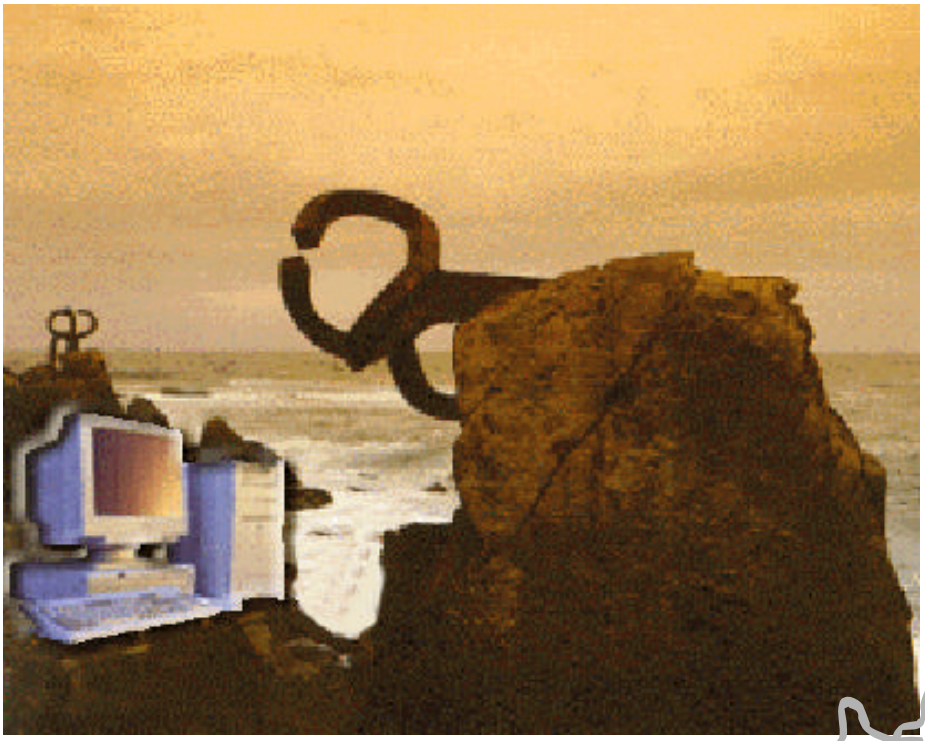
Escuela Superior de Ingenieros Industriales
de San Sebastián

UNIVERSIDAD DE NAVARRA

Aprenda Informática ...

Aprenda C++

como si estuviera en primero



San Sebastián, abril 1998

Javier García de Jalón • José Ignacio Rodríguez • José María Sarriegui • Alfonso Brazález

http://www.tayuda.com/ayudainf/index.htm

Aprenda C++

como si estuviera en primero

Javier García de Jalón
José Ignacio Rodríguez
José María Sarriegui
Alfonso Brazález

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

Editada y mantenida por Javier García de Jalón (jgjalon@etsii.upm.es)
Nuevos títulos y actualizaciones en: http://www.tayuda.com/ayudainf/index.htm

ÍNDICE

1 INTRODUCCIÓN

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++ tradicional a partir del lenguaje C. Su descripción se va a realizar en dos partes: una inicial en la que se contemplan las modificaciones y una posterior con los añadidos. El C++ 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 (las ***modificaciones menores***, que se verán a continuación). 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.

2 MODIFICACIONES MENORES

Como ya se ha dicho, el C++ contiene varias modificaciones menores sobre el C original. Normalmente se trata de aumentar la capacidad del lenguaje y la facilidad de programación en un conjunto de detalles concretos basados en la experiencia de muchos años. Como el ANSI C es posterior a los primeros compiladores de C++, algunas de estas modificaciones están ya introducidas en el ANSI C. En cualquier caso, se trata de modificaciones que facilitan el uso del lenguaje, pero que no cambian su naturaleza.

Hay que indicar que el C++ mantiene compatibilidad casi completa con C, de forma que el viejo estilo de hacer las cosas en C es también permitido en C++, aunque éste disponga de una mejor forma de realizar esas tareas.

2.1 Cambio en la extensión del nombre de los ficheros

El primer cambio que tiene que conocer cualquier programador es que los ficheros fuente de C++ tienen la extensión ***.cpp*** (de ***C plus plus***, que es la forma oral de llamar al lenguaje en inglés), en lugar de ***.c***. Esta distinción es muy importante, pues determina ni más ni menos el que se utilice el compilador de C o el de C++. La utilización de nombres incorrectos en los ficheros puede dar lugar a errores durante el proceso de compilación.

2.2 Comentarios introducidos en el programa

En C los comentarios empiezan por los caracteres **/*** y terminan con los caracteres ***/**. Pueden comprender varias líneas y estar distribuidos de cualquier forma, pero todo aquello que está entre el **/*** (inicio del comentario) y el ***/** (fin del comentario) es simplemente ignorado por el compilador. Algunos ejemplos de formato de comentarios son los siguientes:

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

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

/************************************
*  Esto es un comentario de varias    *
*  líneas, encerrado en una caja para *
*  llamar la atención.                *
************************************/
```

En C++ se admite el mismo tipo de comentarios que en C, pero además se considera que son comentarios todo aquel texto que está desde dos barras consecutivas (//) hasta el fin de la línea[2]. 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.
```

[2] El ANSI C permite el mismo tipo de comentarios que el C++, utilizando la doble barra **//**.

```
//***********************************
// Esto es un comentario de varias    *
// líneas, encerrado en una caja para *
// llamar la atención.                *
//***********************************
```

La ventaja de este nuevo método es que no se pueden comentar inadvertidamente varias líneas de un programa abriendo un indicador de comentario que no se cierre en el lugar adecuado.

2.3 Declaración simplificada de variables tipo enumeración

Las ***enumeraciones*** (variables ***enum***) permiten definir variables de tipo ***entero*** con un número pequeño de valores que están representados por identificadores alfanuméricos. Estos identificadores permiten que el programa se entienda más fácilmente, dando un significado a cada valor de la variable entera. Las variables tipo ***enum*** son adecuadas para representar de distintas formas valores binarios (SI o NO; VERDADERO o FALSO; EXITO o FRACASO, etc.), los días de la semana (LUNES, MARTES, MIERCOLES, ...), los meses del año (ENERO, FEBRERO, MARZO, ...), y cualquier conjunto análogo de posibles valores. En C las variables de tipo ***enum*** se hacían corresponder con enteros, y por tanto no hacían nada que no se pudiera hacer también con enteros. En C++ las variables ***enum*** son verdaderos *tipos de variables*, que necesitan un ***cast*** para que un valor entero les pueda ser asignado (ellas son promovidas a enteros cuando hace falta de modo automático). Esto quiere decir que si una función espera recibir como argumento un tipo ***enum*** sólo se le puede pasar un entero con un ***cast***. Por el contrario, si espera recibir un entero se le puede pasar un valor ***enum*** directamente.

La principal razón de ser de las variables ***enum*** es mejorar la claridad y facilidad de comprensión de los programas fuente.

Por ejemplo, si se desean representar los colores *rojo, verde, azul* y *amarillo* se podría definir un tipo de variable ***enum*** llamada ***color*** cuyos cuatro valores estarían representados por las constantes ROJO, VERDE, AZUL Y AMARILLO, respectivamente. Esto se puede hacer de la siguiente forma:

```
enum color {ROJO, VERDE, AZUL, AMARILLO};
```

Utilizar mayúsculas para los identificadores que representan constantes es una convención estilística ampliamente adoptada. En el ejemplo anterior se ha definido el tipo ***color***, pero no se ha creado todavía ninguna variable con ese tipo.

Por defecto los valores enteros asociados empiezan en 0 y van aumentando de uno en uno. Así, por defecto, los valores asociados serán:

```
ROJO = 0   VERDE = 1  AZUL = 2   AMARILLO = 3
```

Sin embargo, el programador puede asignar el valor que desee a cada uno de esos identificadores, asignando incluso el mismo entero a varios identificadores diferentes. por ejemplo, siguiendo con el tipo *color*:

```
enum color {ROJO = 3, VERDE = 5, AZUL = 7, AMARILLO};
```

Lógicamente en este caso los valores enteros asociados serán:

```
ROJO = 3   VERDE = 5  AZUL = 7   AMARILLO = 8
```

Cuando no se establece un entero determinado para un identificador dado, se toma el entero siguiente al anteriormente asignado. Por ejemplo, en el caso anterior al AMARILLO se le asigna un 8, que es el número siguiente al asignado al AZUL.

Una vez que se ha definido un tipo ***enum***, se pueden definir cuantas variables de ese tipo se desee. Esta definición es distinta en C y en C++. Por ejemplo, para definir las variables ***pintura*** y ***fondo***, de tipo ***color***, en C hay que utilizar la sentencia:

```
enum color pintura, fondo;              /* esto es C */
```

mientras que en C++ bastaría hacer:

```
color pintura, fondo;                   // esto es C++
```

Así pues en C++ no es necesario volver a utilizar la palabra ***enum***. Los valores que pueden tomar las variables ***pintura*** y ***fondo*** son los que puede tomar una variable del tipo ***color***, es decir: ROJO, VERDE, AZUL Y AMARILLO. Se puede utilizar, por ejemplo, la siguiente sentencia de asignación:

```
pintura = ROJO;
```

Hay que recordar que al imprimir una variable ***enum*** se imprime su valor entero y no su valor asociado[3].

2.4 Declaración simplificada de variables correspondientes a estructuras

De modo análogo a lo que pasa con la palabra clave ***enum***, en C++ no es necesario colocar la palabra clave ***struct*** para declarar una variable del tipo de una estructura definida por el usuario. Por ejemplo, si se define la estructura ***alumno*** del modo siguiente:

```
struct alumno {
    long nmat;
    char nombre[41];
};
```

en C++ se puede declarar después una variable ***delegado*** del tipo ***alumno*** simplemente con:

```
alumno delegado;                    // esto es C++
```

mientras que en C es necesario utilizar también la palabra ***struct*** en la forma:

```
struct alumno delegado;             /* esto es C */
```

2.5 Mayor flexibilidad en la declaración de variables

La declaración de variables en C++ es similar a la de C, pero con una importante diferencia. En ANSI C las variables tenían que ser declaradas (salvo que fueran ***extern***) *al comienzo de un bloque*, antes de la primera sentencia ejecutable de dicho bloque.

En C++ las variables pueden ser declaradas *en cualquier lugar de un bloque*[4]. Esto permite acercar la declaración de las variables al lugar en que se utilizan por primera vez. Las variables ***auto*** declaradas de esta forma existen desde el momento en que se declaran, hasta que se llega al fin del bloque correspondiente.

Un caso importante son los bucles ***for***. En C++ la variable que sirve de contador al bucle puede declararse e inicializarse en la propia sentencia ***for***. Por ejemplo, considérese el siguiente bucle para sumar los elementos de un vector:

---

[3] En C++ se podría conseguir que escribiera correctamente el tipo ***enum***, sobrecargando el operador << de modo adecuado, según se verá en secciones posteriores. La opción por defecto es que el tipo ***enum*** se promueve a entero y se imprime su valor.

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

```
for (double suma = 0.0, int i = 0; i<n; i++)
    suma += a[i];
```

donde las variables ***suma*** e ***i*** son declaradas y creadas como *double* e *int* en el momento de iniciarse la ejecución del bucle ***for***.

2.6 *Scope* o visibilidad de variables

La ***visibilidad*** de una variable es la parte del programa en la que esa variable está definida y puede ser utilizada. La ***duración*** hace referencia al tiempo que transcurre entre la creación de una variable y el instante en que es destruida. En general la visibilidad de una variable *auto* abarca desde el punto en el que se define hasta que finaliza el bloque en el que está definida. Si la declaración de una variable no se encuentra dentro de ningún bloque (variable *global* o *extern*), la visibilidad se extiende desde el punto de declaración hasta el final del fichero (otros ficheros pueden ver dicha variable sólo si la declaran como *extern*).

Las reglas de duración y visibilidad de C++ son similares a las de C. En C++ la visibilidad de una variable puede ser *local*, a nivel de *fichero* o a nivel de *clase*. Este último concepto, la clase, es la base de la ***Programación Orientada a Objetos*** y se estudiará detenidamente a partir del Capítulo 3.

Las variables *locales* se crean dentro de un bloque y sólo son visibles dentro del bloque en el que han sido definidas y en sus bloques anidados, salvo que sean ocultadas por una nueva variable del mismo nombre declarada en uno de esos bloques anidados.

Las variables que tienen visibilidad a nivel de *fichero* –variables *globales*– se definen fuera de cualquier bloque, función o clase.

Una variable *local* declarada dentro de un bloque oculta una variable *global* del mismo nombre u otra variable *local* también del mismo nombre declarada en un bloque más exterior. Por ejemplo, puede suceder que en un bloque, hasta la declaración de una variable ***x*** se pueda estar utilizando otra variable con el mismo nombre ***x*** de otro bloque que contenga al primero. A partir de su declaración y hasta el final de su bloque, la nueva variable ***x*** será la local del bloque más interior. Véase el ejemplo siguiente:

```
...
{
    double x = 2.0;
    printf("lf", x);                   // se imprime    2.0
    {
        printf("lf", x);               // se imprime    2.0
        double x = 3.0;
        printf("lf", x);               // se imprime    3.0
    }
    printf("lf", x);                   // se imprime    2.0
}
...
```

En C++ las variables definidas dentro de una *clase –variables miembro–* pueden ser declaradas como *privadas* o como *públicas*[5]. Las variables miembro que han sido declaradas como privadas no son visibles fuera de la clase; si se declaran como públicas se puede acceder a ellas mediante los operadores *punto* (`.`) y *flecha* (`->`), con las mismas reglas que para las variables miembro de las estructuras de C. Las funciones miembro de una clase tienen visibilidad directa sobre todas las variables miembro de esa clase, sin necesidad de que les sean pasadas como argumento.

---

[5] Más adelante se verá que existe una tercera forma de declarar las variables miembro: *protected*.

La ***duración*** (***lifetime***) de una variable es el período de tiempo en que esta variable existe durante la ejecución del programa. La duración de una variable puede ser ***automatic*** (opción por defecto) o ***static***. En el primer caso –el caso de las variables declaradas dentro de un bloque – la variable se crea y se destruye cada vez que se pasa por el bloque. Las variables ***static*** existen hasta que termina la ejecución del programa. Su valor se conserva entre las distintas pasadas por un bloque. Para que una variable local sea ***static*** hay que declararla como tal dentro del bloque.

Debe recordarse que aunque una variable exista durante toda la ejecución de un programa, sólo puede utilizarse en la zona del programa en que esa variable es visible.

C++ dispone del operador (::), llamado ***operador de resolución de visibilidad*** (***scope resolution operator***). Este operador, antepuesto al nombre de una ***variable global*** que está oculta por una variable local del mismo nombre, permite acceder al valor de la variable global[6]. Considérese el siguiente ejemplo:

```
int a = 2;                        // declaración de una variable global a

void main(void)
{
    ...
    printf("a = %d", a);          // se escribe  a = 2
    int a = 10;                   // declaración de una variable local a
    printf("a = %d", a);          // se escribe  a = 10
    printf("a = %d", ::a);        // se escribe  a = 2
}
```

El operador (::) no permite acceder a una variable local definida en un bloque más exterior oculta por otra variable local del mismo nombre. Este operador sólo permite acceder a una variable global oculta por una variable local del mismo nombre.

2.7 Especificador *const* para variables

En C++ el especificador ***const*** se puede utilizar con variables y con punteros[7]. 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.

Las variables ***const*** de C++ pueden ser utilizadas para definir el tamaño de un vector en la declaración de éste, cosa que no está permitida en C. Así las siguientes sentencias, que serían ilegales en C, son ahora aceptadas en C++:

---

[6] El operador (::) no puede utilizarse para ver una *variable local* oculta por otra *variable local* del mismo nombre.

[7] En ANSI C el especificador ***const*** también se puede utilizar con variables y con punteros, pero con estos últimos sólo de una de las dos formas posibles en C++.

```
void main(void)
{
    const int SIZE = 5;
    char  cs[SIZE] ;
}
```

De todas formas, nunca puede declararse ninguna variable *array* cuyo tamaño sea desconocido en tiempo de compilación. Si el tamaño de una variable va a ser conocido sólo en tiempo de ejecución, hay que utilizar reserva dinámica de memoria tanto en C como en C++.

