite hallar el contenido o valor almacenado en esa dirección. Por ejemplo:

```
variable_3 = *variable_1;
```

El contenido de la dirección de memoria representada por la ***variable puntero* variable_1** se recupera y se asigna a la variable **variable_3**.

Como ya se ha indicado, las ***variables puntero*** y los operadores ***dirección*** (&) e ***indirección*** (*) serán explicados con mucho más detalle en una sección posterior.

– Operadores ***new*** y ***delete***.

Hasta ahora sólo se han visto dos posibles tipos de duración de las variables: ***static***, las cuales existen durante toda la ejecución del programa, y ***automatic***, que existen desde que son declaradas hasta que finaliza el bloque donde han sido declaradas.

Con los operadores ***new*** y ***delete*** el programador tiene entera libertad para decidir crear o destruir sus variables cuando las necesite. Una variable creada con el operador ***new*** dentro de cualquier bloque, perdura hasta que es explícitamente borrada con el operador ***delete***. Puede traspasar la frontera de su bloque y ser manipulada por instrucciones de otros bloques.

Estos operadores serán descritos más ampliamente en el apartado dedicado a la reserva dinámica de memoria.

– Operador ***de resolución de visibilidad*** (::).

Este operador permite acceder a una variable global cuando ésta se encuentra oculta por otra variable local del mismo nombre. Considérese el siguiente ejemplo:

```
int a=2;
void main(void){
     int a =10;
     cout << a << endl;          //muestra en pantalla 10
     cout << ::a << endl;        //muestra en pantalla 2
}
```

– Operador ***opcional*** (?:).

Este operador permite controlar el flujo de ejecución del programa. Se ampliarán detalles en el apartado de las bifurcaciones.

4.2. Reglas de precedencia y asociatividad

El resultado de una expresión depende del orden en que se ejecutan las operaciones. El siguiente ejemplo ilustra claramente la importancia del orden. Considérese la expresión:

```
3 + 4 * 2
```

Si se realiza primero la suma (3+4) y después el producto (7*2), el resultado es 14; si se realiza primero el producto (4*2) y luego la suma (3+8), el resultado es 11. Con objeto de que el resultado de cada expresión quede claro e inequívoco, es necesario definir las reglas que definen el orden con el que se ejecutan las expresiones de C++. Existe dos tipos de reglas para determinar este orden de evaluación: las reglas de ***precedencia*** y de ***asociatividad***. Además, el orden de evaluación puede modificarse por medio de paréntesis, pues *siempre se realizan primero las operaciones encerradas en los paréntesis más interiores*. Los distintos operadores de C++ se ordenan según su distinta precedencia o prioridad; para operadores de la misma precedencia o prioridad, en algunos el orden de ejecución es de izquierda a derecha, y otros de derecha a izquierda (se dice que *se asocian* de izda a dcha, o de dcha a izda). A este orden se le llama ***asociatividad***.

En la Tabla 4.1 se muestra la precedencia –disminuyendo de arriba a abajo– y la asociatividad de los operadores de C++. En dicha Tabla se incluyen también algunos operadores que no han sido vistos hasta ahora.

Tabla 4.1. Precedencia y asociatividad de los operadores de C++.

Precedencia	Asociatividad
`()  []  ->  .`	izda a dcha
`++  --  !  sizeof  (tipo)` `+(unario)  -(unario)  *(indir.)  &(dirección)`	dcha a izda
`*  /  %`	izda a dcha
`+  -`	izda a dcha
`<  <=  >  >=`	izda a dcha
`==  !=`	izda a dcha
`&&`	izda a dcha
`||`	izda a dcha
`?:`	dcha a izda
`=  +=  -=  *=  /=`	dcha a izda

`, (operador coma)`	`izda a dcha`

4.3. Expresiones

Ya han aparecido algunos ejemplos de expresiones del lenguaje C++ en las secciones precedentes. Una expresión es una combinación de variables y/o constantes, y operadores. La expresión es equivalente al resultado que proporciona al aplicar sus operadores a sus operandos. Por ejemplo, 1+5 es una expresión formada por dos valores (1 y 5) y un operador (el +); esta expresión es equivalente al valor 6, lo cual quiere decir que allí donde esta expresión aparece en el programa, en el momento de la ejecución es evaluada y sustituida por su resultado. Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos. En C++ existen distintos tipos de expresiones.

4.3.1. EXPRESIONES ARITMÉTICAS

Están formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales (`+, -, *, /, %, ++, --`). Como se ha dicho, también se pueden emplear paréntesis de tantos niveles como se desee, y su interpretación sigue las normas aritméticas convencionales. Por ejemplo, la solución de la ecuación de segundo grado:

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

se escribe, en C++ en la forma:

```
x=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. El conjunto de la variable que está a la izquierda del signo (=), el operador de asignación, la expresión aritmética y el carácter (;) constituyen una ***sentencia***. En la expresión anterior aparece la llamada a la ***función de librería sqrt()***, que tiene como ***valor de retorno*** la raíz cuadrada de su único ***argumento***. En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

```
x = (-b + sqrt((b * b) - (4 * a * c)))/(2 * a);
```

4.3.2. EXPRESIONES LÓGICAS

Los elementos con los que se forman estas expresiones son ***valores lógicos***; *verdaderos* (***true***, o distintos de 0) y *falsos* (***false***, o iguales a 0), y los ***operadores lógicos*** **`||`**, **`&&`** y **`!`**. También se pueden emplear los ***operadores relacionales*** (`<, >, <=, >=, ==, !=`) para producir estos valores lógicos a partir de valores numéricos. Estas expresiones equivalen siempre a un valor 1 (***true***) o a un valor 0 (***false***). Por ejemplo:

```
a = ((b>c)&&(c>d))||((c==e)||(e==b));
```

donde de nuevo la ***expresión lógica*** es lo que está entre el operador de asignación (=) y el (;). La variable **a** valdrá 1 si **b** es mayor que **c** *y* **c** mayor que **d**, ó si **c** es igual a **e** ó **e** es igual a **b**.

4.3.3. EXPRESIONES GENERALES

Una de las características más importantes (y en ocasiones más difíciles de manejar) del C++ es su flexibilidad para combinar expresiones y operadores de distintos tipos en una expresión que se podría llamar *general*, aunque es una expresión absolutamente ordinaria de C++.

Recuérdese que el resultado de una expresión lógica es siempre un valor numérico (un 1 ó un 0); esto permite que cualquier expresión lógica pueda aparecer como sub-expresión en una expresión aritmética. Recíprocamente, cualquier valor numérico puede ser considerado como un valor lógico: ***true*** si es distinto de 0 y ***false*** si es igual a 0. Esto permite introducir cualquier expresión aritmética como sub-expresión de una expresión lógica. Por ejemplo:

```
(a - b*2.0) && (c != d)
```

A su vez, *el operador de asignación* (=), además de introducir un nuevo valor en la variable que figura a su izda, *deja también este valor disponible para ser utilizado* en una expresión más general. Por ejemplo, supóngase el siguiente código que inicializa a 1 las tres variables **a**, **b** y **c**:

```
a = b = c = 1;
```

que equivale a:

```
a = (b = (c = 1));
```

En realidad, lo que se ha hecho ha sido lo siguiente. En primer lugar se ha asignado un valor unidad a **c**; el resultado de esta asignación es también un valor unidad, que está disponible para ser asignado a **b**; a su vez el resultado de esta segunda asignación vuelve a quedar disponible y se puede asignar a la variable **a**.

4.4. Sentencias

Las ***expresiones*** de C++ son unidades o componentes elementales de unas entidades de rango superior que son las ***sentencias***. Las sentencias son unidades completas, ejecutables en sí mismas. Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

4.4.1. SENTENCIAS SIMPLES

Una sentencia simple es una expresión de algún tipo terminada con un carácter (;). Un caso típico son las declaraciones o las sentencias aritméticas. Por ejemplo:

```
float real;
espacio = espacio_inicial + velocidad * tiempo;
```

4.4.2. SENTENCIA VACÍA Ó NULA

En algunas ocasiones es necesario introducir en el programa una sentencia *que ocupe un lugar, pero que no realice ninguna tarea*. A esta sentencia se le denomina ***sentencia vacía*** y consta de un simple carácter (;). Por ejemplo:

```
;
```

4.4.3. SENTENCIAS COMPUESTAS O BLOQUES

Muchas veces es necesario poner varias sentencias en un lugar del programa donde debería haber una sola. Esto se realiza por medio de ***sentencias compuestas***. Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves **`{...}`**. También se conocen con el nombre de ***bloques***. Una sentencia compuesta puede incluir otras sentencias, simples y compuestas.

Un ejemplo de sentencia compuesta es el siguiente:

```
{
     int i = 1, j = 3, k;
     double masa;

     masa = 3.0;
     k = y + j;
}
```

Las sentencias compuestas se utilizarán con mucha frecuencia en el Capítulo 5, al introducir las sentencias que permiten modificar el flujo de control del programa.

5. CONTROL DEL FLUJO DE EJECUCIÓN

En principio, las sentencias de un programa en C++ se ejecutan *secuencialmente*, esto es, cada una a continuación de la anterior empezando por la primera y acabando por la última. El lenguaje C++ dispone de varias sentencias para modificar este flujo secuencial de la ejecución. Las más utilizadas se agrupan en dos familias: las ***bifurcaciones***, que permiten elegir entre dos o más opciones según ciertas condiciones, y los ***bucles***, que permiten ejecutar repetidamente un conjunto de instrucciones tantas veces como se desee, cambiando o actualizando ciertos valores.

5.1. Bifurcaciones

5.1.1. OPERADOR CONDICIONAL

El operador condicional es un operador con tres operandos (ternario) que tiene la siguiente forma general:

```
expresion_1 ? expresion_2 : expresion_3;
```

Explicación: Se evalúa **expresion_1**. Si el resultado de dicha evaluación es ***true*** (!=0), se ejecuta **expresion_2**; si el resultado es ***false*** (=0), se ejecuta **expresion_3**.

5.1.2. SENTENCIA IF

Esta sentencia de control permite ejecutar o no una sentencia según se cumpla o no una determinada condición. Esta sentencia tiene la siguiente forma general:

```
if (expresion)
     sentencia
```

Explicación: Se evalúa **expresion**. Si el resultado es ***true*** (!=0), se ejecuta **sentencia**; si el resultado es ***false*** (=0), se salta **sentencia** y se prosigue en la línea siguiente. Hay que recordar que **sentencia** puede ser una sentencia simple o compuesta.

5.1.3. SENTENCIA IF ... ELSE

Esta sentencia permite realizar una *bifurcación*, ejecutando una parte u otra del programa según se cumpla o no una cierta condición. La forma general es la siguiente:

```
if (expresion)
     sentencia_1
else
     sentencia_2
```

Explicación: Se evalúa **expresion**. Si el resultado es ***true*** (!=0), se ejecuta **sentencia_1** y se prosigue en la línea siguiente a **sentencia_2**; si el resultado es ***false*** (=0), se salta **sentencia_1**, se ejecuta **sentencia_2** y se prosigue en la línea siguiente. Hay que indicar aquí también que **sentencia_1** y **sentencia_2** pueden ser sentencias simples o compuestas.

5.1.4. SENTENCIA IF ... ELSE MÚLTIPLE

Esta sentencia permite realizar una ramificación múltiple, ejecutando *una* entre varias partes del programa según se cumpla *una* entre *n* condiciones.

La forma general es la siguiente:

```
if (expresion_1)
     sentencia_1
else if (expresion_2)
     sentencia_2
else if (expresion_3)
     sentencia_3
else if (...)
     ...
[else
     sentencia_n]
```

Explicación: Se evalúa **expresion_1**. Si el resultado es ***true***, se ejecuta **sentencia_1**. Si el resultado es ***false***, se salta **sentencia_1** y se evalúa **expresion_2**. Si el resultado es ***true*** se ejecuta **sentencia_2**, mientras que si es ***false*** se evalúa **expresion_3** y así sucesivamente. Si ninguna de las expresiones o condiciones es ***true*** se ejecuta **expresion_n** que es la opción por defecto (puede ser la sentencia vacía, y en ese caso puede eliminarse junto con la palabra **else**). Todas las sentencias pueden ser simples o compuestas.

5.1.5. SENTENCIA SWITCH

La sentencia que se va a describir a continuación desarrolla una función similar a la de la sentencia ***if ... else*** con múltiples ramificaciones, aunque como se puede ver presenta también importantes diferencias. La forma general de la sentencia ***switch*** es la siguiente:

```
switch (expresion) {
     case expresion_cte_1:
          sentencia_1
     case expresion_cte_2:
          sentencia_2
     ...
     case expresion_cte_n:
          sentencia_n
     [default:
          sentencia]
}
```

Explicación: Se evalúa **expresion** y se considera el resultado de dicha evaluación. Si dicho resultado coincide con el valor constante **expresion_cte_1**, se ejecuta **sentencia_1** seguida de **sentencia_2**, **sentencia_3**, ..., **sentencia**. Si el resultado coincide con el valor constante **expresion_cte_2**, se ejecuta **sentencia_2** seguida de **sentencia_3**, ..., **sentencia**. En general, se ejecutan todas aquellas sentencias que están a continuación de la **expresion_cte** cuyo valor coincide con el resultado calculado al principio. Si ninguna **expresion_cte** coincide se ejecuta la **sentencia** que está a continuación de **default**. Si se desea ejecutar únicamente una **sentencia_i** (y no todo un conjunto de ellas), basta poner una sentencia **break** a continuación (en algunos casos puede utilizarse la sentencia **return** o la función **exit()**). El efecto de la sentencia **break** es dar por terminada la ejecución de la sentencia **switch**. Existe también la posibilidad de ejecutar la misma **sentencia_i** para varios valores del resultado de **expresion**, poniendo varios **case expresion_cte** seguidos.

El siguiente ejemplo ilustra las posibilidades citadas:

```
int opc;
cout <<"Introduzca la opcion deseada: ";
cin>>opc;
switch (opc){
     case 1 : cout << "caso 1\n" ; break;
     case 2 : cout << "caso 2\n" ; break;
     case 3 : cout << "caso 3\n" ; break;
     case 4 : cout << "caso 4\n" ; break;
     default : cout << "caso por defecto";
}
```

5.1.6. SENTENCIAS IF ANIDADAS

Una sentencia ***if*** puede incluir otros ***if*** dentro de la parte correspondiente a su **sentencia**, A estas sentencias se les llama ***sentencias anidadas*** (una dentro de otra), por ejemplo,

```
if (a >= b)
     if (b != 0.0)
          c = a/b;
```

En ocasiones pueden aparecer dificultades de interpretación con sentencias ***if...else*** anidadas, como en el caso siguiente:

```
if (a >= b)
     if (b != 0.0)
          c = a/b;
     else
          c = 0.0;
```

En principio se podría plantear la duda de a cuál de los dos ***if*** corresponde la parte ***else*** del programa. Los espacios en blanco –las *indentaciones* de las líneas– parecen indicar que la sentencia que sigue a ***else*** corresponde al segundo de los ***if***, y así es en realidad, pues la regla es que el ***else*** pertenece al ***if*** más cercano. Sin embargo, no se olvide que el compilador de C++ no considera los espacios en blanco (aunque sea muy conveniente introducirlos para hacer más claro y legible el programa), y que si se quisiera que el ***else*** perteneciera al primero de los ***if*** no bastaría cambiar los espacios en blanco, sino que habría que utilizar *llaves*, en la forma:

```
if (a >= b) {
     if (b != 0.0)
          c = a/b;
     }
else
     c = 0.0;
```

Recuérdese que todas las sentencias ***if*** e ***if...else***, equivalen a una única sentencia por la posición que ocupan en el programa.

5.2. Bucles

Además de ***bifurcaciones***, en el lenguaje C++ existen también varias sentencias que permiten repetir una serie de veces la ejecución de unas líneas de código. Esta repetición se realiza, bien un

número determinado de veces, bien hasta que se cumpla una determinada condición de tipo lógico o aritmético. De modo genérico, a estas sentencias se les denomina ***bucles***. Las tres construcciones del lenguaje C++ para realizar bucles son el ***while***, el ***for*** y el ***do...while***.

5.2.1. SENTENCIA WHILE

Esta sentencia permite ejecutar repetidamente, *mientras se cumpla una determinada condición*, una sentencia o bloque de sentencias. La forma general es como sigue:

```
while (expresion_de_control)
     sentencia
```

Explicación: Se evalúa **expresion_de_control** y si el resultado es ***false*** se salta **sentencia** y se prosigue la ejecución. Si el resultado es ***true*** se ejecuta **sentencia** y se vuelve a evaluar **expresion_de_control** (evidentemente alguna variable de las que intervienen en **expresion_de_control** habrá tenido que ser modificada, pues si no el *bucle* continuaría indefinidamente). La ejecución de **sentencia** prosigue hasta que **expresion_de_control** se hace ***false***, en cuyo caso la ejecución continúa en la línea siguiente. En otras palabras, **sentencia** se ejecuta repetidamente mientras **expresion_de_control** sea ***true***, y se deja de ejecutar cuando **expresion_de_control** se hace ***false***. Obsérvese que en este caso el *control* para decidir si se sale o no del *bucle* está antes de **sentencia**, por lo que es posible que **sentencia** no se llegue a ejecutar ni una sola vez.

5.2.2. SENTENCIA FOR

For es quizás el tipo de bucle mas versátil y utilizado del lenguaje C++. Su forma general es la siguiente:

```
for (inicializacion; expresion_de_control; actualizacion)
     sentencia;
```

Explicación: Posiblemente la forma más sencilla de explicar la sentencia ***for*** sea utilizando la construcción ***while*** que sería equivalente. Dicha construcción es la siguiente:

```
inicializacion;
while (expresion_de_control) {
     sentencia;
     actualizacion;
}
```

donde **sentencia** puede ser una única sentencia terminada con (;), otra sentencia de control ocupando varias líneas (***if***, ***while***, ***for***, ...), o una sentencia compuesta. Antes de iniciarse el bucle se ejecuta *inicializacion*, que es una o más sentencias que asignan valores iniciales a ciertas variables o contadores. A continuación se evalúa **expresion_de_control** y si es ***false*** se prosigue en la sentencia siguiente a la construcción ***for***; si es ***true*** se ejecutan **sentencia** y *actualizacion*, y se vuelve a evaluar **expresion_de_control**. El proceso prosigue hasta que **expresion_de_control** sea ***false***. La parte de *actualizacion* sirve para actualizar variables o incrementar contadores. Un ejemplo típico puede ser el producto escalar de dos vectores ***a*** y ***b*** de dimensión ***n***:

```
for (pe =0.0, i=0; i<n; i++){        /*i empieza en 0, porque los
                                     vectores empiezan a contar desde
                                     la posición cero*/
     pe += a[i]*b[i];
}
```

Primeramente se inicializa la variable **pe** a cero y la variable **i** a 0; el ciclo se repetirá mientras que i sea menor que **n**, y al final de cada ciclo el valor de **i** se incrementará en una unidad. En total, el bucle se repetirá **n** veces. La ventaja de la construcción ***for*** sobre la construcción ***while*** equivalente está en que en la cabecera de la construcción ***for*** se tiene toda la información sobre como se inicializan, controlan y actualizan las variables del bucle. Obsérvese que la *inicializacion* consta de dos sentencias separadas por el operador (,).

5.2.3. SENTENCIA DO ... WHILE

Esta sentencia funciona de modo análogo a ***while***, con la diferencia de que la evaluación de **expresion_de_control** se realiza al final del bucle, después de haber ejecutado al menos una vez las sentencias entre llaves; éstas se vuelven a ejecutar mientras **expresion_de_control** sea ***true***. La forma general de esta sentencia es:

```
do
     sentencia;
while(expresion_de_control);
```

donde **sentencia** puede ser una única sentencia o un bloque, y en la que debe observarse que *hay que poner (;) a continuación del paréntesis* que encierra a **expresion_de_control**, entre otros motivos para que esa línea se distinga de una sentencia ***while*** ordinaria.

5.3. Sentencias *break, continue, goto*

La instrucción ***break*** interrumpe la ejecución del bucle donde se ha incluido, haciendo al programa salir de él aunque la **expresion_de_control** correspondiente a ese bucle sea verdadera.

La sentencia ***continue*** hace que el programa comience el siguiente ciclo del bucle donde se halla, aunque no haya llegado al final de las sentencia compuesta o bloque.

La sentencia ***goto*** *etiqueta* hace saltar al programa a la sentencia donde se haya escrito la *etiqueta* correspondiente. Por ejemplo:

```
sentencias ...
...
if (condicion)
     goto otro_lugar;
sentencia_1;
sentencia_2;
...
otro_lugar:
sentencia_3;
...
```

Obsérvese que la *etiqueta* termina con el carácter (:). La sentencia ***goto*** no es una sentencia muy prestigiada en el mundo de los programadores de C++, pues disminuye la claridad y legibilidad del código. Fue introducida en el lenguaje por motivos de compatibilidad con antiguos hábitos de programación, y siempre puede ser sustituida por otras construcciones más claras y estructuradas.

6. TIPOS DE DATOS DERIVADOS

Además de los tipos de datos fundamentales vistos en la Sección 2, en C++ existen algunos otros tipos de datos muy utilizados y que se pueden considerar derivados de los anteriores. En esta sección se van a presentar los ***punteros***, las ***matrices*** y las ***estructuras***.

6.1. Punteros

6.1.1. CONCEPTO DE PUNTERO O APUNTADOR

El valor de cada variable está almacenado en un lugar determinado de la memoria, caracterizado por una ***dirección*** (que se suele expresar en hexadecimal). El ordenador mantiene una ***tabla de direcciones*** (ver Tabla 6.1) que relaciona el nombre de cada variable con su dirección en la memoria. Gracias a los nombres de las variables (identificadores), de ordinario no hace falta que el programador se preocupe de la dirección de memoria donde están almacenados sus datos. Sin embargo, en ciertas ocasiones es más útil trabajar con las direcciones que con los propios nombres de las variables. El lenguaje C++ dispone del ***operador dirección*** (&) que permite determinar la dirección de una variable, y de un tipo especial de variables destinadas a contener direcciones de variables. Estas variables se llaman ***punteros*** o ***apuntadores*** (en inglés *pointers*).

Así pues, un ***puntero*** es una variable que puede contener la ***dirección*** de otra variable. Por tanto, los *punteros* están almacenados en algún lugar de la memoria y tienen su propia dirección (más adelante se verá que existen *punteros a punteros*). Se dice que un *puntero apunta a una variable* si su contenido es la dirección de esa variable. Un *puntero* ocupa de ordinario 4 bytes de memoria, y *se debe declarar o definir de acuerdo con el tipo del dato* al que apunta. Por ejemplo, un *puntero* a una variable de tipo ***int*** se *declara* del siguiente modo:

```
int *direc;
```

lo que quiere decir que a partir de este momento, la variable ***direc*** podrá contener la dirección de cualquier variable entera. La regla nemotécnica es que el valor al que apunta ***direc*** (es decir ****direc***, como luego se verá), es de tipo *int*. Los *punteros* a *long*, *char*, *float* y *double* se definen análogamente a los *punteros* a *int*.

6.1.2. OPERADORES DIRECCIÓN (&) E INDIRECCIÓN (*)

Como se ha dicho, el lenguaje C++ dispone del ***operador dirección*** (&) que permite hallar la dirección de la variable a la que se aplica. Un *puntero* es una verdadera variable, y por tanto puede cambiar la variable a la que apunta. Para acceder al valor depositado en la zona de memoria a la que apunta un *puntero* se debe utilizar el ***operador indirección*** (*). Por ejemplo, supóngase las siguientes declaraciones y sentencias,

```
int i, j, *p;      /* p es un puntero a int */
p = &i;            /* p contiene la dirección de i */
*p = 10;           /* i toma el valor 10 */
p = &j;            /* p contiene ahora la dirección de j */
*p = -2;           /* j toma el valor -2 */
```

Las constantes y las expresiones no tienen dirección, por lo que no se les puede aplicar el operador (&). Tampoco puede cambiarse la dirección de una variable. Los valores posibles para un puntero son las direcciones posibles de memoria. Un puntero puede tener valor 0 (equivalente a la

constante simbólica predefinida NULL). No se puede asignar una dirección absoluta directamente (habría que hacer un *casting*). Las siguientes sentencias son ilegales:

```
p = &34;           /* las constantes no tienen dirección */
p = &(i+1);        /* las expresiones no tienen dirección */
&i = p;            /* las direcciones de las variables no se pueden cambiar */
p = 17654;         /* habría que escribir p = (int *)17654; */
```

No se permiten asignaciones directas (sin *casting*) entre punteros que apuntan a distintos tipos de variables. Sin embargo, existe un tipo indefinido de punteros (***void ***, o ***punteros a void***), que puede asignarse y al que puede asignarse cualquier tipo de puntero. Sin embargo, no se les puede asignar ningún valor. Por ejemplo:

```
int *p;
double *q;
void *r;
p = q;             /* ilegal */
p = (int *)q;      /* legal */
r = q;             /* legal */
p = r;             /* ilegal */
*r=3;              /* ilegal */
```

6.1.3. ARITMÉTICA DE PUNTEROS

Como ya se ha visto, los *punteros* son unas variables un poco especiales, ya que guardan información –no sólo de la dirección a la que apuntan–, sino también del *tipo* de variable almacenado en esa dirección. Esto implica que no van a estar permitidas las operaciones que no tienen sentido con direcciones, como multiplicar o dividir, pero sí otras como sumar o restar. Además estas operaciones se realizan de un modo correcto, pero que no es el ordinario. Así, la sentencia:

```
p = p+1;
```

hace que **p** apunte a la dirección siguiente de la que apuntaba, teniendo en cuenta el tipo de dato. Por ejemplo, si el valor apuntado por **p** es *short int* y ocupa 2 bytes, el sumar 1 a **p** implica desplazar 2 bytes la dirección que contiene, mientras que si **p** apunta a un *double*, sumarle 1 implica desplazarlo 8 bytes.

También tiene sentido la *diferencia de punteros* al mismo *tipo* de variable. El resultado es la *distancia* entre las direcciones de las variables apuntadas por ellos, no en *bytes* sino en *datos* de ese mismo tipo. Las siguientes expresiones tienen pleno sentido en C++:

```
p = p + 1;
p = p + i;
p += 1;
p++;
```

Tabla 6.1. Tabla de direcciones.

Variable	Dirección de memoria
A	00FA:0000
B	00FA:0002
C	00FA:0004
p1	00FA:0006
p2	00FA:000A

P	00FA:000E

El siguiente ejemplo ilustra la aritmética de punteros:

```
void main(void) {
     int   a, b, c;
     int   *p1, *p2;
     int  **p;     /* p es un puntero que apuntará a otro puntero */

     p1 = &a;    /* Paso 1. La dirección de a es asignada a p1 */
     *p1 = 1;    /* Paso 2. p1 (a) es igual a 1. Equivale a a = 1; */
     p2 = &b;    /* Paso 3. La dirección de b es asignada a p2*/
     *p2 = 2;    /* Paso 4. p2 (b) es igual a 2. Equivale a b = 2; */
     p1 = p2;    /* Paso 5. El valor del p1 = p2 */
     *p1 = 0;    /* Paso 6. b = 0 */
     p2 = &c;    /* Paso 7. La dirección de c es asignada a p2 */
     *p2 = 3;    /* Paso 8. c = 3 */
     cout << a << b << c << endl;    /* Paso 9. Se imprime 103 */

     p = &p1;    /* Paso 10. p contiene la dirección de p1 */
     *p = p2;    /* Paso 11. p1= p2; */
     *p1 = 1;    /* Paso 12. c = 1 */
     cout << a << b << c << endl;    /* Paso 13. Se imprime 101 */
}
```

Supóngase que en el momento de comenzar la ejecución, las direcciones de memoria de las distintas variables son las mostradas en la Tabla 6.1.

La dirección de memoria está en hexadecimal, con el *segmento* y el *offset* separados por dos puntos (:); basta prestar atención al segundo de estos números, esto es, al *offset*.

La Tabla 6.2 muestra los valores de las variables en la ejecución del programa paso a paso. Se muestran en negrita y cursiva los cambios entre paso y paso. Es importante analizar y entender los cambios de valor.

Tabla 6.2. Ejecución paso a paso de un ejemplo con punteros.

Paso	**a** 00FA:0000	**b** 00FA:0002	**c** 00FA:0004	**p1** 00FA:0006	**p2** 00FA:000A	**p** 00FA:000E
1				***00FA:0000***		
2	***1***			00FA:0000		
3	1			00FA:0000	***000FA:0002***	
4	1	***2***		00FA:0000	000FA:0002	
5	1	2		***000FA:0002***	000FA:0002	
6	1	***0***		000FA:0002	000FA:0002	
7	1	0		000FA:0002	***000FA:0004***	
8	1	0	***3***	000FA:0002	000FA:0004	
9	1	0	3	000FA:0002	000FA:0004	
10	1	0	3	000FA:0002	000FA:0004	***000FA:0006***
11	1	0	3	***000FA:0004***	000FA:0004	000FA:0006
12	1	0	***1***	000FA:0004	000FA:0004	000FA:0006

13	1	0	1	000FA:0004	000FA:0004	000FA:0006

6.2. Vectores, matrices y cadenas de caracteres

Un ***array*** (también conocido como *arreglo*, *vector* o *matriz*) *es un modo de manejar una gran cantidad de datos del mismo tipo bajo un mismo nombre o identificador*. Por ejemplo, mediante la sentencia:

```
double a[10];
```

se reserva espacio para 10 variables de tipo ***double***. Las 10 variables se llaman **a** y se accede a una u otra por medio de un ***subíndice***, que es una *expresión entera* escrita a continuación del nombre entre corchetes [...]. La forma general de la declaración de un vector es la siguiente:

```
tipo nombre[numero_elementos];
```

Los elementos se numeran desde 0 hasta (numero_elementos-1). El tamaño de un vector puede definirse con cualquier expresión constante entera. Para definir tamaños son particularmente útiles las *constantes simbólicas*. En C++ no se puede operar con todo un vector o toda una matriz como una única entidad, sino que hay que tratar sus elementos uno a uno. Los vectores (mejor dicho, los elementos de un vector) se utilizan en las expresiones de C++ como cualquier otra variable. Ejemplos de uso de vectores son los siguientes:

```
a[5] = 0.8;
a[9] = 30. * a[5];
a[0] = 3. * a[9] - a[5]/a[9];
a[3] = (a[0] + a[9])/a[3];
```

Una ***cadena de caracteres*** no es sino un vector de tipo ***char***, con alguna particularidad que conviene resaltar. Las cadenas suelen contener texto (nombres, frases, etc.), y éste se almacena en la parte inicial de la cadena (a partir de la posición cero del vector). Para separar la parte que contiene texto de la parte no utilizada, se utiliza un *carácter fin de texto* que es el carácter nulo ('\0') según el código ASCII. Este carácter se introduce automáticamente al leer o inicializar las cadenas de caracteres, como en el siguiente ejemplo:

```
char ciudad[20] = "San Sebastián";
```

donde a los 13 caracteres del nombre de esta ciudad se añade un decimocuarto: el '\0'. El resto del espacio reservado –hasta la posición **ciudad[19]**– no se utiliza. De modo análogo, una cadena constante tal como "**mar**" ocupa 4 bytes (para las 3 letras y el '\0').

Las ***matrices*** se declaran de forma análoga, con corchetes independientes para cada subíndice. La forma general de la declaración es:

```
tipo nombre[numero_filas][numero_columnas];
```

donde tanto las *filas* como las *columnas* se numeran también a partir de 0. La forma de acceder a los elementos de la matriz es utilizando su nombre, seguido de las expresiones enteras correspondientes a los dos subíndices, entre corchetes.

En C++ tanto los vectores como las matrices admiten los *tipos* de las variables escalares (***char***, ***int***, ***long***, ***float***, ***double***, etc.), y los *modos de almacenamiento* ***auto***, ***extern*** y ***static***, con las mismas características que las variables normales (escalares). No se admite el modo *register*. Los arrays *static* y *extern* se inicializan a cero por defecto. Los arrays *auto* no se inicializan, contienen basura informática.

Las matrices en C++ *se almacenan por filas*, en posiciones consecutivas de memoria. En cierta forma, una matriz se puede ver como un *vector de vectores-fila*. Si una matriz tiene N filas (numeradas de 0 a N-1) y M columnas (numeradas de 0 a la M-1), el elemento (i, j) ocupa el lugar:

posición_elemento(0, 0) + i * M + j

A esta fórmula se le llama *fórmula de direccionamiento* de la matriz.

En C++ pueden definirse *arrays* con tantos subíndices como se desee. Por ejemplo, la sentencia,

```
double a[3][5][7];
```

declara una ***hipermatriz*** con tres subíndices, que podría verse como un conjunto de 3 matrices de dimensión (5x7). En la fórmula de direccionamiento correspondiente, el último subíndice es el que varía más rápidamente.

Como se verá más adelante, los *arrays* presentan una especial relación con los *punteros*. Puesto que los elementos de un vector y una matriz están almacenados consecutivamente en la memoria, la *aritmética de punteros* descrita previamente presenta muchas ventajas. Por ejemplo, supóngase el código siguiente:

```
int  vect[10], mat[3][5], *p;
p = &vect[0];
cout << *(p + 2) << endl;        /* se imprimirá vect[2]*/
p = &mat[0][0];
cout << *(p + 2) << endl;        /* se imprimirá mat[0][2]*/
cout << *(p + 4) << endl;        /* se imprimirá mat[0][4]*/
cout << *(p + 5) << endl;        /* se imprimirá mat[1][0]*/
cout << *(p + 12) << endl;       /* se imprimirá mat[2][2]*/
```

6.2.1. RELACION ENTRE VECTORES Y PUNTEROS

Existe una relación muy estrecha entre los vectores y los punteros. De hecho, el *nombre de un vector es un puntero* (constante, en el sentido de que no puede apuntar a otra variable distinta de aquélla a la que apunta) a la dirección de memoria que contiene el primer elemento del vector. Supónganse las siguientes declaraciones y sentencias:

```
double vect[10];  /* vect es un puntero a vect[0] */
double *p;
...
p = &vect[0];     /* p = vect; */
...
```

El identificador **vect**, es decir *el nombre del vector*, es un *puntero* al primer elemento del vector **vect[ ]**. Esto es lo mismo que decir que el valor de **vect** es **&vect[0]**. Existen más puntos de coincidencia entre los vectores y los punteros:

– Puesto que el nombre de un vector es un *puntero*, obedecerá las leyes de la aritmética de punteros. Por tanto, si **vect** apunta a **vect[0]**, (**vect+1**) apuntará a **vect[1]**, y (**vect+i**) apuntará a **vect[i]**.

– Recíprocamente (y esto resulta quizás más sorprendente), a los *punteros* se les pueden poner *subíndices*, igual que a los vectores. Así pues, si **p** apunta a **vect[0]** se puede escribir:

```
p[3]=p[2]*2.0;     /* equivalente a vect[3]=vect[2]*2.0; */
```

– Si se supone que **p=vect**, la relación entre *punteros* y *vectores* puede resumirse como se indica en las líneas siguientes:

```
*p     equivale a vect[0], a *vect     y a p[0]
*(p+1) equivale a vect[1], a *(vect+1) y a p[1]
*(p+2) equivale a vect[2], a *(vect+2) y a p[2]
```

Como ejemplo de la relación entre vectores y punteros, se van a ver varias formas posibles para sumar los N elementos de un vector **a[ ]**. Supóngase la siguiente declaración y las siguientes sentencias:

```
int a[N], suma, i, *p;

for(i=0, suma=0; i<N; ++i)          /* forma 1 */
     suma += a[i];

for(i=0, suma=0; i<N; ++i)          /* forma 2 */
     suma += *(a+i);

for(p=a, i=0, suma=0; i<N; ++i)     /* forma 3 */
     suma += p[i];

for(p=a, suma=0; p<&a[N]; ++p)      /* forma 4 */
     suma += *p;
```

6.2.2. RELACIÓN ENTRE MATRICES Y PUNTEROS

En el caso de las *matrices* la relación con los *punteros* es un poco más complicada. Supóngase una declaración como la siguiente

```
int mat[5][3], *p;
```

El *nombre de la matriz* (**mat**) es un *puntero* al primer elemento de un *vector de punteros* **mat[ ]** (por tanto, existe un vector de punteros que tiene también el mismo nombre que la matriz), cuyos elementos contienen las direcciones del primer elemento de cada fila de la matriz. El nombre **mat** es pues un *puntero a puntero*. El *vector de punteros* **mat[ ]** se crea automáticamente al crearse la matriz. Así pues, **mat** es igual a **&mat[0]**; y **mat[0]** es **&mat[0][0]**. Análogamente, **mat[1]** es **&mat[1][0]**, **mat[2]** es **&mat[2][0]**, etc. La dirección base sobre la que se direccionan todos los elementos de la matriz no es **mat**, sino **&mat[0][0]**. Recuérdese también que, por la relación entre vectores y punteros, (**mat+i**) apunta a **mat[i]**. Recuérdese que la fórmula de direccionamiento de una matriz de N filas y M columnas establece que la dirección del elemento (i, j) viene dada por:

dirección (i, j) = dirección (0, 0) + i*M + j

Teniendo esto en cuenta y haciendo **p = mat;** se tendrán las siguientes formas de acceder a los elementos de la matriz:

```
*p     es el valor de mat[0]      **p         es mat[0][0]
*(p+1) es el valor de mat[1]      **(p+1)     es mat[1][0]
                                  *(*(p+1)+1) es mat[1][1]
```

Por otra parte, si la matriz tiene M columnas y si se hace **p = &mat[0][0]** (dirección base de la matriz. Recuérdese que esto es diferente del caso anterior **p = mat**), el elemento **mat[i][j]** puede ser accedido de varias formas. Basta recordar que dicho elemento tiene por delante **i** filas completas, y **j** elementos de su fila:

```
*(p + M*i + j)     /* fórmula de direccionamiento */
*(mat[i] + j)      /* primer elemento fila i desplazado j elementos */
(*(mat + i))[j]    /* [j] equivale a sumar j a un puntero */
*((*(mat 0+ i)) + j)
```

Todas estas relaciones tienen una gran importancia, pues implican una correcta comprensión de los punteros y de las matrices. De todas formas, hay que indicar que las *matrices* no son del todo idénticas a los *vectores de punteros*: Si se define una matriz explícitamente por medio de vectores de punteros, las filas pueden tener diferente número de elementos, y no queda garantizado que estén contiguas en la memoria (aunque se puede hacer que sí lo sean). No sería pues posible en este caso utilizar la fórmula de direccionamiento y el acceder por columnas a los elementos de la matriz. La Figura 6.1 resume gráficamente la relación entre matrices y vectores de punteros.

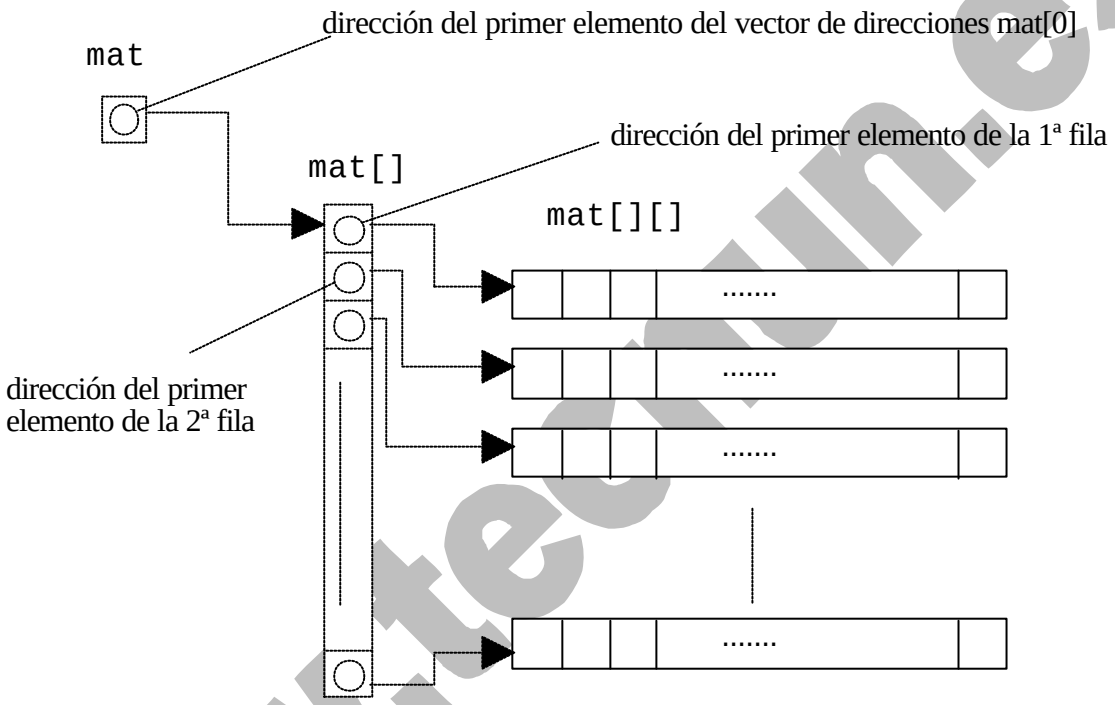


Figura 6.1. Relación entre matrices y punteros.

6.2.3. INICIALIZACIÓN DE VECTORES Y MATRICES

La inicialización de un *array* se puede hacer de varias maneras:

– Declarando el array como tal e inicializándolo luego mediante lectura o asignación por medio de un bucle ***for***:

```
double vect[N];
...
for(i = 0; i < N; i++)
     cin >> vect[i];
...
```

– Inicializándolo en la misma declaración, en la forma:

```
double   v[6] = {1., 2., 3., 3., 2., 1.};
float     d[] = {1.2, 3.4, 5.1};         /* d[3] está implícito */
int    f[100] = {0};                     /* todo se inicializa a 0 */
int     h[10] = {1, 2, 3};               /* restantes elementos a 0 */
int mat[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

6.3. Estructuras

Una ***estructura*** es *una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificador*. Por ejemplo, supóngase que se desea diseñar una estructura que guarde los datos correspondientes a un alumno de primero. Esta *estructura*, a la que se llamará ***alumno***, deberá guardar el nombre, la dirección, el número de matrícula, el teléfono, y las notas en las 10 asignaturas. Cada uno de estos datos se denomina ***miembro*** de la estructura.
El *modelo* o *patrón* de esta estructura puede crearse del siguiente modo:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
};
```

El código anterior crea el *tipo* de dato **alumno**, pero aún no hay ninguna variable declarada con este nuevo tipo. Obsérvese la necesidad de incluir un carácter (;) después de cerrar las llaves. Para declarar dos variables de tipo **alumno** se debe utilizar la sentencia

```
alumno alumno1, alumno2;
```

donde tanto **alumno1** como **alumno2** son una estructura, que podrá almacenar un nombre de hasta 30 caracteres, una dirección de hasta 20 caracteres, el número de matrícula, el número de teléfono y las notas de las 10 asignaturas. También podrían haberse definido **alumno1** y **alumno2** al mismo tiempo que se definía la estructura de tipo **alumno**. Para ello bastaría haber hecho:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
} alumno1, alumno2;
```

Para acceder a los miembros de una estructura se utiliza el ***operador punto*** (.), precedido por el nombre de la *estructura* y seguido del nombre del *miembro*. Por ejemplo, para dar valor al **telefono** del alumno **alumno1** el valor 943903456, se escribirá:

```
alumno1.telefono = 943903456;
```

y para guardar la dirección de este mismo alumno, se escribirá:

```
strcpy(alumno1.direccion , "C/ Penny Lane 1,2-A");
```

El tipo de estructura creado se puede utilizar para definir más variables o estructuras de tipo **alumno**, así como vectores de estructuras de este tipo. Por ejemplo:

```
alumno nuevo_alumno, clase[300];
```

En este caso, **nuevo_alumno** es una estructura de tipo **alumno**, y **clase[300]** es un *vector de estructuras* con espacio para almacenar los datos de 300 alumnos. El número de matrícula del alumno 264 podrá ser accedido como **clase[264].no_matricula**.

Los *miembros* de las estructuras pueden ser variables de cualquier tipo, incluyendo vectores y matrices, e incluso otras estructuras previamente definidas. Las estructuras se diferencian de los *arrays* (vectores y matrices) en varios aspectos. Por una parte, los *arrays* contienen información

múltiple pero homogénea, mientras que los *miembros* de las estructuras pueden ser de naturaleza muy diferente. Además, *las estructuras permiten ciertas operaciones globales que no se pueden realizar con arrays*. Por ejemplo, la sentencia siguiente:

```
clase[298] = nuevo_alumno;
```

hace que se copien todos los miembros de la estructura **nuevo_alumno** en los miembros correspondientes de la estructura **clase[298]**. Estas operaciones globales no son posibles con *arrays*.

Se pueden definir también ***punteros a estructuras***:

```
alumno *pt;

pt = &nuevo_alumno;
```

Ahora, el puntero **pt** apunta a la estructura **nuevo_alumno** y esto permite una nueva forma de acceder a sus miembros utilizando el ***operador flecha*** (->), constituido por los signos (-) y (>). Así, para acceder al teléfono del alumno **nuevo_alumno**, se puede utilizar cualquiera de las siguientes sentencias:

```
pt->telefono;
(*pt).telefono;
```

donde el paréntesis es necesario por la mayor prioridad del operador (.) respecto a (*).