Es muy frecuente que las funciones a las que por motivos de eficiencia (para no tener que sacar copias de los mismos) se les pasan los argumentos por referencia, éstos serán declarados como ***const*** en la definición y en el prototipo de la función, con objeto de hacer imposible una modificación accidental de dichos datos. Esto sucede por ejemplo con las funciones de manejo de cadenas de caracteres. El prototipo de la función ***strcpy()*** puede ser como sigue:

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

donde ***s1*** es la cadena copia y ***s2*** es la cadena original. Como no tiene sentido tratar de modificar la cadena original dentro de la función, ésta se declara como ***const***. En este caso el valor de retorno es un puntero a la cadena copia ***s1***.

2.8 Especificador *const* para punteros

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*** y
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.
```

En ANSI C una variable declarada como ***const*** puede ser modificada a través de un puntero a dicha variable. Por ejemplo, el siguiente programa compila y produce una salida i=3 con el compilador de C, pero da un mensaje de error con el compilador de C++[8]:

[8] En ambos casos se ha utilizado el compilador de Visual C/C++ de Microsoft. La única diferencia es que el fichero fuente termina en ****.c*** para el compilador de C y en ****.cpp*** para el compilador de C++.

```
#include <stdio.h>

void main(void)
{
    const int i = 2;
    int       *p;

    p = &i;
    *p = 3;
    printf("i = %d", i);
}
```

2.9 Conversiones explícitas de tipo

Además de las conversiones implícitas de tipo que tienen lugar al realizar operaciones aritméticas entre variables de distinto tipo –*promociones*– y en las sentencias de asignación, el lenguaje C dispone de una conversión explícita de tipo de variables, directamente controlada por el programador, llamada ***cast***. El ***cast*** se realiza anteponiendo al nombre de la variable o expresión el tipo al que se desea hacer la conversión encerrado entre paréntesis. Por ejemplo, pera devolver como ***int*** un cociente entre las variables ***double x*** e ***y***:

```
return (int) (x/y);
```

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);
return int(x/y);
```

2.10 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.

2.11 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 *original, const int longitud)
{
    strncpy(copia, original, longitud);
}


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

void main(void)
{
    string_copy(string_a, "Aquello");
    string_copy(string_b, "Esto es una cadena", 4);
    cout << string_b << " y " << string_a;
    // La última sentencia es equivalente a un printf() de C
    //   y se explica en un próximo apartado de este manual
}
```

2.12 Valores por defecto de parámetros de una función

En ANSI C se espera encontrar una correspondencia biunívoca entre la lista de *argumentos actuales* (llamada) y la lista de *argumentos formales* (declaración y definición) de una función. Por ejemplo, supóngase la siguiente declaración de una función para calcular el módulo de un vector ***x*** con ***n*** elementos:

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

En C esta función tiene que ser necesariamente llamada con dos argumentos actuales que se corresponden con los dos argumentos formales de la declaración.

En C++ la situación es diferente pues 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.

2.13 Variables de tipo *referencia*

A continuación se va a recordar brevemente cómo se pasaban argumentos *por referencia* en ANSI C. Para ello se va a utilizar la función ***permutar()***:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    int i = 1, j = 2;
    void permutar(int *a, int *b);

    printf("\ni = %d, j = %d", i, j);
    permutar(&i, &j);
    printf("\ni = %d, j = %d", i, j);
}

void permutar(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

La clave para pasar argumentos o parámetros por referencia en C está en el ***uso de punteros***. Al pasar la dirección de la variable[9], ésta es accesible desde dentro de la función y su valor puede ser modificado. De modo análogo, si dentro de una función hay que modificar un puntero habrá que pasar su dirección como argumento, esto es, habrá que pasar un puntero a puntero.

C++ ofrece una nueva 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 –que no existe en C– llamado tipo ***referencia (reference)***.

---

[9] En realidad se pasa una *copia* de la dirección de la variable

Las ***variables referencia*** se declaran por medio del carácter (&)[10]. 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 <stdio.h>
#include <stdlib.h>

void main(void)
{
    int i = 1, j = 2;
    void permutar(int &a, int &b);        // los argumentos son referencias

    printf("\ni = %d, j = %d", i, j);
    permutar(i, j);                      // los argumentos no llevan (*) ni (&)
    printf("\ni = %d, j = %d", i, j);
}

void permutar(int &a, int &b)             // los argumentos son referencias
{
    int 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.

---

[10] 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*** (*).

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.

2.14 Operadores *new* y *delete* para gestión dinámica de memoria

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.

Un aspecto diferente con la función ***malloc()***, que es el método más utilizado para reservar dinámicamente memoria en ANSI C, es que ésta devuelve un ***puntero a void*** (*void) que es después convertido al tipo de variable que se desea. Esa conversión se evita con ***new***, eliminando así una posible fuente de problemas.

Se puede utilizar el operador ***new*** para crear variables de cualquier tipo. ***New*** devuelve, en todos los casos, un puntero a la variable creada. También se pueden crear variables de tipos definidos por el usuario.

```
struct usuario {
    ..........
    };

usuario* Un_Usuario;
Un_Usuario = new usuario;
```

Cuando una variable ya no es necesaria se destruye con el operador ***delete*** para poder utilizar la memoria que estaba ocupando, mediante una instrucción del tipo:

```
delete p;
```

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=4, ncol=3, i, j;
    double **mat;

    // 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;

    }

}
```

2.15 Punteros de tipo *void*

Esta posibilidad ya está presente en el ANSI C pero conviene citarla aquí también. Un puntero a ***void*** es un puntero que no conoce en el momento de su definición a qué tipo de dato va a apuntar. Un buen ejemplo de esto es el valor de retorno de la función ***malloc().*** Esta función reserva memoria dinámicamente para cualquier tipo de dato, incluso para aquellos tipos de datos que haya definido el usuario.

2.16 Nueva forma de realizar las operaciones de entrada y salida.

En C++ además de las funciones ***printf()*** y ***scanf()***, que siguen estando vigentes, se pueden utilizar los operadores ***cin*** y ***cout***. Para utilizar estos nuevos operadores es necesario incluir la librería ***iostream.h*** con la instrucción ***#include <iostream.h>***. Así en un programa en C habría que hacer algo de este estilo:

```
char nombre;
int num=2;
printf ("Introduzca el nombre del fichero %d: ", num);
scanf (" %s", nombre)
```

En C++ podría escribirse así:

```
char nombre;
int num=2;
cout << "Introduzca el nombre del fichero " << num << ": ";
cin >> nombre;
```

Es importante darse cuenta de que ahora ya ***no hace falta especificar el tipo de dato*** que va a ser impreso o leído, asociándolo con un formato determinado. Es el propio programa el que decide el tipo de dato en tiempo de ejecución. Estos operadores están ***sobrecargados***[11] de tal manera que admiten tanto los tipos predefinidos como aquellos tipos de datos definidos por el usuario.

Para poder escribir o leer desde ficheros es necesario incluir la librería ***<fstream.h>***. A continuación se presenta un sencillo ejemplo en el que primero se escriben unas frases en un fichero y después se imprimen en la pantalla leídas desde ese fichero:

```
// Programa de lectura y escritura de ficheros

#include <fstream.h>
#include <stdlib.h>

void main(void)
{
    ofstream out("fichero.h");  // se define un flujo de salida a fichero

    out << "Estamos aprendiendo ";
    out << "como se escribe en un fichero ";
    out << "y como se lee desde el.";

    out.close();

    // Ahora comienza la lectura
    ifstream in("fichero.h");   // se define un flujo de entrada de fichero

    const int SIZE=81;
    char line[SIZE];

    in.getline(line, SIZE);
    cout << line << endl;

    in.getline(line, SIZE);
    cout << line << endl;

    in.getline(line, SIZE);
    cout << line << endl;
}
```

[11] C++ permite ***sobrecargar*** los operadores de modo similar a lo que se hace con las funciones: un mismo operador (+, -, *, /, =, ==, !=, <<, >>, etc.) tiene distinto significado según la naturaleza de los operandos a los que se aplica.

El ejemplo anterior necesita algunas explicaciones extras, aunque más adelante se incluye un capítulo dedicado a las entradas y salidas de datos en C++.

- Se declara un ***objeto***[12] del tipo ***ofstream*** llamado ***out***. Este será un objeto que almacenará la información necesaria para llevar los datos de salida hasta un fichero llamado ***fichero.h***. Esto es el equivalente a utilizar la función ***fopen()*** de ANSI C para abrir un fichero de escritura de datos.
- Hay que darse cuenta de que la primera vez que se abre el fichero se abre en modo de *escritura*. Por eso hay que cerrarlo para después poder abrirlo en modo de *lectura*. Para realizar esta última operación es necesario declarar un objeto, al que se llama ***in***, del tipo ***ifstream***. El ***endl*** del final de las líneas de impresión en pantalla hace que se imprima un caracter de salto de línea y que se vacíe el buffer de salida de modo inmediato.
- La función ***getline*** se utiliza para leer los datos que se introduzcan desde el teclado de forma similar a la que lo haría la función ***scanf()*** leyendo una línea completa hasta el '\n'.

2.17 Funciones con número de parámetros variable

Se pueden definir, tanto en ANSI C como en C++, funciones con un número variable y desconocido a priori de argumentos. Un ejemplo de función de este tipo es la propia función ***main()*** con argumentos, y otros ejemplos, de sobra conocidos, son las funciones ***printf()*** y ***scanf()***.

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.

---

[12] El concepto de ***objeto***, básico en C++, se explica con todo detalle en el Capítulo 3.

3 MODIFICACIONES MAYORES

3.1 Introducción a la Programación Orientada a Objetos (OOP)

La ***Programación Orientada a Objetos*** (POO) permite realizar grandes programas mediante la unión de elementos más simples, que pueden ser diseñados y comprobados de manera independiente del programa que va a usarlos. Muchos de estos elementos podrán ser reutilizados en otros programas.

A estas “piezas”, “módulos” o "componentes", que interactúan entre sí cuando se ejecuta un programa, se les denomina ***objetos***. Estos ***objetos*** contienen tanto ***datos*** como las ***funciones*** que actúan sobre esos datos.

De ordinario, cada uno de estos ***objetos*** corresponde a algún elemento que debe utilizar el programa. Algunos de estos elementos representan entidades del mundo real (matrices, personas, cuentas de banco, elementos mecánicos o eléctricos, ...) y otros pueden ser componentes del ordenador (tanto de software como de hardware: otro programa, un fichero de disco, una impresora conectada en una puerta serie, una ventana abierta en la pantalla, ...). También pueden ser estructuras de datos: colas, pilas, ...

Durante la ejecución del programa, los ***objetos*** interactúan pasándose ***mensajes*** y ***respuestas***. Es fundamental darse cuenta de que un ***objeto*** no necesita conocer el funcionamiento interno de los demás objetos para poder interactuar con ellos (igual que el hombre no necesita conocer cómo funciona por dentro un televisor o un ordenador para poder utilizarlos), sino que le es suficiente con saber la forma en que debe enviarle sus mensajes y cómo va a recibir la respuesta (al hombre le puede ser suficiente con saber cómo funcionan el interruptor, el dial del volumen y los botones de cambio de canal para utilizar un televisor).

Sucede a menudo que hay que utilizar varios ejemplares análogos de un determinado elemento u objeto (por ejemplo varias ventanas en la pantalla del PC, varios usuarios, varios clientes, varias cuentas corrientes de un banco, etc.). La definición genérica de estos ***objetos*** análogos se realizar mediante la ***clase***. Así, una ***clase*** contiene una completa y detallada descripción de la información y las funciones que contendrá cada ***objeto*** de esa clase. Las ***clases*** de C++ se pueden ver como una generalización de las ***estructuras*** de C. Por ejemplo, en el siguiente código,

```
struct Alumno {
    long nmat;
    char nombre[41];
};

Alumno alu1={76986, "Luis Perez"}, alu2 = { 67549, "Mikel Lasa"};
```

se definen las *variables miembro* que va a tener la estructura ***Alumno*** y luego se crean dos variables ***alu1*** y ***alu2*** de esa estructura. En C++ los ***objetos*** son las variables concretas que se crean de una determinada ***clase***. A veces se llaman también ***instances*** o ***data objects***. Las ***clases*** de C++ son como una generalización de las ***estructuras*** de C.

En C++ las ***clases*** son verdaderos tipos de datos definidos por el usuario y pueden ser utilizados de igual manera que los ***tipos de datos*** propios del C++, tales como ***int*** o ***float***. Los ***objetos*** son a las ***clases*** como las ***variables*** a los ***tipos de variables***. Un ***objeto*** tiene su propio conjunto de ***datos o variables miembro***, aunque no de funciones, que aunque se aplican a un objeto concreto son propias de la ***clase*** a la que pertenece el ***objeto***.

3.2 Clases, Objetos y Métodos.

En ANSI C las ***funciones*** son algo relativamente independiente de las ***variables***, y constituyen el centro del lenguaje. Se dice por eso que C es un lenguaje ***algorítmico*** (o ***procedural***, en inglés). Cualquier ***función*** se puede comunicar con las demás a través de ***variables globales***, del ***valor de retorno*** y de los ***argumentos***, pasados *por valor* o *por referencia*. Esta facilidad para comunicarse con otras funciones hace que se puedan producir efectos laterales no deseados.

En un ***Lenguaje Orientado a Objetos*** tal como el C++, el centro del lenguaje no son las funciones sino los ***datos***, o más bien los ***objetos***, que contienen ***datos*** y ***funciones*** concretas que permiten manipularlos y trabajar sobre ellos. Esto hace que la mentalidad con la que se aborda la realización de un programa tenga que ser muy diferente.

Para proteger a las variables de modificaciones no deseadas se introduce el concepto de ***encapsulación***, *ocultamiento* o *abstracción de datos*. Los miembros de una clase se pueden dividir en ***públicos*** y ***privados***. Los miembros ***públicos*** son aquellos a los que se puede acceder libremente desde fuera de la clase. Los miembros ***privados***, por el contrario, solamente pueden ser accedidos por los métodos de la propia clase.

De ordinario una clase ofrece un conjunto de ***funciones públicas*** a través de las cuales se puede actuar sobre los ***datos***, que serán ***privados***. Estas funciones o ***métodos públicos*** constituyen la ***interface*** de la clase. De esta forma se garantiza que se hace buen uso de los objetos, manteniendo la coherencia de la información. Esto sería imposible si se accediera libre e independientemente a cada variable miembro. Al usuario le es suficiente con saber cómo comunicarse con un objeto, pero no tiene por qué conocer el funcionamiento interno del mismo. En C++ los ***métodos*** de una clase pueden ser ***funciones*** u ***operadores***. Todo esto se estudiará en detalle más adelante.

Ya se ha hablado de las ***funciones sobrecargadas***, que son funciones con el mismo nombre pero con distintos argumentos y definición. Otra posibilidad interesante es la de que objetos de distintas clases respondan de manera análoga al aplicarles funciones con idéntico nombre y argumentos. Esta posibilidad da origen a las ***funciones virtuales*** y al ***polimorfismo***, diferente de las funciones sobrecargadas, al que se dedicará un capítulo completo de este manual, ya que es una de las capacidades más importantes del C++.

3.3 Ejemplo de clase en C++: números complejos

Antes de entrar en las explicaciones más detalladas de la *Programación Orientada a Objetos* según el lenguaje C++, se va a presentar un ejemplo relativamente sencillo. No importa que ahora no se entienda este ejemplo en todos sus detalles: lo importante es ver de un modo general las posibilidades que C++ ofrece y el grado de complejidad de sus soluciones. En una primera etapa, la lectura de este apartado podría omitirse. Los programas de este ejemplo tienen las líneas numeradas. Esto no tiene nada que ver con C++; se ha hecho con ***Word*** al objeto de poder hacer referencia con más facilidad al código del programa.

El presente ejemplo define la clase ***complejo***, que permite trabajar con números complejos de una forma muy sencilla y natural. El fichero ***complejo.h*** contiene la definición de la clase; el fichero ***complejo.cpp*** contiene la definición de las funciones y operadores de la clase ***complejo***. Finalmente el fichero ***main.cpp*** contiene un programa principal que utiliza algunas de las posibilidades de esta clase. Esta organización de ficheros es bastante habitual en C++. Todos los usuarios de la clase tienen acceso al fichero ***header*** donde se define la clase, en este caso ***complejo.h***. La definición de las funciones y operadores (implementación de la clase) se hace en otro fichero fuente al cual ya no

es necesario acceder. De ordinario basta acceder al resultado de compilar ***complejo.cpp***, pero los detalles de este código fuente pueden quedar ocultos al usuario de la clase ***complejo***.

El contenido del fichero ***complejo.h*** es como sigue:

```
1.    // fichero complejo.h
2.    // declaración de la clase complejo

3.    #ifndef __COMPLEJO_H__
4.    #define __COMPLEJO_H__

5.    #include <iostream.h>

6.    class complejo
7.    {
8.    private:
9.        double real;
10.       double imag;
11.   public:
12.       // Constructores
13.       complejo(void);
14.       complejo(double, double im=0.0);
15.       complejo(const complejo&);
16.       // SetThings
17.       void SetData(void);
18.       void SetReal(double);
19.       void SetImag(double);
20.       // GetThings
21.       double GetReal(void){return real;}
22.       double GetImag(void){return imag;}
23.       // Sobrecarga de operadores aritméticos
24.       complejo operator+ (const complejo&);
25.       complejo operator- (const complejo&);
26.       complejo operator* (const complejo&);
27.       complejo operator/ (const complejo&);
28.       // Sobrecarga del operador de asignación
29.       complejo& operator= (const complejo&);
30.       // Sobrecarga de operadores de comparación
31.       friend int operator== (const complejo&, const complejo&);
32.       friend int operator!= (const complejo&, const complejo&);
33.       // Sobrecarga del operador de inserción en el flujo de salida
34.       friend ostream& operator<< (ostream&, const complejo&);
35.   };

36.   #endif
```

En los párrafos que siguen se comenta brevemente el contenido del fichero ***complejo.h***:

- La definición de la clase ***complejo*** se ha introducido dentro de una bifurcación ***#ifndef ... #endif*** del preprocesador de C++, con objeto de prevenir una inclusión múltiple en un fichero fuente.
- La clase ***complejo*** tiene dos variabes miembro ***privadas*** tipo ***double***, llamadas ***real*** e ***imag***, que representan la parte real e imaginaria del número complejo (líneas 9-10). Al ser privadas los usuarios de la clase no podrán acceder directamente a ellas por medio de los operadores ***punto*** (.) y ***flecha*** (->), como se hace con las estructuras de C.
- Las líneas 11-34 contienen la declaración de un conjunto de ***funciones y operadores miembro*** de la clase ***complejo***. Todas ellas han sido declaradas en la sección ***public:*** de la clase, por lo que se podrán utilizar desde fuera de la clase sin restricciones. Ya se ve una diferencia importante con las estructuras de C: las clases de C++, además de contener ***variables miembro***, pueden también definir ***funciones y operadores miembro*** que trabajarán con las variables miembro –datos- de la clase.

- Las tres primeras funciones miembro (líneas 13-15) son los ***constructores*** de la clase. Los constructores tienen el mismo nombre que la clase y no tienen valor de retorno, ni siquiera ***void***. Los constructores se llaman de modo automático cada vez que se crea un objeto de la clase (en este caso, cada vez que se cree un número complejo) y su misión es que todas las variables miembro de cualquier objeto estén siempre correctamente inicializadas. La línea 15 contiene la declaración del ***constructor de copia***, que se llama cuando hay que crear un objeto a partir de otro objeto de la misma clase.
- El siguiente grupo de funciones miembro (líneas 17-19) permite ***dar valor*** a las variables miembro ***real*** e ***imag***, que por ser privadas, no son accesibles directamente. Es habitual que el nombre de este tipo de funciones empiece por la palabra inglesa ***set***.
- Las funciones miembro 21-22 permiten ***acceder*** a las variables miembro ***privadas***. Este tipo de funciones suelen empezar con la palabra inglesa ***get***. En este caso, junto con la ***declaración*** de la función se ha incluido su ***definición***. Obsérvese que estas funciones acceden a las variables miembro ***real*** e ***imag*** directamente, sin necesidad de pasárselas como argumento. Ésta es una característica de todas las funciones miembro de una clase.
- Las líneas 24-27 contienen la declaración de los 4 operadores aritméticos como ***operadores miembro*** de la clase. Los operadores son análogos a las funciones sustituyendo el ***nombre de la función*** por la palabra ***operator*** seguida del carácter o caracteres del operador de C++ que se desea sobrecargar. Al dar una nueva definición para los operadores suma (+), resta (-), multiplicación (*) y división (/) acorde con la aritmética de números complejos, se podrán introducir expresiones aritméticas para números complejos enteramente análogas a las de variables *int*, *long*, *float* o *double*. Los cuatro operadores tienen un valor de retorno ***complejo***, pero sólo tienen un argumento que es una referencia constante a ***complejo***. Dado que estos cuatro operadores son ***binarios***, ¿dónde está el otro operando? La respuesta es que en una expresión del tipo ***a+b***, donde tanto ***a*** como ***b*** son números complejos, el primer operando ***a*** es un ***argumento implícito*** (a cuyas variables miembro ***real*** e ***imag*** se accede directamente), mientras que el segundo operando debe ser pasado como ***argumento explícito*** al operador, y se accederá a sus variables miembro a través del ***nombre*** y el ***operador punto*** (.). Esto quedará más claro al ver la definición de estos operadores en el fichero ***complejo.cpp***.
- La línea 29 declara el operador de asignación (=) sobrecargado. En este caso el ***argumento implícito*** es el miembro izquierdo de la igualdad, mientras que el miembro derecho es el argumento implícito. El ***valor de retorno*** es una ***referencia a complejo*** para poder escribir expresiones del tipo ***a=b=c;*** que son legales en C/C++.
- Las líneas 31 y 32 definen los operadores de comparación (==) y (¡=). Estos operadores no se han definido como ***operadores miembro*** de la clase, sino como ***operadores friend***. Los operadores friend no tienen un objeto de la clase como argumento implícito y por ello hay que pasarles los dos argumentos de modo ***explícito***.
- Finalmente, en la línea 34 se declara el operador de ***inserción en el flujo de salida*** (<<), que permitirá imprimir números complejos de forma análoga a lo que se hace con *doubles* o con *cadenas de caracteres*. Es también un operador ***friend***. El primer argumento es una ***referencia al flujo de salida*** y el segundo una ***referencia const al complejo*** a imprimir. El ***valor de retorno*** es la referencia al flujo de salida con el número ***complejo*** ya añadido con el ***formato adecuado***.

A continuación se muestra el contenido del fichero ***complejo.cpp***, que contiene la definición de las funciones ***miembro*** y ***friend*** declaradas en ***complejo.h***. No se van a dar unas explicaciones

tan pormenorizadas como en el caso anterior, pero no es difícil que en una segunda lectura (después de haber leído los siguientes capítulos) las cosas queden bastante claras en la mente del lector.

```
37.   // fichero complejo.h
38.   // funciones y operadores de la clase complejo

39.   #include "complejo.h"

40.   // constructor por defecto
41.   complejo::complejo(void)
42.   {
43.       real = 0.0;
44.       imag = 0.0;
45.   }

46.   // constructor general
47.   complejo::complejo(double re, double im)
48.   {
49.       real = re;
50.       imag = im;
51.   }

52.   // constructor de copia
53.   complejo::complejo(const complejo& c)
54.   {
55.       real = c.real;
56.       imag = c.imag;
57.   }

58.   // función miembro SetData()
59.   void complejo::SetData(void)
60.   {
61.       cout << "Introduzca el valor real del complejo: ";
62.       cin  >> real;
63.       cout << "Introduzca el valor imaginario del complejo: ";
64.       cin  >> imag;
65.   }

66.   void complejo::SetReal(double re)
67.   {
68.       real = re;
69.   }

70.   void complejo::SetImag(double im)
71.   {
72.       imag = im;
73.   }

74.   // operador miembro + sobrecargado
75.   complejo complejo::operator+ (const complejo &a)
76.   {
77.       complejo suma;
78.       suma.real = real + a.real;
79.       suma.imag = imag + a.imag;
80.       return suma;
81.   }

82.   // operador miembro - sobrecargado
83.   complejo complejo::operator- (const complejo &a)
84.   {
85.       complejo resta;
86.       resta.real = real - a.real;
87.       resta.imag = imag - a.imag;
88.       return resta;
89.   }
```

```
90.   // operador miembro * sobrecargado
91.   complejo complejo::operator* (const complejo &a)
92.   {
93.       complejo producto;
94.       producto.real = real*a.real - imag*a.imag;
95.       producto.imag = real*a.imag + a.real*imag;
96.       return producto;
97.   }

98.   // operador miembro / sobrecargado
99.   complejo complejo::operator/ (const complejo &a)
100.  {
101.      complejo cociente;
102.      double d = a.real*a.real + a.imag*a.imag;
103.      cociente.real = (real*a.real + imag*a.imag)/d;
104.      cociente.imag = (-real*a.imag + imag*a.real)/d;
105.      return cociente;
106.  }

107.  // operador miembro de asignación sobrecargado
108.  complejo& complejo::operator= (const complejo &a)
109.  {
110.      real = a.real;
111.      imag = a.imag;
112.      return (*this);
113.  }

114.  // operador friend de test de igualdad sobrecargado
115.  int operator== (const complejo& a, const complejo& b)
116.  {
117.      if (a.real==b.real && a.imag==b.imag)
118.          return 1;
119.      else
120.          return 0;
121.  }

122.  // operador friend de test de desigualdad sobrecargado
123.  int operator!= (const complejo& a, const complejo& b)
124.  {
125.      if (a.real!=b.real || a.imag!=b.imag)
126.          return 1;
127.      else
128.          return 0;
129.  }

130.  // operador friend << sobrecargado
131.  ostream& operator << (ostream& co, const complejo &a)
132.  {
133.      co << a.real;
134.      long fl = co.setf(ios::showpos);
135.      co << a.imag << "i";
136.      co.flags(fl);
137.      return co;
138.  }
```

A propósito de las funciones anteriores se pueden hacer los comentarios siguientes:

- En C++ es habitual distinguir entre la ***declaración*** de una clase (fichero ***complejo.h***) y su ***implementación*** (fichero ***complejo.cpp***). De ordinario sólo la declaración es pública, quedando oculta a los usuarios de la clase la forma en la que se han programado las distintas funciones y operadores miembro.
- Las ***funciones y operadores miembro*** de una clase se definen anteponiendo a su nombre el nombre de la clase y el ***scope resolution operator (::)***.

- *Las funciones y operadores miembro acceden directamente a las variables miembro del objeto implícito, y por medio del nombre y del operador punto (.) a las del objeto pasado explícitamente*. Por ejemplo, supóngase la operación (***x+y;***), donde tanto ***x*** como ***y*** son complejos. Para hacer esta operación se utilizará el ***operador + sobrecargado***, definido en las líneas 75-81. En este caso ***x*** es el ***argumento implícito***, mientras que el objeto ***y*** se pasa ***explícitamete*** por ventana. Dentro de la función que define el ***operador +***, las *partes real e imaginaria* de ***x*** se acceden como ***real*** e ***imag***, mientras que las de ***y*** se acceden como ***a.real*** y ***a.imag***, pues el *argumento actual* ***y*** se recibe como *argumento formal* ***a***.
- Las ***funciones y operadores friend*** no pertenecen a la clase y por tanto no llevan el nombre de la clase y el ***scope resolution operator (::)***.. Estas funciones no tienen argumento implícito y todos los argumentos deben pasar por ventana.
- En la sobrecarga del operador miembro de asignación (=) (líneas 108-113) aparece la sentencia (***return (*this);***). ¿Qué representa esto? La idea es que las funciones y operadores miembro acceden directamente a las variables miembro del objeto que es su argumento implícito, pero sólo con esto carecerían de una visión de conjunto de dicho argumento. Por ejemplo, en la sentencia (x=y;) el ***operador =*** recibe ***y*** como argumento formal ***a***, y aunque puede acceder a las variables ***real*** e ***imag*** de ***x*** no puede acceder a ***x*** como objeto. Esto se soluciona en C++ con el puntero ***this***, que es siempre ***un puntero al argumento implícito*** de cualquier función u operador miembro de una clase. De acuerdo con esto, ****this*** será el argumento implícito ***x*** y ese es su significado.

Finalmente se muestra el programa principal ***main.cpp*** y la salida por pantalla que origina.

```
139.  // fichero main.cpp

140.  #include "complejo.h"

141.  void main(void)
142.  {

          // se crean dos complejos con el constructor general
143.      complejo c1(1.0, 1.0);
144.      complejo c2(2.0, 2.0);
          // se crea un complejo con el constructor por defecto
145.      complejo c3;
          // se da valor a la parte real e imaginaria de c3
146.      c3.SetReal(5.0);
147.      c3.SetImag(2.0);
          // se crea un complejo con el valor por defecto (0.0) del 2º argumento
148.      complejo c4(4.0);

          // se crea un complejo a partir del resultado de una expresión
          // se utiliza el constructor de copia
149.      complejo suma = c1 + c2;
          // se crean tres complejos con el constructor por defecto
150.      complejo resta, producto, cociente;
          // se asignan valores con los operadores sobrecargados
151.      resta = c1 - c2;
152.      producto = c1 * c2;
153.      cociente = c1 / c2;

          // se imprimen los números complejos con el operador << sobrecargado
154.      cout << c1 << ", " << c2 << ", " << c3 << ", " << c4 << endl;
155.      cout << "Primer complejo: " << c1 << endl;
156.      cout << "Segundo complejo: " << c2 << endl;
157.      cout << "Suma: " << suma << endl;
158.      cout << "Resta: " << resta << endl;
159.      cout << "Producto: " << producto << endl;
160.      cout << "Cociente: " << cociente << endl;
```

```
          // se comparan complejos con los operadores == y != sobrecargados
161.      if (c1==c2)
162.          cout << "Los complejos son iguales." << endl;
163.      else
164.          cout << "Los complejos no son iguales." << endl;

165.      if (c1!=c2)
166.          cout << "Los complejos son diferentes." << endl;
167.      else
168.          cout << "Los complejos no son diferentes." << endl;

169.      cout << "Ya he terminado." << endl;
170.  }
```

La salida por pantalla, basada en los números complejos definidos en el programa principal, es la siguiente:

```
1+1i, 2+2i, 0+0i, 4+0i
Primer complejo: 1+1i
Segundo complejo: 2+2i
Suma: 3+3i
Resta: -1-1i
Producto: 0+4i
Cociente: 0.5+0i
Los complejos no son iguales.
Los complejos son diferentes.
Ya he terminado.
```

El programa principal incluye suficientes comentarios como para entender fácilmente su funcionamiento. Puede observarse la forma tan natural con la que se opera y se imprimen estos números complejos, de un modo completamente análogo a lo que se hace con las variables estándar de C++.

3.4 Clase sin secciones privadas: *struct*

Una ***clase sin secciones privadas*** es aquella en la que no se aplica la ***encapsulación***. Esta posibilidad no es muy interesante en la programación habitual, pero puede servir de introducción a las ***clases***.

Una estructura sencilla podría ser la siguiente (en este caso C y C++ coinciden):

```
struct C_Cuenta {
    double Saldo;
    double Interes;
};
```

A los ***datos*** declarados dentro de una ***estructura*** se les da el nombre de ***variables miembro***, ***datos miembro***, o simplemente ***miembros*** de la estructura. La estructura del ejemplo anterior tiene dos miembros: ***Saldo*** e ***Interes***. La visibilidad de estos dos miembros es el bloque encerrado entre las llaves, pero al igual que en C se puede acceder a ellos desde fuera del bloque por medio de los operadores ***punto*** (.) y ***flecha*** (->).

Para declarar objetos de la estructura ***C_Cuenta,*** en C habría que hacer lo siguiente:

```
struct C_Cuenta c1, c2;
```

pero al programar en C++ es suficiente con escribir:

```
C_Cuenta c1, c2;
```

ya que en C++ una ***estructura o clase definida por el usuario*** es como un ***tipo de variable***.