Las estructuras admiten los mismos modos *auto*, *extern* y *static* que los *arrays* y las variables escalares. Las reglas de inicialización a cero por defecto de los modos *extern* y *static* se mantienen. Por lo demás, una estructura puede inicializarse en el momento de la declaración de modo análogo a como se inicializan los vectores y matrices. Por ejemplo, una forma de declarar e inicializar a la vez la estructura **alumno_nuevo** podría ser la siguiente:

```
struct alumno {
     char nombre[31];
     char direccion[21];
     unsigned long no_matricula;
     unsigned long telefono;
     float notas[10];
} alumno_nuevo = {"Mike Smith", "San Martín 87, 2º A", 62419, 421794};
```

donde, como no se proporciona valor para las notas, estas se inicializan a cero.

Las estructuras constituyen uno de los aspectos más potentes del lenguaje C++. En esta sección se ha tratado sólo de hacer una breve presentación de sus posibilidades.

6.4. Gestión dinámica de la memoria

Según lo visto hasta ahora, *la reserva o asignación de memoria* para vectores y matrices se hace de forma automática con la declaración de dichas variables, asignando suficiente memoria para resolver el problema de tamaño máximo, dejando el resto sin usar para problemas más pequeños. Así, si en una función encargada de realizar un producto de matrices, éstas se dimensionan para un tamaño máximo (100, 100), con dicha función se podrá calcular cualquier producto de un tamaño igual o inferior, pero aun en el caso de que el producto sea por ejemplo de tamaño (3, 3), la memoria reservada corresponderá al tamaño máximo (100, 100). Es muy útil el poder reservar más o menos memoria *en tiempo de ejecución*, según el tamaño del caso concreto que se vaya a resolver. A esto se llama ***reserva o gestión dinámica de memoria.***

Existe en C++ un operador que reserva la cantidad de memoria deseada en tiempo de ejecución. Se trata del operador ***new***, del que ya hablamos en el apartado 4.1.6. Este operador se utiliza de la siguiente forma:

```
tipo_variable *vector;

vector = new tipo_variable [variable];
```

El operador ***new*** se utiliza para crear variables de cualquier tipo, ya sean las estándar o las definidas por el usuario. La mayor ventaja de esta gestión de memoria es que se puede definir el tamaño del vector por medio de una variable que toma el valor adecuado en cada ejecución.

Existe también un operador llamado **delete** que deja libre la memoria reservada por **new** y que ya no se va a utilizar.Recordemos que cuando la reserva es dinámica la variable creada perdura hasta que sea explícitamente borrada por este operador. La memoria no se libera por defecto.

El prototipo de este operador es el siguiente:

```
delete [] vector;
```

A continuación se presenta a modo de ejemplo un programa que reserva memoria de modo dinámico para un vector de caracteres:

```
#include <iostream.h>
#include <string.h>

void main()
{
    char Nombre[50];
    cout << "Introduzca su Nombre:";
    cin  >> Nombre;

    char *CopiaNombre = new char[strlen(Nombre)+1];

    // Se copia el Nombre en la variable CopiaNombre
    strcpy(CopiaNombre, Nombre);
    cout << CopiaNombre;

    delete [] CopiaNombre;
}
```

El siguiente ejemplo reserva memoria dinámicamente para una matriz de ***doubles***:

```
#include <iostream.h>

void main(){
    int    nfil, ncol, i, j;
    double **mat;
    //se pide al usuario el tamaño de la matriz
    cout << “Introduzca numero de filas y columnas: “;
    cin >> nfil >> ncol;

    // se reserva memoria para el vector de punteros
    mat = new double*[nfil];
    // se reserva memoria para cada fila
    for (i=0; i<nfil; i++)
        mat[i] = new double[ncol];

    // se inicializa toda la matriz
    for(i=0; i<nfil; i++)
```

```
            for(j=0; j<ncol; j++)
                mat[i][j]=i+j;

        // se imprime la matriz
        for(i=0; i<nfil; i++){
            for(j=0; j<ncol; j++)
                cout << mat[i][j] << "\t";
            cout << "\n";
        }

    ....// se libera memoria
        for(i=0; i<nfil; i++)
            // se borran las filas de la matriz
            delete [] mat[i];
        // se borra el vector de punteros
        delete [] mat;
        }
    }
```

7. FUNCIONES

Como se explicó en la Sección 1.2, una ***función*** es una parte de código independiente del programa principal y de otras funciones, que puede ser llamada enviándole unos datos o no, para que realice una determinada tarea y/o proporcione unos resultados. Las funciones son una parte muy importante del lenguaje C++. En los apartados siguientes se describen los aspectos más importantes de las funciones.

7.1. Utilidad de las funciones

Parte esencial del correcto diseño de un programa de ordenador es su *modularidad*, esto es su división en partes más pequeñas de finalidad muy concreta, que en C++ reciben el nombre de ***funciones***. Las funciones facilitan el desarrollo y mantenimiento de los programas, evitan errores, y ahorran memoria y trabajo innecesario. Una misma función puede ser utilizada por diferentes programas, y por tanto no es necesario reescribirla. Además, una función es una parte de código independiente del programa principal y de otras funciones, manteniendo una gran independencia entre las variables respectivas, y evitando errores y otros efectos colaterales de las modificaciones que se introduzcan.

Mediante el uso de funciones se consigue un código limpio, claro y elegante. La adecuada división de un programa en funciones constituye un aspecto fundamental en el desarrollo de programas de cualquier tipo.

7.2. Definición de una función

La ***definición de una función*** consiste en la definición del código necesario para que ésta realice las tareas para las que ha sido prevista. La definición de una función se debe realizar en alguno de los ficheros que forman parte del programa. La forma general de la definición de una función es la siguiente:

```
tipo_valor_de_retorno  nombre_funcion(lista de parámetros)
{
     declaración de variables y/o de otras funciones
     codigo ejecutable
     return (expresión);         /* optativo */
}
```

La primera línea recibe el nombre de ***encabezamiento*** (header) y el resto de la definición –encerrado entre llaves– es el ***cuerpo*** de la función. Cada función puede disponer de sus propias variables, *declaradas* al comienzo de su código. Estas variables, por defecto, son de tipo ***auto***, es decir, sólo son visibles dentro del bloque en el que han sido definidas y permanecen ocultas para el resto del programa. También pueden hacerse visibles a la función *variables globales* definidas en otro fichero (o en el mismo fichero, si la definición está por debajo de donde se utilizan), declarándolas con la palabra clave ***extern***.

El *código ejecutable* es el conjunto de instrucciones que deben ejecutarse cada vez que la función es llamada. La *lista de parámetros*, también llamados ***argumentos formales***, es una lista de declaraciones de variables, precedidas por su *tipo* correspondiente y separadas por comas (,). Los *argumentos formales* son la forma más natural y directa para que la función reciba valores desde el programa que la llama, correspondiéndose en número y tipo con otra lista de argumentos -

los ***argumentos actuales***- en el programa que realiza la llamada a la función. Los *argumentos formales* son declarados en el encabezamiento de la función, pero no pueden ser inicializados en él.

Cuando una función es ejecutada, puede devolver al programa que la ha llamado un valor (el *valor de retorno*), cuyo tipo debe ser especificado en el encabezamiento de la función (si no se especifica, se supone por defecto el tipo *int*). Si no se desea que la función devuelva ningún valor, el *tipo del valor de retorno* deberá ser ***void***.

La sentencia ***return*** permite devolver el control al programa que llama. Puede haber varias sentencias *return* en una misma función. Si no hay ningún *return*, el control se devuelve cuando se llega al final del *cuerpo* de la función. Le palabra clave *return* puede ir seguida de una *expresión*, en cuyo caso ésta es evaluada y el valor resultante devuelto al programa que llama como *valor de retorno* (si hace falta, con una conversión previa al *tipo* declarado en el encabezamiento). Los paréntesis que engloban a la *expresión* que sigue a *return* son optativos. El valor de retorno es un valor único: *no puede ser un vector o una matriz*, aunque sí un *puntero* a un vector o a una matriz. Sin embargo, *el valor de retorno sí puede ser una estructura*, que a su vez puede contener vectores y matrices como elementos miembros.

Como ejemplo supóngase que se va a calcular a menudo el *valor absoluto* de variables de tipo *double*. Una solución es definir una función que reciba como argumento el valor de la variable y devuelva ese valor absoluto como valor de retorno. La definición de esta función podría ser como sigue:

```
double valor_abs(double x)
{
     if (x < 0.0)
          return -x;
     else
          return x;
}
```

7.3. Declaración y llamada de una función

7.3.1. DECLARACIÓN DE UNA FUNCIÓN

De la misma manera que en C++ es necesario declarar todas las variables, también *toda función debe ser declarada* antes de ser utilizada en la función o programa que realiza la llamada. De todas formas, ahora se verá que aquí hay una mayor flexibilidad que en el caso de las variables.

La declaración de una función se puede hacer de dos maneras:

a) Mediante una ***definición*** previa de la función. Esta práctica es segura si la definición precede a la *llamada*, pero tiene el inconveniente de que si la definición se cambia de lugar el programa no funcionará.

b) Mediante una ***declaración*** explícita, previa a la *llamada*. Esta es la práctica más segura y la que hay que tratar de seguir siempre. La declaración de la función se hace mediante el ***prototipo*** de la función, bien fuera de cualquier bloque, bien en la parte de declaraciones de un bloque.

   La forma general del *prototipo* de una función es la siguiente:

```
tipo_valor_de_retorno  nombre_funcion(lista_de_tipos_de_parámetros);
```

Esta forma general coincide sustancialmente con la primera línea de la definición -el encabezamiento-, con dos pequeñas diferencias: en vez de la lista de argumentos formales o parámetros, en el *prototipo* basta incluir los *tipos* de dichos argumentos. Se pueden incluir también identificadores a continuación de los tipos, pero son ignorados por el compilador. Además, una segunda diferencia es que el *prototipo* termina con un carácter (;). Cuando no hay argumentos formales, se pone entre los paréntesis la palabra ***void***, y se pone también ***void*** precediendo al nombre de la función cuando no hay valor de retorno.

Los *prototipos* permiten que el compilador realice correctamente la conversión del tipo del valor de retorno, y de los *argumentos actuales* a los tipos de los *argumentos formales*. La declaración de las funciones mediante los *prototipos* suele hacerse al comienzo del fichero, después de los **#define** e **#include**. En muchos casos –particularmente en programas grandes, con muchos ficheros y muchas funciones–, se puede crear un fichero (con la extensión **.h**) con todos los prototipos de las funciones utilizadas en un programa, e incluirlo con un **#include "nombre_del_fichero.h"** en todos los ficheros en que se utilicen dichas funciones.

7.3.2. LLAMADA A UNA FUNCIÓN

La ***llamada a una función*** se hace incluyendo su *nombre*, seguido de una lista de *argumentos* separados por comas y encerrados entre paréntesis, en una expresión o sentencia del programa principal o de otra función. A los argumentos incluidos en la llamada se les llama ***argumentos actuales***, y pueden ser no sólo variables y/o constantes, sino también *expresiones*. Cuando el programa que llama encuentra el nombre de la función, evalúa los *argumentos actuales* contenidos en la llamada, los convierte si es necesario al tipo de los *argumentos formales*, y pasa *copias* de dichos valores a la función junto con el control de la ejecución.