Para acceder a cada una de las ***variables miembro*** de la clase se utiliza el operador ***dot*** o ***punto*** (.), o bien el operador ***arrow*** o ***flecha*** (->). Así ***c1.Saldo*** se refiere a la variable ***Saldo*** de ***c1,*** y ***c2.Interes*** se refiere a la variable ***Interes*** de la ***c2***. Este operador tiene precedencia sobre casi todos los demás operadores. De forma análoga, el operador ***flecha*** (->) se utiliza cuando se dispone de la dirección de un objeto (en el puntero correspondiente), en lugar del nombre del objeto. Como se verá más adelante, en C++ es mucho más habitual utilizar el operador ***flecha*** que el operador ***punto***.

Ya se ha dicho que la principal característica de las ***clases de C++*** era que agrupan datos y funciones. En C++ las ***estructuras*** definidas como en el ejemplo anterior son verdaderas ***clases***. Se pueden añadir algunas funciones a la clase anterior para que sea un caso más real. Estas funciones serán los ***métodos*** que permitirán interactuar con las variables de esa clase. Así la clase ***C_Cuenta*** puede quedar de esta manera:

```
struct C_Cuenta {
    // Variables miembro[13]
    double Saldo;          // Saldo Actual de la cuenta
    double Interes;        // Interés aplicado
    // Métodos[14]
    double GetSaldo();
    double GetInteres();
    void SetSaldo(double unSaldo);
    void SetInteres(double unInteres);
};
```

Las definiciones de esas funciones o métodos podrían ser como sigue:

```
double C_Cuenta::GetSaldo()
    { return Saldo; }        // Se obtiene el valor de la variable Saldo

double C_Cuenta::GetInteres()
    { return Interes;        // Se obtiene el valor de la variable Interes

void C_Cuenta::SetSaldo(double unSaldo)
    { Saldo = unSaldo; }     // Se asigna un valor a la variable Saldo

void C_Cuenta::SetInteres(double unInteres)
    { Interes = unInteres;   // Se asigna un valor a la variable Interes
```

Se puede adelantar ya que las funciones que se acaban de definir van a resultar muy útiles para acceder a la variables miembro de una clase, salvaguardando el principio de ***encapsulación***.

El ***operador (::)*** recibe el nombre de ***operador de resolución de visibilidad*** (***scope resolution operator***). La notación ***double C_Cuenta::GetSaldo()*** indica que se está haciendo referencia a la función ***GetSaldo*** definida como ***función miembro*** en la clase ***C_Cuenta***. Mediante esta notación se puede distinguir entre funciones que tengan el mismo nombre pero distintas visibilidades y permite también acceder a funciones desde puntos del programa en que éstas no son visibles.

La ***definición*** de las funciones miembro puede estar incluida en la definición de la propia clase, en cuyo caso la clase quedaría como se muestra a continuación:

---

[13] A lo largo de este manual se utilizará la letra negrita para remarcar los nuevos aspectos que se vayan añadiendo al ejemplo.

[14] Es habitual llamar con la palabra inglesa ***Get*** a las funciones que devuelven el valor de una variable y con la palabra ***Set*** a las que permiten cambiar el valor de esa variable.

```
struct C_Cuenta {
    // Variables miembro
    double Saldo;          // Saldo Actual de la cuenta
    double Interes;        // Interés aplicado
    // Métodos

    double C_Cuenta::GetSaldo()
        { return Saldo; }

    double C_Cuenta::GetInteres()
        { return Interes;

    void C_Cuenta::SetSaldo(double unSaldo)
        { Saldo = unSaldo; }

    void C_Cuenta::SetInteres(double unInteres)
        { Interes = unInteres;
};

void main(void) {
    C_Cuenta c1;
    c1.Interes = 4.0;    // válida, pero se viola el principio de encapsulación
    c1.SetInteres(4.0);  // correcto
}
```

En el ejemplo anterior aparece ya un pequeño programa principal que utiliza la clase que se acaba de definir.

Se debe distinguir entre los operadores (.) y el (::). El operador ***punto*** (.) se utiliza para acceder a una variable o función miembro a partir del nombre de un determinado objeto, mientras que el operador ***scope resolution*** (::) sirve para designar a un miembro (variable o función) de una clase en su definición. Recuérdese que el operador (::) permite también hacer referencia a una variable global desde dentro de una función de una clase que contenga una variable miembro con el mismo nombre.

Se puede ver que una llamada a la función ***SetInteres()*** en la forma:

```
c1.SetInteres(100.0);
```

es equivalente a la sentencia:

```
c1.Interes = 100.0;
```

mientras que una llamada a la función ***GetSaldo()*** en la forma:

```
cash = c2.GetSaldo();
```

es equivalente a la sentencia:

```
cash = c2.Saldo;
```

Esta última forma de acceder a una variable miembro de una clase atenta contra el ***principio de encapsulación***, que es uno de los objetivos más importantes de la programación orientada a objetos.

Un usuario de una clase sólo necesita conocer el ***interface***, es decir, el aspecto externo de la clase, para poder utilizarla correctamente. En otras palabras, le sería suficiente con conocer la declaración de las funciones miembro públicas y el significado de los argumentos, además de las variables miembro públicas, si las hubiera.

3.5 Clases con secciones privadas.

Las declaraciones de variables y funciones miembro de la clase ***C_Cuenta*** del siguiente ejemplo están divididas en dos grupos, encabezados respectivamente por las palabras ***private:*** y ***public:***.

Habitualmente las variables miembro suelen ser privadas y algunas funciones miembro públicas, pero ya se verán todo tipo de combinaciones posibles.

```
class C_Cuenta {
    // Variables miembro
    private:
        char *Nombre;      // Nombre de la persona
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado

    public:
        // Métodos
        char *GetNombre()
            { return Nombre; }
        double GetSaldo()
            { return Saldo; }
        double GetInteres()
            { return Interes; }
        void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
};
```

Una primera diferencia respecto a los ejemplos anteriores es que se ha sustituido la palabra ***struct*** (bien conocida en C) por la palabra ***class***, que es propia de C++. Ambas palabras pueden ser utilizadas en la definición de las clases de C++. Enseguida se verá la pequeña diferencia que existe entre utilizar una palabra u otra.

La palabra ***private*** precediendo a la declaración de las *variables o funciones miembro* indica al compilador que el acceso a dichos miembros sólo está permitido a las funciones miembro de la clase, de forma que cualquier otra función o el programa principal, tienen prohibido acceder a ellas mediante los ***operadores punto*** (`.`) y ***flecha*** (`->`), y de cualquier otra forma que no sea a través de las funciones miembro de la clase que sean ***public***.

La palabra ***public*** precediendo a la declaración de las *funciones o variables miembro* indica que dichas funciones podrán ser llamadas desde fuera de la clase mediante el procedimiento habitual en C para acceder a los miembros de las estructuras, es decir mediante los ***operadores punto*** (`.`) y ***flecha*** (`->`).

No es necesario poner las dos palabras ***public*** y ***private***. Se tienen las dos opciones siguientes:

1. Definiendo la clase con la palabra ***struct*** la opción por defecto es ***public***, de modo que todas las variables y funciones miembro son ***public*** excepto las indicadas expresamente como ***private***.
2. Por el contrario, utilizando la palabra ***class*** la opción por defecto es ***private***, de modo que todas las variables y funciones miembro son ***private*** excepto las indicadas expresamente como ***public***.

Si se incluyen las dos palabras -***public*** y ***private***- es igual utilizar ***struct*** o ***class***. En lo sucesivo se utilizará la palabra ***class***, que se considera más segura y más propia de C++.

La solución que se utiliza habitualmente en C++ para resolver el problema de la ***encapsulación*** u ***ocultamiento de datos*** (***data hidding***) es que las funciones miembro públicas sean el único camino para acceder a las variables miembro (y funciones miembro) privadas. De esta manera, el programador de la clase puede tomar las medidas necesarias para que otros usuarios de la clase no modifiquen datos o funciones sin estar autorizados para ello.

Las funciones miembro públicas son habitualmente el ***interface*** entre los datos contenidos en los objetos de la clase y los usuarios de la clase. Estos usuarios no necesitan tener acceso a los detalles internos de dichos objetos; sólo necesitan saber cómo se utilizan las funciones miembro públicas. Esto facilita también el mantenimiento y la mejora del programa: la clase y sus funciones miembro pueden ser sustituidas por una versión nueva, más eficiente. En tanto en cuanto la declaración de las funciones miembro públicas se mantenga igual, el cambio no tendrá ninguna repercusión en los demás programas y funciones que hacen uso de la clase.

En el ejemplo anterior se puede ver ya cómo las ***funciones miembro***, que en este caso son todas ***públicas***, forman el ***interface*** utilizada para asignar y leer los valores de las variables miembro de la clase que son privadas. De esta manera se hace cumplir el *principio de encapsulación.*

3.6 Expansión *Inline*

3.6.1 DEFINICIÓN

Así pues, en la mayoría de los casos la programación orientada a objetos obliga a utilizar funciones para poder acceder a las variables miembro de una clase. Por esta razón un programa orientado a objetos contiene muchas llamadas a funciones. Por otra parte, muchas de las funciones que se utilizan contienen sólo unas pocas sentencias o incluso una sola, por ejemplo las funciones de lectura y asignación de valores de una variable.

Cada llamada y retorno de una función tiene un cierto costo computacional, porque es necesario reservar una zona de memoria para los argumentos de las funciones llamadas, que a veces, además tienen que ser copiados. En la mayoría de los casos el tiempo empleado en la transmisión de datos es despreciable frente al empleado en los cálculos. En el caso de que la función sea muy sencilla, sin embargo, no se puede despreciar ese tiempo y el uso frecuente de funciones muy sencillas y breves se revela muy poco eficiente

La expansión ***inline*** ofrece la solución a este problema sustituyendo en el programa la llamada a la función por el código de la misma, modificado para simular el paso de argumentos y valor de retorno. Las funciones ***inline*** eliminan la necesidad de utilizar las ***macros*** de C.

La ventajas de las funciones ***inline*** vienen dadas porque su utilización puede suponer una reducción del tiempo de ejecución de un programa. El inconveniente de usar funciones ***inline*** es que la introducción de una copia del código en cada llamada a una función puede hacer que el tamaño del programa aumente considerablemente. En definitiva, el uso de este tipo de funciones está recomendado en el caso de que sean funciones muy sencillas cuyo tiempo de llamada es comparable al tiempo de cálculo. Por el contrario, en el caso de funciones más grandes, la mejora en el tiempo de ejecución es despreciable y el tamaño del programa puede crecer innecesariamente.

3.6.2 IMPLEMENTACIÓN DE LAS FUNCIONES *INLINE*

C++ permite dos maneras distintas de implementar funciones ***inline***.

1. La primera de ellas consiste en colocar la palabra ***inline*** precediendo a la definición de la función:

```
inline double funcion (double uno)
    { return uno; }
```

   Es importante saber que esta indicación puede ser ignorada por el compilador en el caso de que la función en cuestión sea tan larga o tan complicada que su expansión ***inline*** resulte

desaconsejable. De todos modos, el que se produzca o no la expansión de la función no hace variar nada la definición de la misma.

La definición de las funciones ***inline*** debe hacerse en los ficheros de encabezamiento (extensión ****.h***), y no en los ficheros fuente (extensión ****.cpp***). Cada vez que se utiliza una función ***inline*** su llamada es sustituida por el código de la misma, por lo que éste debe ser accesible para cualquier fichero fuente, y eso se consigue incluyéndola en el fichero ***header***.

2. El segundo método de declaración de funciones ***inline*** consiste en colocar la ***definición*** completa de la misma en la ***declaración*** de la función:

```
class una {
    double tal;

    public:
        double Expandida( )
            { return tal;}
};
```

En el caso de que se incluya la definición de una función ***inline*** en una clase, esta misma ***definición*** sirve también como ***declaración***. Estas definiciones pueden incluir argumentos por defecto.

Tal como se han definido las funciones, incluyendo su definición en la declaración, estas funciones ya eran ***inline***. En el siguiente ejemplo se añade la palabra ***inline***, aunque no es necesaria, para subrayar esta característica de las funciones:

```
class C_Cuenta {
    // Variables miembro
    private:                            // La palabra private no es necesaria
        char *Nombre;      // Nombre de la persona
        double Saldo;      // Saldo actual de la cuenta
        double Interes;    // Interés aplicado

    public:
        // Métodos
        inline char *GetNombre()      // La palabra inline no es necesaria
            { return Nombre; }
        inline double GetSaldo()
            { return Saldo; }
        inline double GetInteres()
            { return Interes; }
        inline void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        inline void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
};
```

3.7 Entrada y salida de datos

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*. C++ hace que tanto la entrada como la salida de datos se vayan dirigiendo a unos *flujos* de entrada y/o de salida, sumándose a lo que ya se arrastra en esa dirección. A veces se comparan los ***streams*** con una cinta transportadora que introduce o saca datos del programa.

Al incluir la librería **<*iostream.h*>** se añaden al programa varios ***streams*** estándar. Los más utilizados son ***cin***, que se utiliza para la entrada de datos (habitualmente desde teclado), ***cout***, que se

utiliza para la salida (habitualmente por pantalla) y ***cerr***, que se utiliza para los mensajes de error (también por pantalla).

El ***operador de inserción*** (<<) se utiliza para insertar datos en un ***stream***. De esta manera, una instrucción del tipo:

```
cout << 500;
```

coloca el valor 500 en el ***stream*** estándar de salida ***cout***, de modo que por pantalla aparecerá el número 500.

Se pueden utilizar varios operadores de inserción en una misma instrucción, de forma que vayan añadiéndose datos al ***stream*** de salida. Así una sentencia del tipo:

```
cout << 111 << 222 << 333;
```

imprimirá por pantalla:

```
111222333
```

De una forma muy parecida se pueden imprimir cadenas de caracteres. La sentencia,

```
cout << "Hola a todos";
```

imprime:

```
Hola a todos
```

Tampoco tiene ninguna dificultad imprimir ***variables*** y/o ***constantes*** que sean combinación de los vistos hasta ahora, como por ejemplo en la sentencia:

```
cout << 500 << " y " << 600 << "son números pares.";
```

que imprime por pantalla el resultado siguiente:

```
500 y 600 son números pares.
```

Al igual que C, C++ utiliza ***secuencias de escape*** para representar símbolos que no son los convencionales. Las secuencias de escape están siempre formadas por el carácter (\) seguido de otro carácter. Toda la secuencia de escape representa un único carácter y se considera indivisible. Las secuencias de escape más utilizadas se incluyen a continuación:

\a	alerta	Hace emitir un beep al ordenador
\n	nueva línea	Hace saltar el cursor a la línea siguiente
\t	tabulador	Desplaza el cursor la distancia de un tabulador
\”	comillas	Imprime unas comilla
\\	barra inclinada	Imprime una barra inclinada hacia la izquierda

Tabla 1: Secuencias de escape

Como se puede ver, la mayoría de estas secuencias de escape son equivalentes a las que ya se usaban en ANSI C.

3.7.1 UNA BREVE COMPARACIÓN CON LA ENTRADA Y SALIDA DE DATOS DE ANSI C

En C++ también se pueden utilizar las funciones ***printf()*** y ***scanf(),*** si se incluye la librería ***<stdio.h>***. En la práctica esta opción no se utiliza mucho, porque los operadores (<<) y (>>) tienen algunas ventajas que se van a citar a continuación.

Con los operadores propios de C++ se evita el chequeo de compatibilidad entre el especificador de formato (lo que sigue al carácter %, como por ejemplo %s, %d, %lf, etc.) y el argumento actual de la función ***printf()*** o ***scanf()***. Esto hace que se elimine una importante fuente de errores, que provenía, además, de una innecesaria duplicidad de información de los tipos de variable a imprimir o leer. En C++ el compilador determina en tiempo de compilación el tipo de variable que el usuario desea imprimir y aplica un formato por defecto adecuado, evitando cualquier incompatibilidad.

Como se verá en el apartado de sobrecarga de operadores, C++ permite el mismo tratamiento a los tipos de variable definidos por el usuario que a los tipos de datos predefinidos del lenguaje. Esto quiere decir que el programador puede definir un operador (**<<**) especial para imprimir por ejemplo números complejos (que incluya la letra ***i*** para indicar la parte imaginaria). Esta capacidad no está soportada por las funciones del ANSI C conocidas, a las que no se pueden añadir nuevos especificadores de formato que permitan leer o escribir directamente los datos definidos por el usuario.

3.8 Operadores *new* y *delete* con clases

Los operadores ***new*** y ***delete*** pueden ser aplicados tanto a variables de los tipos predefinidos (***int***, ***float***, ***double***, …) como a objetos de las clases definidas por el usuario. Su principal aplicación está en la creación de variables con ***reserva dinámica de memoria***, sustituyendo a las funciones ***calloc()***, ***malloc()*** y ***free()*** que se utilizan en C. Aquí es necesario hacer alguna matización referente a su uso con las clases definidas por el usuario.

Una vez creada la clase ***C_Cuenta***, una sentencia del tipo:

```
C_Cuenta *c1 = new C_Cuenta;
```

supone una operación en tres fases:

1. En primer lugar se crea un puntero ***c1*** capaz de contener la dirección de un objeto de la clase.
2. A continuación, por medio del operador ***new*** se reserva memoria para un objeto del tipo ***C_Cuenta***.
3. Finalmente se llama, de modo transparente al usuario, a un ***constructor*** de la clase ***C_Cuenta***.

Los ***constructores*** son funciones que ***se llaman automáticamente*** al crear un objeto de una clase. Su misión es ***dar un valor inicial*** a todas las variables miembro, de modo que no haya nunca objetos con variables sin un valor apropiado (conteniendo basura informática). Los ***constructores*** se explican en el apartado siguiente. El operador ***new*** se puede utilizar para crear ***vectores de objetos*** (en una forma similar a como se crean vectores de variables) con reserva dinámica de memoria, en la forma:

```
C_Cuenta *ctas = new C_Cuenta[100];
```

La diferencia fundamental entre ***new*** y ***delete*** por una parte**,** y ***malloc()*** y ***free()*** por otra es que los primeros crean y destruyen ***objetos,*** mientras que los segundos se limitan a reservar y liberar zonas de memoria. Además hay que tener en cuenta que ***new*** puede ser ***sobrecargado*** como cualquier otro operador, con las ventajas de simplificación de código que ello supone. La sobrecarga de operadores se estudiará en un apartado posterior.

Al utilizar ***delete*** (lo mismo sucede con ***free()***), se libera la zona de memoria a la que apunta, *pero sin borrar el propio puntero*. Si se manda liberar lo apuntado por un puntero nulo no pasa nada especial y el programa no libera memoria. Sin embargo si se ordena *liberar dos veces* lo apuntado

por un puntero las consecuencias son imprevisibles y puede que incluso catastróficas, por lo que es importante evitar este tipo de errores.

En el caso de que se desee liberar la memoria ocupada por un vector creado mediante reserva dinámica de memoria debe emplearse una instrucción del tipo:

```
delete [] ctas;
```

siendo ***ctas*** el puntero al primer elemento del vector (el puntero que recogió el valor de retorno de ***new***).

3.9 Constructores y destructores

Ya se ha apuntado que C++ no permite crear objetos sin dar un valor inicial apropiado a todas sus variables miembro. Esto se hace por medio de unas funciones llamadas ***constructores***, que se llaman automáticamente ***siempre que se crea un objeto*** de una clase.

Se van a estudiar ahora algunas formas posibles de crear e inicializar ***objetos***, tales como ***c1***. Una primera manera en la que se podría hacer esto bien pudiera ser ésta:

```
C_Cuenta  c1;             // se crea el objeto
c1.Saldo = 500.0;         // se da valor a sus variables miembro
c1.Interes= 10.0;
```

Este método, a pesar de ser correcto, viola el *principio de encapsulación* al manejar directamente las variables miembro. Al igual que sucede con los arrays y cadenas de caracteres, C y C++ también permiten declarar e inicializar las variables de la siguiente manera (más compacta, que es una *inicialización*):

```
C_Cuenta c1 = { 500.0, 10.0 };
```

Los valores que aparecen entre las llaves son asignados a las variables miembro de la clase o estructura, *en el mismo orden en que esas variables aparecen en la declaración de esa clase*. De todos modos, esta forma de declarar las variables incumple también el *principio de encapsulación*, que es uno de los objetivos de la programación orientada a objetos.

Para permitir la creación de objetos siendo fieles a la *encapsulación* se utilizan los ***constructores***. Estos son unas funciones miembro de una clase especiales que se llaman de modo automático al crear los objetos, y dan un valor inicial a cada una de las variables miembro. El ***nombre del constructor*** es siempre el mismo que el ***nombre de la clase***. Así el constructor de la clase ***C_Cuenta*** se llamará, a su vez, ***C_Cuenta***. Además, los constructores se caracterizan porque *se declaran y definen sin valor de retorno*, ni siquiera ***void***. Utilizando las capacidades de sobrecarga de funciones de C++, para una clase se pueden definir ***varios constructores***.

El uso del ***constructor*** es tan importante que, en el caso de que el programador no defina ningún constructor para una clase, el compilador de C++ proporciona un ***constructor de oficio*** (también llamado a veces ***por defecto***, aunque como más tarde se verá que esta terminología puede resultar confusa).

Si la clase ***C_Cuenta*** se completa con su ***constructor***, su declaración quedará de la siguiente forma:

```
class C_Cuenta {
    // Variables miembro
    private:
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado
    public:
        // Constructor
        C_Cuenta(double unSaldo, double unInteres);
```

```
        // Acciones básicas
        double GetSaldo()
            { return Saldo; }
        double GetInteres()
            { return Interes; }
        void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
};
```

Conviene insistir en que el ***constructor*** se declara y define sin valor de retorno, y que en una clase puede haber ***varios constructores***, pues como el resto de las funciones puede estar sobrecargado. La definición del ***constructor*** de la clase ***C_Cuenta*** pudiera ser la que a continuación se presenta:

```
C_Cuenta::C_Cuenta(double unSaldo, double unInteres)
{
    // La clase puede hacer llamadas a los métodos
    //    previamente definidos
    SetSaldo(unSaldo);
    SetInteres(unInteres);
}
```

Teniendo en cuenta que el ***constructor*** es una ***función miembro***, y que como tal tiene ***acceso directo a las variables miembro privadas*** de la clase (sin utilizar los operadores punto o flecha), el ***constructor*** también podría definirse del siguiente modo:

```
C_Cuenta::C_Cuenta(double unSaldo, double unInteres)
{
    // El constructor accede a las variables miembro y les asigna el
    //    valor de los parámetros
    Saldo   = unSaldo;
    Interes = unInteres;
}
```

3.9.1 INICIALIZADORES

Todavía se puede programar al ***constructor*** de una tercera forma, con ventajas sobre las explicadas en el apartado anterior. La idea del constructor es ***inicializar variables***, y una sentencia de asignación no es la única ni la mejor forma de inicializar una variable. C++ permite ***inicializar*** variables miembro fuera del cuerpo del constructor, de la siguiente forma:

```
C_Cuenta::C_Cuenta(double unSaldo, double unInteres) :
                    Saldo(unSaldo), Interes(unInteres)       // inicializadores
{    // En este caso el cuerpo del constructor está vacío }
```

donde se ve que los ***inicializadores*** se introducen, tras el carácter dos puntos (:), separados por comas, justo antes de abrir las llaves del cuerpo del constructor. Constan del nombre de la variable miembro seguido, entre paréntesis, del argumento que le da valor. Los ***inicializadores*** son más eficientes que las sentencias de asignación, y además permiten definir variables miembro ***const***, que pueden ser inicializadas pero no asignadas.

3.9.2 LLAMADAS AL CONSTRUCTOR

Ya se ha dicho que el operador ***new*** se encarga de llamar al ***constructor*** de una clase cada vez que se crea un objeto de esa clase.

La llamada al ***constructor*** se puede hacer explícitamente en la forma:

```
C_Cuenta c1 = C_Cuenta(500.0, 10.0);
```

o bien, de una forma implícita, más abreviada, permitida por C++:

```
C_Cuenta c1(500.0, 10.0);
```

Los dos ejemplos que se acaban de presentar tienen en común el que se trata de crear el objeto ***c1*** perteneciente a la clase ***C_Cuenta***. Esto va en la línea de lo ya apuntado: siempre que se crea un ***objeto*** de una clase, se llama implícita o explícitamente al ***constructor*** de la clase para que lo inicialice.

3.9.3 CONSTRUCTOR POR DEFECTO Y CONSTRUCTOR CON PARÁMETROS CON VALOR POR DEFECTO

Se llama ***constructor por defecto*** a un constructor que no necesita que se le pasen parámetros o argumentos para inicializar las variables miembro de la clase. Un ***constructor por defecto*** es pues un constructor que no tiene argumentos o que, si los tiene, todos sus argumentos tienen asignados un valor por defecto en la declaración del constructor. En cualquier caso, puede ser llamado sin tenerle que pasar ningún argumento.

El ***constructor por defecto es necesario*** si se quiere hacer una declaración en la forma:

```
C_Cuenta c1;
```

y también cuando se quiere crear un ***vector de objetos***, por ejemplo en la forma:

```
C_Cuenta cuentas[100];
```

ya que en este caso se crean e inicializan múltiples objetos sin poderles pasar argumentos personalizados o propios para cada uno de ellos.

Al igual que todas las demás funciones de C++, el ***constructor*** puede tener definidos unos ***valores por defecto*** para los parámetros, que se asignen a las variables miembro de la clase. Esto es especialmente útil en el caso de que una variable miembro repita su valor para todos o casi todos los objetos de esa clase que se creen. Considérese el ejemplo siguiente:

```
class C_Cuenta {
    // Variables miembro
    private:
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado
    public:
        // Constructor
        C_Cuenta(double unSaldo=0.0, double unInteres=0.0)
        {
            SetSaldo(unSaldo);
            SetInteres(unInteres);
        }
        // Métodos
        char *GetNombre()
            { return Nombre; }
        double GetSaldo()
            { return Saldo; }
        double GetInteres()
            { return Interes; }
        void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
};
```

```
void main() {
    // Ya es válida la construcción sin parámetros
    C_Cuenta C0;                // unSaldo=0.0 y unInteres=0.0
    // También es válida con un parámetro
    C_Cuenta C1(10.0);          // unSaldo=10.0 y unInteres=0.0
    // y con dos parámetros
    C_Cuenta C2(20.0, 1.0);     // unSaldo=20.0 y unInteres=1.0
    ...
}
```

En el ejemplo anterior se observa la utilización de un mismo ***constructor*** para crear objetos de la clase ***C_Cuenta*** de tres maneras distintas. La primera llamada al ***constructor*** se hace sin argumentos, por lo que las variables miembro tomaran los valores por defecto dados en la definición de la clase. En este caso ***Saldo*** valdrá 0 e ***Interes*** también valdrá 0.

En la segunda llamada se pasa un único argumento, que se asignará a la primera variable de la definición del ***constructor***, es decir a ***Saldo***. La otra variable, para la que no se asigna ningún valor en la llamada, tomará el valor asignado por defecto. Hay que recordar aquí que *no es posible en la llamada asignar un valor al segundo argumento si no ha sido asignado antes otro valor a todos los argumentos anteriores* (en este caso sólo al primero). En la tercera llamada al ***constructor*** se pasan dos argumentos por la ventana de la función, por lo que las variables miembro tomarán esos valores.

3.9.4 CONSTRUCTOR DE OFICIO

¿Qué hubiera pasado si en la clase ***C_Cuenta*** no se hubiera definido ningún ***constructor***? Pues en este caso concreto, no hubiera pasado nada, o al menos nada catastrófico. El compilador de C++ habría creado un ***constructor de oficio***, sin argumentos. ¿Qué puede hacer un ***constructor*** sin argumentos? Pues lo más razonable que puede hacer es inicializar todas las variables miembro a cero. Quizás esto es muy razonable para el ***Saldo***, aunque quizás no tanto para la variable ***Interes***.

Así pues, se llamará en estos apuntes ***constructor por defecto*** a un constructor que no tiene argumentos o que si los tiene, se han definido con valores por defecto. Se llamará ***constructor de oficio*** al ***constructor por defecto*** que define automáticamente el compilador si el usuario no define ningún constructor. Ambos conceptos no son equivalentes, pues si bien todo ***constructor de oficio*** es ***constructor por defecto*** (ya que no tiene argumentos), lo contrario no es cierto, pues el programador puede definir ***constructores por defecto*** que obviamente no son ***de oficio***.

Un punto importante es que el compilador sólo crea un ***constructor de oficio*** en el caso de que el programador no haya definido ***ningún constructor***. En el caso de que el usuario sólo haya definido un ***constructor con argumentos*** y se necesite un ***constructor por defecto*** para crear por ejemplo un ***vector de objetos***, el compilador no crea este ***constructor por defecto*** sino que da un mensaje de error.

Los ***constructores de oficio*** son cómodos para el programador (no tiene que programarlos) y en muchos casos también correctos y suficientes. Sin embargo, ya se verá en un próximo apartado que en ocasiones conducen a resultados incorrectos e incluso a errores fatales.

3.9.5 CONSTRUCTOR DE COPIA

Ya se ha comentado que C++ ***obliga a inicializar*** las variables miembro de una clase llamando a un ***constructor***, cada vez que se crea un objeto de dicha clase. Se ha comentado también que el constructor puede recibir como parámetros los valores que tiene que asignar a las variables miembro, o puede asignar valores por defecto.

Existe un caso particular de gran interés no comprendido en lo explicado hasta ahora y que se produce cuando se ***crea un objeto*** inicializándolo a partir de ***otro objeto de la misma clase***. Por ejemplo, C++ permite crear tres objetos ***c1, c2*** y ***c3*** de la siguiente forma:

```
C_Cuenta c1(1000.0, 8.5);
C_Cuenta c2 = c1;
C_Cuenta c3(c1);
```

En la primera sentencia se crea un objeto ***c1*** con un saldo de 1000 y un interés del 8.5%. En la segunda se crea un objeto ***c2*** a cuyas variables miembro se les asignan los mismos valores que tienen en ***c1***. La tercera sentencia es una forma sintáctica equivalente a la segunda: también ***c3*** se inicializa con los valores de ***c1***.

En las sentencias anteriores se han creado tres objetos y por definición se ha tenido que llamar tres veces a un constructor. Realmente así ha sido: en la primera sentencia se ha llamado al ***constructor con argumentos*** definido en la clase, pero en la segunda y en la tercera se ha llamado a un constructor especial llamado ***constructor de copia*** (***copy constructor***). Por definición, el ***constructor de copia*** tiene ***un único argumento*** que es una ***referencia constante a un objeto de la clase***. Su declaración sería pues como sigue:

```
C_Cuenta(const C_Cuenta&);
```

Las sentencias anteriores de declaración de los objetos ***c2*** y ***c3*** funcionarían correctamente aunque no se haya declarado y definido en la clase ***C_Cuenta*** ningún constructor de copia. Esto es así porque el compilador de C++ proporciona también un ***constructor de copia de oficio***, cuando el programador no lo define. El ***constructor de copia de oficio*** se limita a realizar una ***copia bit a bit*** de las variables miembro del objeto original al objeto copia. En este caso, eso es perfectamente correcto y es todo lo que se necesita. Pronto se verá algún ejemplo en el que esta copia bit a bit no da los resultados esperados. En este caso el programador debe preparar su propio ***constructor de copia*** e incluirlo en la clase como un constructor sobrecargado más.

Además del ejemplo visto de declaración de un objeto iniciándolo a partir de otro objeto de la misma clase, hay otros dos casos muy importantes en los que se utiliza el constructor de copia:

1. Cuando a una función se le pasan objetos como ***argumentos por valor***, y
2. Cuando una función tiene un ***objeto como valor de retorno***.

En ambos casos hay que crear copias del objeto y para ello se utiliza el ***constructor de copia***.