En general el número de *argumentos actuales* en la llamada a una función debe coincidir con el número de *argumentos formales* en la definición y en la declaración, aunque hay dos excepciones:

a) Existe la posibilidad de definir funciones con un número variable o indeterminado de argumentos. Para definir estas funciones se utilizan los *puntos suspensivos* (...), que representan los argumentos desconocidos que puede haber. Un ejemplo de función de este tipo es el siguiente:

```
void mi_funcion(int i, double a, ...);
```

donde los argumentos ***i*** y ***a*** tendrían que estar siempre presentes. Para conocer con más detalle cómo se crean estas funciones se recomienda acudir a alguno de los textos de C++ recomendados en la Bibliografía.

b) En C++ se pueden definir ***valores por defecto*** para todos o algunos de los argumentos formales. Después, en la llamada, en el caso de que algún argumento esté ausente de la lista de argumentos actuales, se toma el valor asignado por defecto a ese argumento. Por ejemplo, la función ***modulo()*** podía haberse declarado del siguiente modo:

```
double modulo(double x[], int n=3);
```

La función ***modulo()*** puede ser llamada en C++ de las formas siguientes:

```
v = modulo(x, n);
v = modulo(x);
```

En el segundo caso se utiliza el valor por defecto ***n=3*** incluido en la declaración.

En C++ se exige que todos los argumentos con valores por defecto estén *al final de la lista de argumentos*. En la llamada a la función pueden omitirse alguno o algunos de los últimos argumentos de la lista. Si se omite un argumento deben de omitirse todos aquellos que se encuentren detrás suyo.

Cuando se llama a una función, después de realizar la conversión de los argumentos actuales, se ejecuta el código correspondiente a la función hasta que se llega a una sentencia ***return*** o al final del cuerpo de la función, y entonces se devuelve el control al programa que realizó la llamada, junto con el *valor de retorno* si es que existe (convertido previamente al *tipo* especificado en el *prototipo*, si es necesario). Recuérdese que el valor de retorno puede ser un valor numérico, una dirección (un puntero), o una estructura, pero no una matriz o un vector.

La llamada a una función puede hacerse de muchas formas, dependiendo de qué clase de tarea realice la función. Si su papel fundamental es *calcular un valor de retorno* a partir de uno o más argumentos, lo más normal es que sea llamada incluyendo su nombre seguido de los argumentos actuales en una *expresión aritmética* o de otro tipo. En este caso, la llamada a la función hace el papel de un operando más de la expresión.

En otros casos, *no existirá valor de retorno* y la llamada a la función se hará incluyendo en el programa una sentencia que contenga solamente el nombre de la función, siempre seguido por los argumentos actuales entre paréntesis y terminando con un carácter (;).

La declaración y la llamada de la función **valor_abs()** antes definida, se podría realizar de la forma siguiente:

```
double valor_abs(double);          /* declaración */

void main (void)
{
     double z, y;

     y = -30.8;
     z = valor_abs(y) + y*y;        /* llamada en una expresion */
}
```

La función **valor_abs()** recibe un valor de tipo *double*. El valor de retorno de dicha función (el valor absoluto de **y**), es introducido en la expresión aritmética que calcula **z**.

La declaración `(double valor_abs(double))` no es estrictamente necesaria cuando la definición de la función está en el mismo archivo que **main()** y está definida antes que la llamada.

7.4. Especificador *inline* para funciones

C++ permite sustituir, en tiempo de compilación, la llamada a una función por el código correspondiente en el punto en que se realiza la llamada. De esta manera la ejecución es más rápida, pues no se pierde tiempo transfiriendo el control y realizando conversiones de parámetros. Como contrapartida, el programa resultante ocupa más memoria, pues es posible que el código de una misma función se introduzca muchas veces, con las repeticiones consiguientes. Las funciones ***inline*** resultan interesantes en el caso de funciones muy breves, que aparecen en pocas líneas de código pero que se ejecutan muchas veces (en un bucle ***for***, por ejemplo). Existen 2 formas de definirlas:

1. Una primera forma de utilizar funciones ***inline*** es anteponer dicha palabra en la declaración de la función, como por ejemplo:

```
inline void permutar(int &a, int &b);
```

2. Otra forma de utilizar funciones ***inline*** sin necesidad de utilizar esta palabra es introducir el código de la función en la ***declaración*** (convirtiéndose de esta manera en ***definición***), poniéndolo entre llaves { } a continuación de ésta. Este segundo procedimiento suele utilizarse por medio de ficheros ***header*** (****.h***), que se incluyen en todos los ficheros fuente que tienen que tener acceso al código de las funciones ***inline***. Considérese el siguiente ejemplo, consistente en una declaración seguida de la definición:

```
void permutar (int *i, int *j) {
int temp; temp = *i; *i = *j; *j = temp; }
```

En cualquier caso, la directiva ***inline*** es sólo una *recomendación al compilador*, y éste puede desestimarla por diversas razones, como coste de memoria excesivo, etc.

7.5. Paso de argumentos por valor y por referencia

En la sección anterior se ha comentado que en la llamada a una función los ***argumentos actuales*** son evaluados y se pasan ***copias*** de estos valores a las variables que constituyen los ***argumentos formales*** de la función. Aunque los argumentos actuales sean variables y no expresiones, y haya una correspondencia biunívoca entre ambos tipos de argumentos, los cambios que la función realiza en los argumentos formales no se trasmiten a las variables del programa que la ha llamado, precisamente porque lo que la función ha recibido son *copias*. El modificar una copia no repercute en el original. A este mecanismo de paso de argumentos a una función se le llama ***paso por valor***. Considérese la siguiente función para permutar el valor de sus dos argumentos **x** e **y**:

```
void permutar(double x, double y)          /* funcion incorrecta
*/
{
     double temp;
     temp = x;
     x = y;
     y = temp;
}
```

La función anterior podría ser llamada y comprobada de la siguiente forma:

```
#include <iostream.h>

void main(void)
{
     double a=1.0, b=2.0;
     void permutar(double, double);

     cout << a << "," << b << endl;
     permutar(a, b);
     cout << a << "," << b << endl;
}
```

Compilando y ejecutando este programa se ve que **a** y **b** siguen teniendo los mismos valores antes y después de la llamada a **permutar()**, a pesar de que en el interior de la función los valores sí se han permutado (es fácil de comprobar introduciendo en la función los **cout** correspondientes). La razón está en que *se han permutado los valores de las copias* de **a** y **b**, pero no los valores de las propias variables. Las variables podrían ser permutadas si se recibieran sus direcciones (o copias de dichas direcciones). Las direcciones deben recibirse en *variables puntero*, por lo que los

argumentos formales de la función deberán ser punteros. Una versión correcta de la función **permutar()** que pasa direcciones en vez de valores sería como sigue:

```
void permutar(double *x, double *y)
{
     double temp;
     temp = *x;
     *x = *y;
     *y = temp;
}
```

que puede ser llamada y comprobada de la siguiente forma:

```
#include <iostream.h>

void main(void)
{
     double a=1.0, b=2.0;
     void permutar(double *, double *);

     cout << a << "," << b << endl;
     permutar(&a, &b);
     cout << a << "," << b << endl;
}
```

Al mecanismo de paso de argumentos mediante direcciones en lugar de valores se le llama ***paso por referencia***, y deberá utilizarse siempre que la función deba devolver argumentos modificados.

C++ ofrece otra forma de ***pasar argumentos por referencia*** a una función, que no obliga a utilizar –dentro de la función– el operador ***indirección*** (*) para acceder al valor de la variable que se quiere modificar. Esto se hace por medio de un nuevo tipo de dato llamado tipo ***referencia (reference)***.

Las ***variables referencia*** se declaran por medio del carácter (&)[5]. Por lo demás, son variables normales que contienen un valor numérico o alfanumérico. Antes de pasar a explicarlas con más detenimiento, se presenta de nuevo el ejemplo de la función ***permutar()*** utilizando variables ***referencia*** en lugar de ***punteros***.

```
// Este programa requiere compilador de C++

#include <iostream.h>

void main(void){
     double a=1.0, b=2.0;
     void permutar(double &, double &);  /* los argumentos son referencias
*/

     cout << a << "," << b << endl;
     permutar(a, b);             /* en la llamada no se usa ni (*) ni (&) */
     cout << a << "," << b << endl;
}
```

[5] No se debe confundir el uso de (&) en la declaración de una ***referencia*** con el operador ***dirección*** (&), de la misma forma que no se debe confundir el carácter (*) en la declaración de un ***puntero***, con el operador ***indirección*** (*).

En este caso la definición de la función ***permutar()*** sería la siguiente:

```
void permutar(double &a, double &b)            // los argumentos son
referencias
{
    double temp;

    temp = a;                          // no hace falta utilizar
    a = b;                             //el operador indirección (*)
    b = temp;
}
```

Los dos programas dan idéntico resultado, sin embargo, el segundo tiene la ventaja de que no hay que utilizar el operador ***indirección*** dentro de la función ***permutar()***. C++ permite pasar argumentos por referencia sin más que anteponer el carácter (&) a los argumentos correspondientes, tanto en el prototipo como en el encabezamiento de la definición. En la llamada a la función los argumentos se ponen directamente, sin anteponerles ningún carácter u operador.

En C++ existe realmente un tipo llamado ***referencia*** que va más allá del paso de argumentos a funciones tal y como se acaba de explicar. Las variables de tipo ***referencia*** se declaran con el operador (&) y *deben ser inicializadas a otra variable o a un valor numérico*. Por ejemplo:

```
int  i=2;
int& iref = i;     // declaración de referencia válida
int& jref;         // declaración de referencia no válida
```

La variable ***i*** es una variable normal tipo ***int***. La variable ***iref*** es una variable ***referencia*** que se asocia con ***i***, en el sentido de que ambas variables comparten la misma posición de memoria: si se modifica ***i*** se modifica ***iref***, y viceversa. En este sentido, ***iref*** es un ***alias*** de ***i***. La diferencia con un puntero que apuntase a la dirección de ***i*** está en que, una vez que una variable ***referencia*** ha sido declarada como ***alias*** de ***i*** no puede ser declarada como ***alias*** de otra variable. Siempre se referirá a la misma posición de memoria. Es como un puntero a una posición de memoria fija. En la función ***permutar()*** los ***argumentos formales***, que son ***referencias***, se inicializan y se convierten en ***alias*** de los ***argumentos actuales***, que son variables ordinarias.

El principal uso de las variables ***referencia*** es como *valor de retorno* o *argumentos* de funciones. Los vectores y matrices (***arrays***) no pueden ser declarados como variables referencia, porque ya tienen una forma propia y natural de ser pasados como argumentos a una función.

El que una función tenga como *valor de retorno* una variable tipo ***referencia*** permite utilizarla de una manera un poco singular. Considérese el siguiente ejemplo:

```
int& maxref(int& a, int& b)
{
if (a >= b)
    return a;
else
    return b;
}
```

La función ***maxref()*** tiene ***referencias*** como *valor de retorno* y como *argumentos*. Esto permite utilizarla, por ejemplo, del siguiente modo:

```
maxref(i, j) = 0;
```

Ésta es una forma un poco extraña de utilizar una función: la llamada está a la izquierda del operador de asignación, en vez de aparecer a la derecha en una expresión aritmética o de otro tipo. El resultado de esta llamada también es un poco extraño: el valor de retorno es una ***referencia***, esto

es un *alias* del argumento de valor máximo. Cuando la llamada a la función se sustituye por su valor de retorno, el resultado de la sentencia anterior es que la variable pasada como argumento que tiene mayor valor se hace igual a cero. Este mismo efecto puede conseguirse mediante ***punteros***, pero con ***referencias*** resulta mucho más sencillo.