3.9.6 NECESIDAD DE ESCRIBIR UN CONSTRUCTOR DE COPIA

Ha llegado ya el momento de explicar cómo surge la necesidad de escribir un ***constructor de copia*** distinto del que proporciona el compilador. Considérese una clase ***Alumno*** con dos variables miembro: un puntero a ***char*** llamado ***nombre*** y un ***long*** llamado ***nmat*** que representa el número de matrícula..

```
class Alumno {
    char* nombre;
    long  nmat;
    ...
};
```

En realidad, *esta clase no incluye el nombre del alumno*, sino sólo un puntero a carácter que permitirá almacenar la dirección de memoria donde está realmente almacenado el nombre. Esta memoria se reservará dinámicamente cuando el objeto vaya a ser inicializado. Lo importante es darse cuenta de que *el nombre no es realmente una variable miembro de la clase*: la variable miembro es un puntero a la zona de memoria donde está almacenado. Esta situación se puede ver gráficamente en la figura 1, en la que se muestra un objeto ***a*** de la clase ***Alumno***.

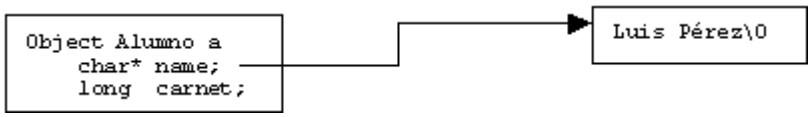


Figura 1. Objeto con reserva dinámica de memoria.

Supóngase ahora que con el ***constructor de copia*** suministrado por el compilador se crea un nuevo objeto ***b*** a partir de ***a***. Las variables miembro de ***b*** van a ser una ***copia bit a bit*** de las del objeto ***a***. Esto quiere decir que la variable miembro ***b.nombre*** contiene la misma dirección de memoria que ***a.nombre***. Esta situación se representa gráficamente en la figura 2: ambos objetos apuntan a la misma dirección de memoria.

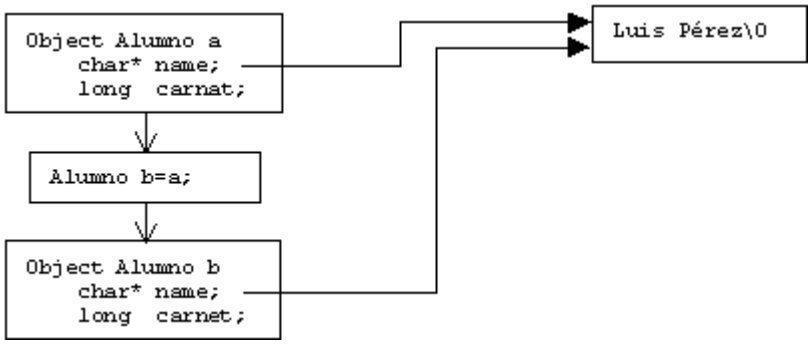


Figura 2. Copia bit a bit del objeto de la figura 1.

La situación mostrada en la figura 2 puede tener consecuencias no deseadas. Por ejemplo, si se quiere cambiar el nombre del Alumno ***a***, lo primero que se hace es liberar la memoria a la que apunta ***a.nombre***, reservar memoria para el nuevo nombre haciendo que ***a.nombre*** apunte al comienzo de dicha memoria, y almacenar allí el nuevo nombre de ***a***. Como el objeto ***b*** no se ha tocado, su variable miembro ***b.nombre*** se ha quedado apuntado a una posición de memoria que ha sido liberada en el proceso de cambio de nombre de ***a***. La consecuencia es que ***b*** ha perdido información y los más probable es que el programa falle.

Se llega a una situación parecida cuando se destruye uno de los dos objetos ***a*** o ***b***. Al destruir uno de los objetos se libera la memoria que comparten, con el consiguiente perjuicio para el objeto que queda, puesto que su puntero contiene la dirección de una zona de memoria liberada, disponible para almacenar otra información.

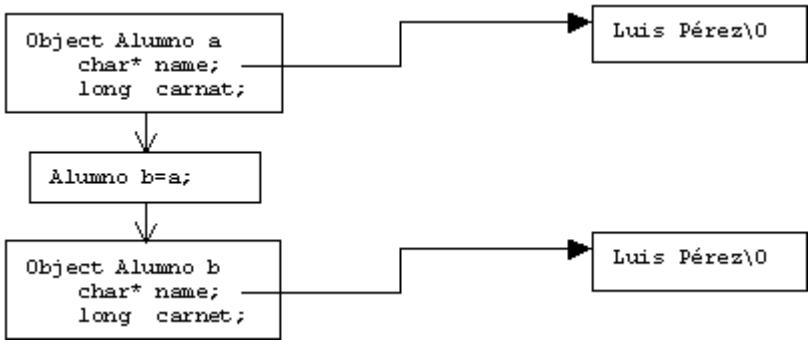


Figura 3. Copia correcta del objeto de la figura 1.

Finalmente, la figura 3 muestra la situación a la que se llega con un ***constructor de copia*** correctamente programado por el usuario. En este caso, el constructor ***no copia bit a bit*** la dirección contenida en ***a.nombre***, sino que reserva memoria, copia a esa memoria el contenido apuntado por ***a.nombre***, y guarda en ***b.nombre*** la dirección de esa nueva memoria reservada. Ninguno de los problemas anteriores sucede ahora.

3.9.7 LOS CONSTRUCTORES Y EL OPERADOR DE ASIGNACIÓN (=)

En un apartado anterior se ha visto una declaración de un objeto en la forma:

```
C_Cuenta c2 = c1;
```

Esta sentencia es completamente diferente de una simple ***sentencia de asignación*** entre dos objetos previamente creados, tal como la siguiente:

```
c2 = c1;
```

En este último caso no se llama a ningún ***constructor***, porque se supone que ***c1*** y ***c2*** existían previamente. En este segundo caso se utiliza el ***operador de asignación*** (=) estándar de C y C++, que permite realizar asignaciones entre objetos de estructuras o clases. Este operador realiza una ***asignación bit a bit*** de los valores de las variables miembro de ***c1*** en ***c2***. En este sentido es similar al ***constructor de copia*** y, por las mismas razones que éste, da lugar también a errores o resultados incorrectos.

La solución en este caso es ***redefinir*** o ***sobrecargar*** el ***operador de asignación*** (=) de modo que vaya más allá de la ***copia bit a bit*** y se comporte adecuadamente. En el apartado anterior se ha explicado a fondo el origen y la solución de este problema, común al ***constructor de copia*** y al ***operador de asignación*** (=). Básicamente, el ***operador de asignación sobrecargado*** (=) debe de llegar a una situación como la que se muestra en la figura 3.

3.9.8 DESTRUCTORES

El complemento a los constructores de una clase es el ***destructor***. Así como el constructor se llama al declarar o crear un objeto, el ***destructor*** es llamado cuando el objeto va a dejar de existir por haber llegado al final de su vida. En el caso de que un objeto (***local*** o ***auto***) haya sido definido dentro de un bloque {…}, el ***destructor*** es llamado cuando el programa llega al final de ese bloque. Si el objeto es ***global*** o ***static*** su duración es la misma que la del programa, y por tanto el ***destructor*** es llamado al terminar la ejecución del programa. Los objetos creados con ***reserva dinámica de memoria*** (en general, los creados con el operador ***new***) no están sometidos a las reglas de duración habituales, y existen hasta que el programa termina o hasta que son explícitamente destruidos con el operador ***delete***. En este caso la responsabilidad es del programador, y no del compilador o del sistema operativo.

A diferencia del ***constructor***, el ***destructor*** es siempre único (no puede estar sobrecargado) y ***no tiene argumentos*** en ningún caso. Tampoco tiene ***valor de retorno***. Su nombre es el mismo que el de la clase precedido por el carácter tilde (~), carácter que se consigue con ***Alt+126*** en el teclado del PC. En el caso de que el programador no defina un ***destructor***, el compilador de C++ proporciona un ***destructor de oficio***, que es casi siempre plenamente adecuado (excepto para liberar memoria de vectores y matrices).

En el caso de que la clase ***C_Cuenta*** necesitase un ***destructor***, la declaración sería así:

```
~C_Cuenta();
```

y la definición de la clase, añadiendo en este caso como variable miembro una cadena de caracteres que contenga el nombre del titular, podría ser como sigue:

```
class C_Cuenta {
    // Variables miembro
    private:
        char   *Nombre;    // Nombre de la persona
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado
    public:
        //Constructor
        C_Cuenta(const char *unNombre, double unSaldo=0.0, double unInteres=0.0)
        {
            Nombre = new char[strlen(unNombre)+1];
            strcpy(Nombre, unNombre);
            Saldo = unSaldo;
            Interes = unInteres;
        }
        // Destructor
        ~C_Cuenta()
            { delete [] Nombre; }   // Libera la memoria apuntada por el puntero

        // Métodos
        char *GetNombre()
            { return Nombre; }
        double GetSaldo()
            { return Saldo; }
        double GetInteres()
            { return Interes; }
        void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
};

void main(void) {
    C_Cuenta C0("Igor");
    // Tambien es valida con dos argumentos
    C_Cuenta C1("Juan", 10.0);
    //   y con los tres argumentos
    C_Cuenta C2("Itxaso", 20.0, 1.0);
    ...
}
```

3.10 Clases y funciones *friend*

Ya se ha visto anteriormente que declarar como ***privadas*** las ***variables miembro*** de una clase ofrece muchas ventajas. De todos modos, en algunos casos, puede suceder que dos clases vayan a trabajar conjuntamente con los mismos datos, y utilizar las funciones públicas de acceso a esos datos no es la manera más eficiente de hacerlo; conviene recordar que llamar a una función tiene un coste y que es mucho más eficiente acceder directamente a una variable. Ésta no es la única limitación de las ***funciones miembro***, ya que una función ***sólo puede ser miembro de una única clase***. Además una función miembro convencional sólo puede actuar directamente sobre un único objeto de la clase (su argumento implícito, que es el objeto con el que ha sido llamada por medio del operador punto (.) o flecha (->), como por ejemplo en ***c1.Ingreso(10000)***). Para que actúe sobre un segundo o un tercer objeto -por ejemplo, para hacer transferencias- hay que pasárselos como argumentos.

Se puede concluir que a pesar de las grandes ventajas que tiene el ***encapsulación***, en muchas ocasiones es necesario dotar a la *programación orientada a objetos* de una mayor ***flexibilidad***. Esto se consigue por medio de las funciones ***friend***. Una función ***friend*** de una clase es una función que ***no pertenece a la clase***, pero que ***tiene permiso*** para acceder a sus variables y funciones miembro privadas por medio de los *operadores punto* (.) y *flecha* (->), sin tener que recurrir a las funciones

miembro públicas de la clase. Si una clase se declara ***friend*** de otra, todas sus funciones miembro son ***friend*** de esta segunda clase. El carácter de ***friend*** puede restringirse a funciones concretas, que pueden ser miembro de alguna clase o pueden ser funciones generales que no pertenecen a ninguna clase..

Para ***declarar*** una función o una clase como ***friend*** de otra clase, es necesario hacerlo ***en la declaración de la clase que debe autorizar el acceso*** a sus datos privados. Esto se puede hacer de forma indiferente en la zona de los datos o en la de los datos privados. Un ejemplo de declaración de una clase ***friend*** podría ser el que sigue:

```
class Cualquiera
{
    friend class Amiga;
    private:
        int secreto;
};
```

Es muy importante tener en cuenta que esta relación funciona sólo en una dirección, es decir, las funciones miembro de la clase ***Amiga*** pueden acceder a las variables privadas de la clase ***Cualquiera***, por ejemplo a la variable entera ***secreto***, pero esto no es cierto en sentido inverso: las funciones miembro de la clase ***Cualquiera*** no puede acceder a un dato privado de la clase ***Amiga***.

Si se quiere que varias clases tengan acceso mutuo a todas las variables y funciones miembro privadas, cada una debe declararse como ***friend*** en todas las demás, para conseguir una relación recíproca.

Otro aspecto que hay que mencionar es que al definir una clase como ***friend*** *no se está haciendo* ***friend*** *a todas las clases que se deriven de ella* (esto se entenderá al llegar al capítulo de la ***herencia***).

Hay que tener en cuenta que para que una función que no es miembro de una clase pueda recibir como argumentos explícitos objetos de esa clase, debe ser declarada ***friend*** de esa clase. Un ejemplo de una función ***friend*** es el que sigue:

```
class Cualquiera
{
    friend void fAmiga(Cualquiera);
    private:
        int secreto;
};

void fAmiga(Cualquiera Una)
{
    Una.secreto++;          // Modifica el valor de la variable privada
}
```

Recuérdese que una función puede ser declarada ***friend*** de cuantas clases se quiera, pero sólo puede ser *miembro* de una única clase. En algunos casos interesará que no sea miembro de ninguna y que sea ***friend*** de una o más clases.

Se plantea un problema cuando dos ***clases*** deben ser declaradas mutuamente ***friend*** la una de la otra. Considérense por ejemplo las clases ***vector*** y ***matriz***. Las declaraciones podrían ser como se muestra a continuación:

```
// declaración de las clases vector y matriz (mutuamente friend)
class matriz;    // declaración anticipada de la clase matriz

class vector {
// declaración de funciones miembro de la clase vector
    ...
    friend matriz;
};
```

```
class matriz {
// declaración de funciones miembro de la clase matriz
    ...
    friend vector;
};
```

A primera vista sorprende la segunda línea de lo que sería el fichero de declaración de ambas clases. En esta línea aparece la sentencia ***class matriz;***. Esto es lo que se llama una ***declaración anticipada***, que es necesaria por el motivo que se explica a continuación. Sin declaración anticipada cuando la clase ***matriz*** aparece declarada como ***friend*** en la clase ***vector***, el compilador aún no tiene noticia de dicha clase, cuya declaración viene después. Al no saber lo que es ***matriz***, da un error. Si la clase ***matriz*** se declara antes que la clase ***vector***, el problema se repite, pues ***vector*** se declara como ***friend*** en ***matriz***. La única solución es esa ***declaración anticipada*** que advierte al compilador que ***matriz*** es una clase cuya declaración está más adelante.

3.11 El puntero *this*

El puntero ***this*** es una variable predefinida para todas las ***funciones u operadores miembro*** de una clase. Este puntero contiene la dirección del objeto concreto de la clase al que se está aplicando la función o el operador miembro. Se puede decir que ***this** es un alias del objeto correspondiente. Conviene tener en cuenta que cuando una función miembro se aplica a un objeto de su clase (su argumento implícito), accede directamente a las variables miembro (sin utilizar el operador punto o flecha), pero no tiene forma de referirse al objeto como tal, pues no le ha sido pasado explícitamente como argumento. Este problema se resuelve con el puntero ***this***. Considérese el siguiente ejemplo:

```
Class C_Cuenta {
        //...
    public:
        // ...
        inline double GetInteres()
        {
            // Igual a : return Interes;
            return this->Interes;
        }
        // ....
};
```

En el caso de operadores miembro sobrecargados, el puntero ***this*** es la forma que se utiliza para referirse al objeto al que se está aplicando el operador como primer operando.

Hay que señalar que las funciones ***friend*** que no son miembros de ninguna clase no disponen de puntero ***this***.

3.12 Sobrecarga de operadores

Los ***operadores*** de C++, al igual que las funciones, pueden ser sobrecargados (***overloaded***). Este es uno de los aspectos más característicos de este lenguaje. La ***sobrecarga de operadores*** quiere decir que se pueden ***redefinir algunos de los operadores existentes*** en C++ para que actúen de una determinada manera, definida por el programador, con los *objetos* de una clase determinada. Esto puede ser muy útil por ejemplo, para definir operaciones matemáticas con elementos tales como ***vectores*** y ***matrices***. Así, sobrecargando adecuadamente los operadores ***suma*** (+) y ***asignación*** (=), se puede llegar a sumar dos matrices con una sentencia tan simple como:

```
C = A + B;
```

Otra capacidad muy utilizada es la de sobrecargar los ***operadores de inserción y extracción*** en los flujos de entrada y salida (>> y <<), de manera que puedan imprimir o leer estructuras o clases complejas con una sentencia estándar.

Los únicos operadores de C que no se pueden sobrecargar son el operador *punto* (.), el *if aritmético* (?:) y el operador *sizeof*. C++ añade otros 2 a esta lista: el *scope resolution operator* (::) y *puntero a miembro de un objeto* (.*).