En C++ las ***referencias*** son muy utilizadas para pasar argumentos a funciones (y como valores de retorno), no sólo para poderlos modificar dentro de la función, sino también por motivos de eficiencia, pues es mucho más rápido pasar un *puntero* o un *alias* de una variable que una copia del valor de esa variable. Si además la variable es una *estructura*, las ventajas de eficiencia son todavía mucho más palpables.

7.6. La función main() con argumentos

Cuando se ejecuta un programa desde ***MS-DOS*** tecleando su nombre, existe la posibilidad de pasarle algunos datos, tecleándolos a continuación en la misma línea. Por ejemplo, se le puede pasar algún valor numérico o los nombres de algunos ficheros en los que tiene que leer o escribir información. Esto se consigue por medio de argumentos que se pasan a la función **main()**, como se hace con otras funciones.

Otra manera de pasarle argumentos a **main()** es por medio del menú *project / settings / debug*, donde se pueden introducir los argumentos en la casilla *program arguments.*

Así pues, a la función **main()** se le pueden pasar argumentos y también puede tener valor de retorno. El primero de los argumentos de **main()** se suele llamar **argc**, y es una variable *int* que contiene el número de palabras que se teclean a continuación del nombre del programa cuando éste se ejecuta. El segundo argumento se llama **argv**, y es un *vector de punteros a carácter* que contiene las direcciones de la primera letra o carácter de dichas palabras. A continuación se presenta un ejemplo:

```
int main(int argc, char *argv[])
{
     int cont;
     for (cont=0; cont<argc; cont++){
          cout <<"El argumento es: \n" << argv[cont];
          cout << endl;
     }
     return 0;
}
```

7.7. Punteros como valor de retorno

A modo de resumen, recuérdese que una función es un conjunto de instrucciones C++ que:

- Es llamado por el programa principal o por otra función.
- Recibe datos a través de una lista de argumentos, o a través de variables *extern*.
- Realiza una serie de tareas específicas, entre las que pueden estar cálculos y operaciones de lectura/escritura en el disco, en teclado y pantalla, etc.
- Devuelve resultados al programa o función que la ha llamado por medio del *valor de retorno* y de los *argumentos* que hayan sido *pasados por referencia* (punteros).

El utilizar *punteros como valor de retorno* permite superar la limitación de devolver un único valor de retorno. Puede devolverse un puntero al primer elemento de un vector o a la dirección base de una matriz, lo que equivale a devolver múltiple valores. El valor de retorno ***puntero a void*** (***void ****) es un puntero de tipo indeterminado que puede asignarse sin *casting* a un puntero de cualquier tipo.

7.8. Paso de arrays como argumentos a una función

Para considerar el paso de ***arrays*** (vectores y matrices) como argumentos de una función, hay que recordar algunas de sus características, en particular su relación con los ***punteros*** y la forma en la que las matrices se almacenan en la memoria. Este tema se va a presentar por medio de un ejemplo: el producto de matriz cuadrada por vector ([a]{x}={y}).

Para la definición de la función es necesario dar las dimensiones de la matriz (excepto la 1ª, es decir, excepto el nº de filas), con objeto de poder reconstruir la fórmula de direccionamiento, en la que interviene el número de columnas pero no el de filas. El encabezamiento de la definición sería como sigue:

```
void prod(int n, double a[][10], double x[], double y[])
{...}
```

Dicho encabezamiento se puede también establecer en la forma:

```
void prod(int n, double (*a)[10], double *x, double *y)
{...}
```

donde el paréntesis es necesario para que sea "puntero a vector de tamaño 10". Sin paréntesis sería "vector de tamaño 10, cuyos elementos son punteros", por la mayor prioridad del operador [ ] sobre el operador *.

La declaración de la función **prod()** se puede hacer en la forma:

```
void prod(int, double a[][10], double x[], double y[]);
```

o bien,

```
void prod(int n, double (*a)[10], double *x, double *y);
```

Para la llamada basta simplemente utilizar los nombres de los argumentos:

```
double a[10][10], x[10], y[10];
...
prod(nfilas, a, x, y);
...
```

7.9. Funciones recursivas

La recursividad es la posibilidad de que una función se llame a sí misma, bien directa o indirectamente. Un ejemplo típico es el cálculo del factorial de un número, definido en la forma:

```
N! = N * (N-1)! = N * (N-1) * (N-2)! = N * (N-1)*(N-2)*...*2*1
```

La función ***factorial***, escrita de forma recursiva, sería como sigue:

```
unsigned long factorial(unsigned long numero)
{
     if ( numero == 1 || numero == 0 )
          return 1;
```

```
        else
            return numero*factorial(numero-1);
}
```

Supóngase la llamada a esta función para N=4, es decir **factorial(4)**. Cuando se llame por primera vez a la función, la variable **numero** valdrá 4, y por tanto devolverá el valor de ***4*factorial(3);*** pero **factorial(3)** devolverá ***3*factorial(2)***; **factorial(2)** a su vez es ***2*factorial(1)*** y dado que **factorial(1)** es igual a 1 (es importante considerar que sin éste u otro caso particular, la función recursiva no terminaría nunca de llamarse a sí misma), el resultado final será 4*(3*(2*1)).

Por lo general la recursividad no ahorra memoria, pues ha de mantenerse una pila[6] con los valores que están siendo procesados. Tampoco es más rápida, sino más bien todo lo contrario, pero el código recursivo es más compacto y a menudo más sencillo de escribir y comprender.

7.10. Sobrecarga de funciones

La sobrecarga (***overload***) de funciones consiste en declarar y definir varias funciones distintas que tienen *un mismo nombre*. Dichas funciones se definen de forma diferente. En el momento de la ejecución se llama a una u otra función dependiendo del número y/o tipo de los argumentos actuales de la llamada a la función. Por ejemplo, se pueden definir varias funciones para calcular el valor absoluto de una variable, todas con el mismo nombre ***abs()***, pero cada una aceptando un tipo de argumento diferente y con un valor de retorno diferente.

La sobrecarga de funciones no admite funciones que difieran sólo en el tipo del valor de retorno, pero con el mismo número y tipo de argumentos. De hecho, el valor de retorno no influye en la determinación de la función que es llamada; sólo influyen el número y tipo de los argumentos. Tampoco se admite que la diferencia sea el que en una función un argumento se pasa por valor y en otra función ese argumento se pasa por referencia.

A continuación se presenta un ejemplo con dos funciones sobrecargadas, llamadas ambas ***string_copy()***, para copiar cadenas de caracteres. Una de ellas tiene dos argumentos y la otra tres. Cada una de ellas llama a una de las funciones estándar del C++: ***strcpy()*** que requiere dos argumentos, y ***strncpy()*** que requiere tres. El número de argumentos en la llamada determinará la función concreta que vaya a ser ejecutada:

```
// Ejemplo de función sobrecargada
#include <iostream.h>
#include <string.h>

inline void string_copy(char *copia, const char *original){
    strcpy(copia, original);
}

inline void string_copy(char *copia, const char *original, const int
longitud){
    strncpy(copia, original, longitud);
}

static char string_a[20], string_b[20];

void main(void){
```

[6] Una *pila* es un tipo especial de estructura de datos que se estudiará en Informática II.

```
    string_copy(string_a, "Aquello");
    string_copy(string_b, "Esto es una cadena", 4);
    cout << string_b << " y " << string_a;
}
```

7.11. Funciones para cadenas de caracteres

En C++, existen varias funciones útiles para el manejo de cadenas de caracteres. Las más utilizadas son: **strlen()**, **strcat()**, **strcmp()** y **strcpy()**. Sus prototipos o declaraciones están en el fichero **string.h**, y son los siguientes:

```
unsigned strlen(const char *s);
```

Explicación: Su nombre proviene de ***string length***, y su misión es contar el número de caracteres de una cadena, sin incluir el '\0' final. El paso del argumento se realiza *por referencia*, pues como argumento se emplea un puntero a la cadena (tal que el valor al que apunta es constante para la función; es decir, ésta no lo puede modificar), y devuelve un entero sin signo que es el número de caracteres de la cadena.

```
char *strcat(char *s1, const char *s2);
```

Explicación: Su nombre proviene de ***string concatenation***, y se emplea para unir dos cadenas de caracteres poniendo **s2** a continuación de **s1**. El valor de retorno es un puntero a **s1**. Los argumentos son los punteros a las dos cadenas que se desea unir. La función almacena la cadena completa en la primera de las cadenas. ¡PRECAUCIÓN! Esta función no prevé si tiene sitio suficiente para almacenar las dos cadenas juntas en el espacio reservado para la primera. Esto es responsabilidad del programador.

```
int strcmp(const char *s1, const char *s2)
```

Explicación: Su nombre proviene de ***string comparison***. Sirve para comparar dos cadenas de caracteres. Como argumentos utiliza punteros a las cadenas que se van a comparar. La función devuelve cero si las cadenas son iguales, un valor menor que cero si **s1** es menor –en orden alfabético– que **s2**, y un valor mayor que cero si **s1** es mayor que **s2**.

```
char *strcpy(char *s1, const char *s2)
```

Explicación: Su nombre proviene de ***string copy*** y se utiliza para copiar cadenas. Utiliza como argumentos dos punteros a carácter: el primero es un puntero a la cadena copia, y el segundo es un puntero a la cadena original. El valor de retorno es un puntero a la cadena copia **s1**.

8. FLUJOS DE ENTRADA/SALIDA

El lenguaje C++ no dispone de sentencias de entrada/salida. En su lugar se utilizan operadores contenidos en la librería estándar y que forman parte integrante del lenguaje.

La librería de C++ proporciona algunas herramientas para la entrada y salida de datos que la hacen más versátil, y más complicada en ocasiones, que la de C. En C++ las entradas son leídas desde ***streams*** y las salidas son escritas en ***streams***. La palabra stream quiere decir algo así como *canal*, *flujo* o *corriente.*

Los ***streams*** más utilizados para introducir y sacar datos son ***cin*** y ***cout*** respectivamente. Para la utilización de dichos ***streams*** es necesario incluir, al comienzo del programa, el archivo **iostream.h** en el que están definidos sus prototipos.:

```
#include <iostream.h>
```

donde ***iostream*** proviene de ***input-output-stream*** (flujo de entrada/salida)

8.1. Salida de datos

El ***stream*** o flujo de salida **cout** imprime en la unidad de salida (el monitor, por defecto), el texto, y las constantes y variables que se indiquen. Para poder insertar datos en un ***stream*** es necesario utilizar el ***operador de inserción*** (<<). Así, la forma general de utilizar el flujo de salida ***cout*** se puede estudiar viendo su *prototipo*:

```
cout << "texto" << variables o expresiones << ...;
```

Explicación: El objeto **cout** imprime el texto contenido entre las comillas tal y como está escrito. Entre las comillas se pueden incluir palabras reservadas (sin que C++ las interprete como tales) y las secuencias de escape, las cuales C++ ejecuta y no escribe. Separados por el operador (<<) se pueden incluir variables y/o expresiones (incluso las creadas por el usuario si el operador (<<) ha sido sobrecargado) que **cout** es capaz de intepretar correctamente e imprimirá en pantalla su valor.

Considérese el ejemplo siguiente,

```
int i = 1;
double tiempo = 10;
float  acel = 9.8;
const float Vo = 5;

cout << "Resultado numero: " << i << " En el instante " << tiempo << "
la velocidad vale " << Vo + acel * tiempo << "\n";
```

en el que se imprimen 2 variables (**i** y **tiempo**) y una expresión (el cálculo de la velocidad). Esto será lo que se imprimirá en pantalla:

Resultado numero: 1 En el instante 10 la velocidad vale 103

(Además el cursor pasará a la siguiente línea por la secuencia de escape (\n))

8.2. Entrada de datos

El ***stream*** o flujo **cin** es análoga en muchos aspectos a **cout**, y se utiliza para leer datos de la entrada estándar (que por defecto es el teclado). Junto con ***cin***, también es necesario utilizar el ***operador de extracción*** (>>). Así, la forma general de utilizar este flujo es la siguiente:

```
cin >> variable1 >> variable2 >> ...;
```

Explicación: El flujo **cin** almacena en las variables los datos introducidos por el teclado, interpretando el final de cada una al encontrar un espacio o un final de línea.

Se puede ver la ventaja de los operadores propios de C++ para la extracción de datos de consola, y es el hecho de que se evita el chequeo de compatibilidad de entre las variables. Es decir que la *variable1* puede ser tipo ***int*** y la *variable2* tipo ***double***.

Al trabajar con variables tipo ***char[ ]***, el flujo ***cin*** sólo acepta texto sin espacios. Si lo que se desea es introducir en la misma variable varias palabras (texto con espacios incluidos, por ejemplo un nombre y apellidos), se debe utilizar una función del flujo cin , de la siguiente forma:

```
cin.getline (variable1, límite_de_longitud, carácter_delimitador)
```

donde *límite_de_longitud* indica que en la variable no se almacenarán más caracteres que los señalados y *carácter_delimitador* precisa que si antes del límite se hallara el carácter señalado no se almacenaría nada más en la variable.

9. EL PREPROCESADOR

El ***preprocesador*** del lenguaje C++ permite sustituir ***macros*** (sustitución en el programa de constantes simbólicas o texto, con o sin parámetros), realizar compilaciones condicionales e incluir archivos, todo ello antes de que empiece la compilación propiamente dicha. El preprocesador de C++ reconoce los siguientes comandos:

```
#define, #undef
#if, #ifdef, #ifndef, #endif, #else, #elif
#include
#pragma
#error
#line
```

Los comandos más utilizados son: **#include**, **#define**.

9.1. Comando #include

Cuando en un archivo ***.cpp*** se encuentra una línea con un **#include** seguido de un nombre de archivo, el preprocesador la sustituye por el contenido de ese archivo.

La sintaxis de este comando es la siguiente:

```
#include "nombre_del_archivo"
#include <nombre_del_archivo>
```

La diferencia entre la primera forma (con comillas "...") y la segunda forma (con los símbolos <...>) estriba en el directorio de búsqueda de dichos archivos. En la forma con comillas se busca el archivo en el directorio actual y posteriormente en el directorio estándar de librerías (definido normalmente con una variable de entorno del MS-DOS llamada INCLUDE, en el caso de los compiladores de Microsoft). En la forma que utiliza los símbolos <...> se busca directamente en el directorio estándar de librerías. En la práctica los archivos del sistema (iostream.h, math.h, etc.) se incluyen con la segunda forma, mientras que los archivos hechos por el propio programador se incluyen con la primera.

Este comando del preprocesador se utiliza normalmente para incluir archivos con los prototipos (declaraciones) de las funciones de librería, o con módulos de programación y prototipos de las funciones del propio usuario. Estos archivos suelen tener la extensión **.h,** aunque puede incluirse cualquier tipo de archivo de texto.

9.2. Comando #define

El comando **#define** establece una ***macro*** en el código fuente. Existen dos posibilidades de definición:

```
#define NOMBRE texto_a_introducir
#define NOMBRE(parámetros) texto_a_introducir_con_parámetros
```

Antes de comenzar la compilación, el preprocesador analiza el programa y cada vez que encuentra el identificador NOMBRE lo sustituye por el texto que se especifica a continuación en el comando **#define**. Por ejemplo, si se tienen las siguientes líneas en el código fuente:

```
#define E 2.718281828459
...
void main(void) {
     double a;
     a= (1.+1./E)*(1.-2./E);
     ...
}
```

al terminar de actuar el preprocesador, se habrá realizado la sustitución de **E** por el valor indicado y el código habrá quedado de la siguiente forma:

```
void main(void) {
     double a;
     a= (1.+1./2.718281828459)*(1.-2./2.718281828459);
     ...
}
```

Este mecanismo de sustitución permite definir constantes simbólicas o valores numéricos (tales como **E**, **PI**, **SIZE**, etc.) y poder cambiarlas fácilmente, a la vez que el programa se mantiene más legible.

De la forma análoga se pueden definir ***macros con parámetros***: Por ejemplo, la siguiente macro calcula el cuadrado de cualquier variable o expresión,

```
#define CUAD(x) ((x)*(x))

void main() {
     double a, b;
     ...
     a = 7./CUAD(b+1.);
     ...
}
```

Después de pasar el preprocesador la línea correspondiente habrá quedado en la forma:

```
a = 7./((b+1.)*(b+1.));
```

Obsérvese que los paréntesis son necesarios para que el resultado sea el deseado, y que en el comando **#define** no hay que poner el carácter (;). Otro ejemplo de ***macro*** con dos parámetros puede ser el siguiente:

```
#define C_MAS_D(c,d) (c + d)

void main() {
     double a, r, s;
     a = C_MAS_D(s,r*s);
     ...
}
```

con lo que el resultado será:

```
     a = (s + r*s);
```

El resultado es correcto por la mayor prioridad del operador (*) respecto al operador (+). Cuando se define una *macro con argumentos* conviene ser muy cuidadoso para prever todos los posibles resultados que se pueden alcanzar, y garantizar que todos son correctos. En la definición de una *macro* pueden utilizarse *macros* definidas anteriormente. En muchas ocasiones, *las macros son*

más eficientes que las funciones, pues realizan una sustitución directa del código deseado, sin perder tiempo en copiar y pasar los valores de los argumentos.

Es recomendable tener presente que el comando **#define**:

– No define variables.

– Sus parámetros no son variables.

– En el preprocesamiento no se realiza una revisión de tipos, ni de sintaxis.

– Sólo se realizan sustituciones de código.

Por estas razones los posibles errores señalados por el compilador en una línea de código fuente con **#define** se deben analizar con las sustituciones ya realizadas. Por convención entre los programadores, ***los nombres de las macros se escriben con mayúsculas***.

Existen también muchas *macros predefinidas*. Algunas se muestran en la Tabla 9.1.

Tabla 9.1. Algunas macros predefinidas.

`__DATE__`	Fecha de compilación
`__FILE__`	Nombre del archivo
`__LINE__`	Número de línea
`__TIME__`	Hora de compilación

Se puede definir una *macro* sin *texto_a_sustituir* para utilizarla como ***señal*** a lo largo del programa. Por ejemplo:

```
#define COMP_HOLA
```

La utilidad de esta línea se observará en el siguiente apartado.

9.3. Comandos #ifdef, #ifndef, #else, #endif, #undef

Uno de los usos más frecuentes de las *macros* es para establecer bloques de compilación opcionales. Por ejemplo:

```
#define COMP_HOLA

void main() {
    /* si está definida la macro llamada COMP_HOLA */
    #ifdef COMP_HOLA
        cout << "hola";
    #else                       /* si no es así */
        cout << "adios";
    #endif
}
```

El código que se compilará será `cout <<"hola"` en caso de estar definida la *macro* `COMP_HOLA`; en caso contrario, se compilará la línea `cout <<"adios"`. Esta compilación condicional se utiliza con frecuencia para desarrollar *código portable* a varios distintos tipos de computadores; según de qué computador se trate, se compilan unas líneas u otras.

Para eliminar una *macro* definida previamente se utiliza el comando ***#undef***:

```
#undef COMP_HOLA
```

De forma similar, el comando **#ifndef** pregunta por la *no-definición* de la *macro* correspondiente.

10. LAS LIBRERÍAS DEL LENGUAJE C++

A continuación se incluyen algunas de las funciones de librería más utilizadas.

Función	Tipo	Propósito	lib
abs(i)	int	Retorna el valor absoluto de I	stdlib.h
acos(d)	double	Retorna el arco coseno de d	math.h
asin(d)	double	Retorna el arco seno de d	math.h
atan(d)	double	Retorna el arco targente de d	math.h
atof(s)	double	Convierte la cadena s en una cantidad de doble precisión	math.h
atoi(s)	long	Convierte la cadena s en un entero	stdlib.h
cos(d)	double	Retorna el coseno de d	math.h
exit(u)	void	Cerrar todos los archivos y buffers terminando el programa.	stdlib.h
exp(d)	double	Elevar e a la potencia d (e=2.77182...)	math.h
fabs(d)	double	Retorna el valor absoluto de d	math.h
fclose(f)	int	Cierra el archivo f.	stdio.h
feof(f)	int	Determina si se ha encontrado un fin de archivo.	stdio.h
fgetc(f)	int	Leer un carácter del archivo f.	stdio.h
fgets(s,i,f)	char *	Leer una cadena s, con I caracteres del archivo f	stdio.h
floor(d)	double	Retorna un valor redondeado por defecto al entero más cercano.	math.h
fmod(d1,d2)	double	Retorna el resto de d1/d2 (con el mismo signo de d1)	math.h
fopen(s1,s2)	file *	Abre un archivo llamado s1, del tipo s2. Retorna el puntero al archivo.	stdio.h
fputc(c,f)	int	Escribe un carácter en el archivo f.	stdio.h
getc(f)	int	Leer un carácter del archivo f	stdio.h
getchar()	int	Leer un carácter desde el dispositivo de entrada estándar.	stdio.h
log(d)	double	Retorna el logaritmo natural de d.	math.h
pow(d1,d2)	double	Retorna d1 elevado a la potencia d2.	math.h
rand(void)	int	Retorna un valor aleatorio positivo.	stdlib.h
sin(d)	double	Retorna el seno de d.	math.h
sqrt(d)	double	Retorna la raíz cuadrada de d.	math.h
srand(int)	void	Pone punto de inicio para rand().	
strcat(s1,s2)	char *	Añade s2 a s1.	string.h
strcmp(s1,s2)	int	Compara dos cadenas lexicográficamente.	string.h
strcomp(s1,s2)	int	Compara dos cadenas lexicográficamente, sin considerar mayúsculas o minúsculas.	string.h
strcpy(s1,s2)	char *	Copia la cadena s2 en la cadena s1	string.h
strlen(s1)	int	Retorna el número de caracteres en la cadena s.	string.h
system(s)	int	Pasa la orden s al sistema operativo.	stdlib.h
tan(d)	double	Retorna la tangente de d	math.h
time(p)	long int	Retorna el número de segundos transcurridos después de un tiempo base designado.	time.h
toupper(c)	int	convierte una letra a mayúscula	stdlib.h o ctype.h

Nota: La columna ***tipo*** se refiere al tipo de la cantidad devuelta por la función. Un asterisco denota puntero, y los argumentos que aparecen en la tabla tienen el significado siguiente:

c	denota un argumento de tipo carácter.	l	denota un argumento entero largo.
d	denota un argumento de doble precisión.	p	denota un argumento puntero.
f	denota un argumento archivo.	s	denota un argumento cadena.
i	denota un argumento entero.	u	argumento entero sin signo.

Si ha llegado hasta aquí, puede pasar al segundo manual: ***Programación Orientada a Objetos***