El objetivo último de la sobrecarga de operadores es simplificar al máximo el código a escribir, a cambio de complicar algo la definición de las clases. Una clase que disponga de operadores sobrecargados es una clase más compleja de definir, pero más sencilla e intuitiva de utilizar.

Las ventajas de la sobrecarga de operadores terminan cuando se utiliza de modo que añade complejidad o confusión a los programas. Por ejemplo, aunque esté permitido por el lenguaje, no se deberá nunca utilizar el operador (-) para multiplicar matrices o el (+) para imprimir vectores.

La sobrecarga de operadores tiene dos limitaciones teóricas y una práctica:

- Se puede modificar la ***definición*** de un operador pero no su ***gramática***, es decir, el número de operandos sobre los que actúa, la precedencia y la asociatividad.
- Es necesario que al menos un operando sea un ***objeto de la clase*** en la que se ha definido el operador sobrecargado.
- Como se verá más adelante al presentar el ejemplo ***cadena***, la sobrecarga de operadores puede resultar bastante ***ineficaz***, desde el punto de vista de utilización de los recursos del ordenador.

Un operador puede estar sobrecargado o redefinido varias veces, de tal manera que actúe de un modo distinto dependiendo del tipo de objetos que tenga como operandos. Es precisamente el tipo de los operandos lo que determina qué operador se utiliza en cada caso.

Un ***operador sobrecargado*** puede ser ***miembro*** o ***friend*** de la clase para la que se define (nunca las dos cosas a la vez). El que se defina de una forma u otra es en ocasiones cuestión de conveniencia o incluso de preferencia personal, mientras que en otros casos la decisión está impuesta. Habitualmente:

- Se suelen declarar ***miembros de la clase*** los operadores ***unarios*** (es decir, aquellos que actúan sobre un único objeto), o los que ***modifican*** el primer operando, como sucede con los operadores de asignación.
- Por el contrario, los operadores que actúan sobre varios objetos sin modificarlos (por ejemplo los operadores aritméticos y lógicos) se suelen declarar como ***friends***.

Para los *operadores sobrecargados miembro* de una clase, *el primer operando debe de ser siempre un objeto de esa clase*, en concreto el objeto que constituye el ***argumento implícito***. En la declaración y en la definición sólo hará falta incluir en la lista de argumentos el segundo operando. En los operadores ***friend*** el número de argumentos deberá ser el estándar del operador (unario o binario).

3.12.1 CLASE ***CADENA*** PARA MANEJO DE CADENAS DE CARACTERES.

Para explicar la sobrecarga de operadores y los diversos conceptos que implica se va a utilizar un ejemplo diferente a la ***cuenta corriente*** que se ha estado considerando hasta ahora. Se va a crear una clase llamada ***cadena*** que permita trabajar con cadenas de caracteres de un modo más directo e

intuitivo de lo que permite C. Por ejemplo, en C no se puede copiar una cadena de caracteres en otra con el ***operador de asignación*** (=), sino que es necesario utilizar la función ***strcpy()***; tampoco se pueden concatenar cadenas con el ***operador suma*** (+), ni se pueden comparar con los ***operadores relacionales*** (==) y (!=), sino que hay que utilizar las funciones ***strcat()*** y ***strcmp()***, respectivamente.

Las variables miembro de la clase ***cadena*** van a ser el número de caracteres ***nchar*** (sin incluir el '\0' final) y un puntero a ***char*** llamado ***pstr*** que contendrá la dirección del primer carácter de la cadena. La declaración de la clase ***cadena***, con todas sus funciones y operadores miembro, y con otros operadores que sólo son ***friend***, está contenida en un fichero llamado ***cadena.h***, y es como sigue:

```
// fichero cadena.h
#include <iostream.h>
#ifndef __CADENA_H
#define __CADENA_H

class cadena {
private:
    char* pstr;
    int   nchar;            // nº de caracteres (sin el '\0')
public:
    cadena();               // constructor por defecto
    cadena(char*);          // constructor general
    cadena(const cadena&);  // constructor de copia
    ~cadena();              // destructor
    void setcad(char*);     // dar valor a la variable privada pstr

    // sobrecarga de operadores
    cadena& operator= (const cadena&);
    friend cadena operator+ (const cadena&, const cadena&);
    friend cadena operator+ (const cadena&, const char* );
    friend cadena operator+ (const char*, const cadena&);
    friend int operator== (const cadena&, const cadena&);
    friend int operator!= (const cadena&, const cadena&);
    friend ostream& operator<< (ostream&, const cadena&);

/*    Otros operadores que se podrían sobrecargar:
    friend int operator== (const cadena&, const char*);
    friend int operator== (const char*, const cadena&);
    friend int operator!= (const cadena&, const char*);
    friend int operator!= (const char*, const cadena&);
*/

};
#endif // __CADENA_H
```

Obsérvese que los operadores sobrecargados se declaran de forma muy similar a la de las funciones, sustituyendo el nombre de la función por la palabra ***operator*** y el ***operador correspondiente***.

En la declaración de la clase ***cadena*** se pueden observar algunos hechos interesantes, que se comentan a continuación. Recuérdese que la declaración de una clase es de ordinario lo único que conocen los *programadores-usuarios* de la clase: al código, esto es, a la definición de las funciones y operadores sólo acceden los que programan la clase. Lo importante es resaltar que la ***declaración de la clase contiene toda la información necesaria*** para utilizarla y sacarle partido. Los aspectos a destacar son los siguientes:

1. En la definición de la clase no se ha reservado memoria para la cadena de caracteres, sólo para el puntero ***pstr***. La razón es que no se sabe a priori cuántos caracteres va a tener cada objeto de la clase ***cadena*** y se piensa por ello utilizar reserva dinámica de memoria: cuando se sepa el

texto que se quiere guardar en un objeto determinado, se reservará la memoria necesaria para ello.

2. Se han definido ***tres constructores*** y un ***destructor***. El primer constructor es un ***constructor por defecto*** que no requiere ningún argumento, pues inicializa el objeto a una cadena vacía de cero caracteres. Al segundo constructor se le pasa como argumento un puntero a una cadena de caracteres cualquiera y crea un objeto que apunta a esa cadena. El tercer constructor es un ***constructor de copia*** que recibe como argumento una ***referencia constante a un objeto de la clase cadena***. El argumento -un objeto de la clase ***cadena***- se pasa ***por referencia*** por motivos de eficiencia (para no tener que sacar una copia); por otra parte se pasa como ***const*** por motivos de seguridad (para evitar que sea modificado por el constructor). En lo sucesivo ya no se volverá a justificar el paso ***por referencia*** y como ***const*** de los argumentos que no deban ser modificados. En este caso, los ***constructores de oficio*** no sirven, pues ya se han definido otros constructores y además ***una variable miembro es un puntero*** y se va a utilizar ***reserva dinámica de memoria***.

3. Se ha definido un ***destructor*** porque las cadenas de caracteres son ***arrays*** y se va a utilizar ***reserva dinámica de memoria***, memoria que habrá que liberar expresamente.

4. La función miembro ***setcad()*** permite proporcionar o cambiar el valor de la cadena de caracteres que contiene un objeto. Es una función miembro típica y sólo se ha introducido aquí a modo de ejemplo. Se le pasa como argumento un puntero a ***char*** que contiene la dirección del primer carácter del texto a introducir. No necesita devolver ningún valor.

5. Se han definido ***siete operadores sobrecargados*** -uno como ***miembro*** y seis como ***friends***-, y hay *cuatro operadores relacionales* más incluidos entre comentarios que se podrían terminar de definir de modo similar.

6. El primer operador sobrecargado es el ***operador de asignación*** (=). Como es un operador binario que modifica el primer operando ***debe necesariamente ser definido como miembro***. Este operador asigna un objeto ***cadena*** a otro. El miembro izquierdo de la igualdad es el primer operando y en este caso es el argumento implícito del operador. En la lista de argumentos formales que figura entre paréntesis sólo hay que incluir el segundo operando, que es un objeto cadena que se pasa ***por referencia*** y como ***const***, pues no debe ser modificado. El ***valor de retorno*** de este operador requiere un comentario especial.

7. Estrictamente hablando, en este caso el ***operador de asignación (=) no necesita ningún valor de retorno***: su misión en una sentencia tal como ***c1 = c2;*** queda suficientemente cumplida si hace que las variables miembro de ***c1*** sean iguales a las de ***c2*** (siendo ***c1*** y ***c2*** dos objetos de la case ***cadena***). Sin embargo el ***operador (=) estándar de C*** tiene un valor de retorno que es una referencia al resultado de la asignación. Esto es lo que permite escribir sentencias como la siguiente: ***a = b = c;*** que es equivalente a hacer ***b*** igual a ***c***, y devolver un valor que puede ser asignado a ***a***. Al final las tres variables tienen el mismo valor. Para que el operador sobrecargado se parezca todo lo posible al de C y para poder escribir sentencias de asignación múltiples con objetos de la clase ***cadena*** (***c1 = c2 = c3;***), es necesario que el operador de asignación sobrecargado (=) tenga valor de retorno. El ***valor de retorno es una referencia al primer operando*** por motivos de eficiencia, pues si no hay que crear un objeto nuevo, diferente de los dos operandos, para devolverlo como valor de retorno.

8. A continuación aparecen tres ***operadores suma*** (+) sobrecargados para realizar ***concatenación de cadenas***. Puede observarse que ***no son operadores miembro*** sino ***friend***. Así pues, no hay argumento implícito, y los dos argumentos aparecen entre paréntesis. Los tres operadores (+) tienen un objeto de la clase ***cadena*** como valor de retorno. El primero de ellos ***concatena dos objetos*** de la clase ***cadena***, y devuelve un nuevo objeto ***cadena*** cuyo texto es la concatenación

de las cadenas de caracteres de los objetos operandos. Este resultado es diferente del de la función ***strcat()***, que añade el texto del segundo argumento sobre el primer argumento (modificándolo por tanto). En este caso se ha preferido obtener un nuevo objeto y no modificar los operandos. Los otros dos operadores (+) sobrecargados concatenan el texto de un ***objeto cadena*** y una ***cadena de caracteres estándar***, y una ***cadena de caracteres estándar*** y el texto de un ***objeto cadena***, respectivamente. Recuérdese que el orden y el tipo de los argumentos deciden qué definición del operador sobrecargado se va a utilizar. Si se quiere poder escribir tanto ***c1+" anexo"*** como ***"prefacio "+c2*** es necesario programar dos operadores distintos, además del que concatena dos objetos ***c1+c2***.

9. ¿Por qué los operadores (+) anteriores se han definido como ***friend*** y no como ***miembros***? La verdad es que ésta es una cuestión interesante y no fácil de ver a primera vista. En realidad, la primera y la segunda de las definiciones podrían haberse hecho con operadores miembro. Sin embargo, la tercera no puede hacerse más que como ***friend***. ¿Por qué? Pues porque ***los operadores miembro siempre tienen un primer operando que es miembro de la clase*** y que va como ***argumento implícito***. La tercera definición del operador (+) ***no cumple*** con esta condición, pues su primer operando (para concatenar por ejemplo ***"prefacio "+c2***) es una simple cadena de caracteres. Este hecho es muy frecuente cuando se sobrecargan los operadores aritméticos: puede por ejemplo sobrecargarse el operador producto (*) para pre y post-multiplicar matrices por un escalar. Al menos en el caso de la pre-multiplicación por el escalar es necesario que el operador (*) no sea miembro de la clase matriz.

10. A continuación figura la declaración de los ***operadores relacionales*** (==) y (!=), que permiten saber si dos objetos contienen o no el mismo texto. En este caso el valor de retorno del ***test de igualdad*** (==) es un ***int*** que representará ***true*** (1) o ***false*** (0). Para el ***operador de desigualdad*** (!=) la situación es un poco más compleja, pues se desea que sirva para ordenar cadenas alfabéticamente, de modo similar a la función ***strcmp()***. Por ello el valor de retorno de ***c1!=c2*** es un (-1) si ***c1*** es diferente y anterior alfabéticamente a ***c2***, un cero (0 ó ***false***) si las cadenas son idénticas, y un (1) si son diferentes y ***c1*** es posterior a ***c2*** en orden alfabético.

11. Los ***operadores relacionales*** cuyas declaraciones están contenidas entre comentarios /*…*/ ***no son necesarios***. No hay inconvenientes en comparar objetos de la clase ***cadena*** y cadenas de caracteres estándar, aunque no se disponga de los operadores relacionales cuyos argumentos se ajusten a los tipos exactos. La razón está en la ***inteligencia*** contenida en los ***compiladores de C++***: cuando el compilador encuentra un operador (==) o (!=) que relaciona una ***cadena estándar*** y un objeto de la clase ***cadena***, intenta ver si dispone de alguna forma ***segura*** de ***promover o convertir*** la ***cadena estándar*** a un objeto de la clase ***cadena***. Como tiene un ***constructor general*** que inicializa un objeto ***cadena*** a partir de una cadena estándar, utiliza dicho constructor para convertir la cadena estándar en un objeto ***cadena*** y luego poder utilizar la definición del operador relacional que compara dos objetos de la clase ***cadena***. Esto lo hace independientemente de qué operando es el objeto y cuál es la cadena estándar. Esto es similar a lo que hace el ***compilador de C*** cuando se le pide que compare un ***int*** con un ***double***: antes de hacer la comparación ***promueve*** (*convierte*) el ***int*** a ***double***, y luego realiza la comparación entre dos variables ***double***.

12. La pregunta obligada es, ***¿y no pasa lo mismo con el operador (+)***? ¿No bastaría con definir un único operador (+) que concatenase ***objetos de la clase cadena*** y confiar a la inteligencia del compilador el ***promover a objetos*** las cadenas de caracteres estándar que se quisieran concatenar, ya fueran el primer o el segundo operando? La respuesta es que sí: en este caso ***no hace falta sobrecargar el operador (+) con tres definiciones***. Se ha hecho porque el problema se resuelve de modo ***más eficiente*** (no hay que perder tiempo en promover variables creando objetos) y porque en otras aplicaciones más complicadas que las cadenas de caracteres lo de la

promoción de una variable estándar a un objeto de una clase puede que no esté nada claro. Por ejemplo, en la clase ***matriz***, ¿cómo se promueve un escalar a matriz? Si es para la ***operación suma*** se podría hacer con una matriz cuyos elementos fueran todos igual al escalar, pero si es para la ***operación producto*** sería más razonable promover el escalar a una matriz diagonal. En definitiva, C++ ofrece posibilidades muy interesantes, pero no conviene abusar de ellas.

13. Finalmente, en la declaración de la clase ***cadena*** se muestra la sobrecarga del operador de ***inserción en el flujo de salida*** (<<), que tiene como finalidad el poder utilizar ***cout*** con objetos de la clase. Este operador, al igual que el (>>), se suele sobrecargar como operador ***friend***. Recibe dos argumentos: Una referencia al flujo de salida (***ostream***, de *output stream*) y una referencia constante al objeto ***cadena*** que se desea insertar en dicho flujo. El ***valor de retorno*** es una referencia al flujo de salida ***ostream*** en el que ya se habrá introducido el objeto ***cadena***.

3.12.2 DEFINICIÓN DE FUNCIONES Y OPERADORES DE LA CLASE CADENA

Una vez vistas y explicadas la declaración de la clase cadena y de sus funciones y operadores ***miembro*** y ***friend***, contenidas en el fichero ***cadena.h***, se va a presentar la definición de todas estas funciones y operadores, que están contenidas en otro fichero llamado ***cadena.cpp***. Obsérvese que se han introducido algunas ***sentencias de escritura de mensajes***, con objeto de determinar con claridad cuándo y en qué circunstancias se ejecuta cada función.

Los usuarios de la clase ***cadena*** no tendrían por qué tener acceso a este fichero fuente, aunque sí al fichero objeto correspondiente. A continuación se muestra el contenido del fichero ***cadena.cpp*** intercalando algunos comentarios explicativos.

```
// cadena.cpp

#include <string.h>
#include "cadena.h"

// constructor por defecto
cadena::cadena()
{
    pstr = new char[1];
    strcpy(pstr, "");
    nchar = 0;
    cout << "Constructor por defecto " << (long)this << endl;
}
```

Dos observaciones acerca del ***constructor por defecto*** (sin argumentos). La primera es que la variable miembro ***psrt***, que es un puntero a *char*, se inicializa apuntando a una cadena vacía (sólo tiene el '\0' de fin de cadena). Se utiliza reserva dinámica de memoria con el operador ***new***. La variable miembro ***nchar***, que representa el número de caracteres, no incluye el carácter de final de cadena. Se ha incluido una sentencia de escritura que imprimirá un mensaje avisando de que se ha utilizado este constructor. Imprime también la ***dirección en memoria del objeto creado***, con idea de saber cuándo se crea y se destruye cada objeto concreto del programa. Para ello se utiliza el puntero ***this*** y un ***cast*** a ***long***.

```
// constructor general
cadena::cadena(char* c)
{
    nchar = strlen(c);
    pstr = new char[nchar + 1];
    strcpy(pstr, c);
    cout << "Constructor general     " << (long)this << endl;
}
```

El ***constructor general*** admite como argumento la dirección del carácter inicial de una cadena de caracteres, a partir de la cual se inicializará el nuevo objeto. Se utiliza también reserva dinámica

de memoria. Se utilizan las funciones ***strlen()*** y ***strcpy()*** para determinar el número de caracteres y para copiar el argumento en la dirección a la que apunta la variable miembro ***pstr***. Se imprime un mensaje que permite saber qué constructor se ha llamado y qué objeto ha sido creado.

```
// constructor de copia
cadena::cadena(const cadena& cd)
{
    nchar = cd.nchar;
    pstr = new char[nchar +1];
    strcpy(pstr, cd.pstr);
    cout << "Constructor de copia    " << (long)this << endl;
}
```

Puede verse que el ***constructor de copia*** es muy similar al ***constructor general***, con la única diferencia de que recibe como argumento una referencia a un objeto ***cadena***, a partir del cual inicializa el nuevo objeto con reserva dinámica de memoria. Obsérvese que se tiene acceso a las variables miembro del objeto cadena pasado como argumento, pero que hay que utilizar el operador punto (.). A las variables miembro del objeto pasado como argumento implícito se tiene acceso directo.

```
// destructor
cadena::~cadena()
{
    delete [] pstr;
    cout << "Destructor "           << (long)this << endl;
}
```

En este caso no sirve el destructor de oficio, porque se está utilizando reserva dinámica de memoria. El destructor debe liberar la memoria ocupada por las cadenas de caracteres, para lo que hay que utilizar la sentencia ***delete [] pstr;***. Además se imprime un mensaje incluyendo la dirección del objeto borrado. De este modo se puede saber cuándo se crea y se destruye cada objeto, lo cual será de gran utilidad en los ejemplos que se presentarán más adelante.

```
// función miembro setcad()
void cadena::setcad(char* c)
{
    nchar = strlen(c);
    delete [] pstr;
    pstr = new char[nchar + 1];
    strcpy(pstr, c);
    cout << "Función setcad()" << endl;
}
```

La función miembro ***setcad()*** permite sustituir el contenido de un objeto ***cadena*** a partir de una cadena de caracteres estándar. Esta función se diferencia del constructor general visto previamente en que actúa sobre un objeto que ya existe. Esto hace que la primera tarea a realizar sea liberar la memoria a la que la variable ***pstr*** apuntaba anteriormente. Después se reserva memoria para el nuevo contenido al que ***pstr*** apuntará. Los constructores siempre actúan sobre un objeto recién creado, y por eso no tienen que liberar memoria.

```
// operador de asignación sobrecargado (=)
cadena& cadena::operator= (const cadena& cd)
{
    if(*this != cd) {
        nchar = cd.nchar;
        delete [] pstr;
        pstr = new char[nchar + 1];
        strcpy(pstr, cd.pstr);
        cout << "Operador =" << endl;
    }
    return *this;
}
```

Este operador miembro permite asignar un objeto ***cadena*** a otro, por ejemplo en la forma ***c2=c1;***, donde se supone que ambos objetos existían previamente. El objeto ***c2*** sería el argumento implícito y ***c1*** el que se pasa explícitamente por ventana. Por eso a las variables miembro de ***c2*** se accede directamente, mientras que a las de ***c1*** -representado por ***cd***- hay que acceder con el operador punto (.). Como ***c1*** ya existía, hay que liberar la memoria que estaba ocupando. Este operador se define con un ***valor de retorno*** que es una ***referencia*** a un objeto ***cadena*** para poderlo utilizar en asignaciones múltiples (***c3=c2=c1;***). Como valor de retorno se devuelve el propio miembro izquierdo de la asignación, que es el argumento implícito del operador miembro; para hacer referencia a dicho objeto hay que utilizar el puntero ***this*** (****this*** es el objeto, mientras que ***this*** es su dirección).

En la definición del ***operador miembro*** (=) llama la atención la sentencia ***if*** que aparece al comienzo de la función. ¿Cuál es su razón de ser? Su razón de ser es evitar los efectos perjudiciales que puede tener una sentencia de ***asignación de un objeto a sí mismo*** (***c1=c1;***). Esta sentencia no es verdaderamente muy útil, pero no hay razón para no prever las cosas de modo que sea ***inofensiva***; y la verdad es que si no se introduce ese ***if*** esta sentencia es todo menos inofensiva. Al asignarse un objeto a otro, lo primero que se hace es liberar la memoria ocupada por el primer operando (el que está a la izquierda), para después sacar una copia de la memoria a la que apunta el segundo operando y asignar su dirección a la variable miembro del primero. El problema es que si ambos objetos coinciden (son el mismo objeto), al liberar la memoria del primer operando, el segundo (que es el mismo) la pierde también y ya no hay nada para copiar ni para asignar, llegándose en realidad a la destrucción del objeto. El remedio es chequear, a la entrada de la definición del ***operador*** (=), que los dos operandos son realmente objetos distintos. Una vez más, es necesario utilizar el puntero ***this***.

```
// operador de inserción en ostream
ostream& operator<< (ostream& co, const cadena& cad) {
    co << cad.pstr;
    return co;
}
```

La definición del ***operador inserción*** (<<) sobrecargado es muy sencilla, pues lo único que se hace es insertar en el flujo de salida la cadena de caracteres estándar a la que apunta la variable miembro ***pstr***.

```
// Definiciones del operador de concatenación de cadenas
cadena operator+ (const cadena& a, const cadena& b)
{
    cadena c;
    c.nchar = a.nchar + b.nchar;
    c.pstr = new char[c.nchar + 1];
    strcpy(c.pstr, a.pstr);
    strcat(c.pstr, b.pstr);
    return c;
}

cadena operator+ (const cadena& a, const char* ch)
{
    cadena c;
    c.nchar = a.nchar + strlen(ch);
    c.pstr = new char[c.nchar + 1];
    strcpy(c.pstr, a.pstr);
    strcat(c.pstr, ch);

    return c;
}
```

```
cadena operator+ (const char* ch, const cadena& b)
{
    cadena c;
    c.nchar = strlen(ch) + b.nchar;
    c.pstr = new char[c.nchar + 1];
    strcpy(c.pstr, ch);
    strcat(c.pstr, b.pstr);
    return c;
}
```

Las tres definiciones anteriores del ***operador de concatenación*** (+) son casi evidentes. En todos los casos se crea un nuevo objeto cadena en el que se concatenan las cadenas de los dos operandos. Se podría haber realizado la programación de modo que el segundo operando se añadiera al primero (de modo semejante a como actúa la función ***strcat()***). Se ha preferido hacerlo así para dejar intactos los operandos, y porque la otra forma de actuar sería más propia del operador de ***asignación incremental*** (+=).

```
// sobrecarga de los operadores relacionales
int operator== (const cadena& c1, const cadena& c2)
{
    if(strcmp(c1.pstr, c2.pstr)==0)
        return 1;
    return 0;
}

int operator!= (const cadena& c1, const cadena& c2)
{
    int dif = strcmp(c1.pstr, c2.pstr);
    if(dif<0)
        return (-1);
    if(dif==0)
        return 0;
    else
        return 1;
}
// fin del fichero cadena.cpp
```

La sobrecarga de los ***operadores relacionales*** está muy clara y no requiere explicaciones distintas de las que se han dado al hablar de su declaración.

3.12.3 EJEMPLO DE UTILIZACIÓN DE LA CLASE CADENA

Una vez vistas todas las definiciones contenidas en el fichero ***cadena.cpp***, queda por ver un programa principal que de alguna manera utilice las capacidades que se han explicado: Un programa principal posible, contenido en un fichero llamado ***pruebacad.cpp***, se muestra a continuación:

```
// fichero pruebacad.cpp

#include "cadena.h"

void main(void)
{
    // creación de un objeto con constructor por defecto
    cadena c1;
    // creación de un objeto con constructor general
    cadena c2("Ingenieros");
    // operaciones de concatenación
    c2 = (c2 + " ") + "Industriales ";
    // creación de un objeto con constructor de copia
    cadena c3 = c2; // también: cadena c3(c2);

    // primero se convierte a objeto y luego se asigna
    c1 = "Escuela Superior de ";

    // c3 se hace primero igual a c2 y luego a c1
    (c3 = c2) = c1;
    if(c3 == c1) {
        cout << "c3 y c1 son iguales" << endl;
        // un contructor es un conversor de tipo
        c3 = cadena("de San Sebastian");
    }
    // salida de reultados
    cout << "\nc1 vale: " << c1 << "\nc2 vale: " << c2
        << "\nc3 vale: " << c3 << "\nAgur" << endl;
    cout << c1+c2+c3 << endl;
}   // se detruyen c1, c2 y c3
```

Es muy interesante ver cuál es la ***salida*** que produce el anterior ***programa principal***, que se muestra un poco más adelante. Dicha salida consta de los mensajes del propio programa principal y de los que producen los ***constructores***, el ***destructor*** y el ***operador*** (=) cada vez que son llamados (la mayoría de los mensajes proviene de ellos). Se puede observar que hay más llamadas a los constructores y al destructor de las que cabría pensar. Para identificar estas llamadas se ha incluido en **negrita** el código del programa principal, que aparece justo antes de que se llame, es decir, justo antes de que se muestren los mensajes que esa sentencia genera. Se puede concluir que ***la facilidad que C++ da a los programadores tiene un precio en términos de eficiencia en los cálculos***. Esta eficiencia puede mejorarse utilizando siempre que sea posible ***referencias a objetos***, evitando así llamadas a los ***constructores*** y al ***destructor***.

Nótese que cada llamada al ***operador (+)*** implica la creación de un objeto con el ***constructor por defecto*** para crear el objeto resultante de la concatenación, y luego una llamada al ***constructor de copia*** y dos llamadas al ***destructor*** para devolver el objeto resultante como valor de retorno. En efecto, ***para devolver un objeto como valor de retorno***, C++ crea un ***objeto invisible*** con el ***constructor de copia***, destruye el objeto creado anteriormente con el ***constructor por defecto*** y luego destruye el ***objeto invisible*** creado con el ***constructor de copia***. El resultado del programa principal anterior es el siguiente:

```
cadena c1;
Constructor por defecto 1245036
cadena c2("Ingenieros");
Constructor general     1245028
c2 = (c2 + " ") + "Industriales ";
Constructor por defecto 1244916
Constructor de copia    1245012
Destructor           1244916
Constructor por defecto 1244920
Constructor de copia    1245004
Destructor           1244920
```

```
Operador =
Destructor           1245004
Destructor           1245012
cadena c3 = c2;
Constructor de copia    1245020
c1 = "Escuela Superior de ";
Constructor general     1244996
Operador =
Destructor           1244996
(c3 = c2) = c1;
Operador =
if(c3 == c1) {
cout << "c3 y c1 son iguales" << endl;
c3 y c1 son iguales
c3 = cadena("de San Sebastian");
Constructor general     1244988
Operador =
Destructor           1244988
cout << "c1 vale: " ...
c1 vale: Escuela Superior de
c2 vale: Ingenieros Industriales
c3 vale: de San Sebastian
Agur
cout << c1+c2+c3 << endl;
Constructor por defecto 1244912
Constructor de copia    1244980
Destructor           1244912
Constructor por defecto 1244916
Constructor de copia    1244972
Destructor           1244916
Escuela Superior de Ingenieros Industriales de San Sebastian
Destructor           1244972
Destructor           1244980
Destructor           1245020
Destructor           1245028
Destructor           1245036
```

3.12.4 SOBRECARGA DE LOS OPERADORES (++) Y (--)

Los operadores de ***incremento*** (++) y ***decremento*** (--) son un caso especial en C++. Ambos son operadores unarios que siempre se definen como ***miembros*** de una clase. En realidad estos operadores tienen dos significados, según se ***antepongan*** o se ***postpongan*** al nombre de la variable o del objeto al que se aplican. Esto crea una dificultad de definición, porque ¿a cuál de los dos significados se refiere la siguiente declaración?:

```
cadena& operator++ ();
```

En sí no hay nada en esta declaración que indique a cuál de los dos significados de este operador se refiere el programador. Por eso C++ utiliza un ***convenio especial***: la declaración anterior se refiere al ***operador (++) antepuesto***, mientras que la siguiente declaración:

```
cadena& operator++ (int j);
```

se refiere al ***operador (++) postpuesto***. La variable j tiene valor 0 y no se utiliza más que como criterio de distinción. Lo mismo sucede con el operador (--).

3.13 Objetos miembro de otros objetos.

Una ***clase***, a semejanza de una estructura, puede contener ***variables miembro*** que sean ***objetos*** de otra ***clase*** definida por el usuario. Ésta es una relación de ***pertenencia*** que no tiene nada que ver con la ***herencia***, que se explicará en el capítulo siguiente. El ***constructor*** de la clase que contiene objetos de otras clases debe llamar a los ***constructores de los objetos contenidos***, si no se quiere que se utilicen los ***constructores por defecto***. Esta llamada es muy similar a la de los constructores de las clases derivadas, como se verá más adelante. Considérese el siguiente ejemplo:

```
Clase_Continente::Clase_Continente (Lista de argumentos)
    : Clase_Contenida1 (Lista de argumentos),
          Clase_Contenida2 (Lista de argumentos), otros inicializadores
    {
         Asignación de otras variables;
    }
```

Los ***constructores*** de las clases contenidas se llaman de la misma forma que los ***inicializadores*** de las variables miembro ordinarias: después del carácter dos puntos (:) que aparece tras los argumentos, separados por comas, y antes del bloque del constructor. Puede considerarse que los ***constructores*** no son más que ***inicializadores*** especializados.

A continuación se va a presentar un ejemplo. La clase ***persona*** tiene tres variables miembro privadas: el ***nombre***, el ***número de DNI*** y la ***fecha de nacimiento***. A su vez la fecha de nacimiento es un objeto de la clase ***Date***, que tiene tres variables miembro: el ***día***, ***mes*** y ***año*** de nacimiento. Como la fecha de nacimiento no se puede cambiar, el correspondiente objeto en la clase ***persona*** se ha declarado como ***const***.

La clase ***Date*** dispone de funciones miembro públicas para obtener el día, mes y año de cualquier objeto de dicha clase, así como de funciones para cambiar esas variables miembro (que no se pueden utilizar con un objeto ***const***). La declaración de la clase ***Date*** con funciones ***inline*** es como sigue:

```
// fichero date.h

// definición de la clase Date
class Date {
private:
    int day;
    int month;
    int year;
public:
    // constructor con valores por defecto
    Date(int dia=1, int mes=1, int anio=1900) :
          day(dia), month(mes), year(anio) {}
    int getDay() const { return day; }
    int getMonth() const { return month; }
    int getYear() const { return year; }
    // función para calcular los años completos entre dos fechas
    friend int ElapsedYears(const Date&, const Date&);
    //    Si el objeto no es const, añadir estas funciones miembro:
    void setDay(int dia) { day=dia; }
    void setMonth(int mes) { month=mes; }
    void setYear(int anio) { year=anio; }
};
```

Las funciones ***getDay()***, ***getMont()*** y ***getYear()*** han sido declaradas como ***const***. Esto quiere decir que son funciones que no modifican las variables miembro, es decir ***funciones de sólo lectura***. La palabra ***const*** debe figurar tanto en la *declaración* como en la *definición*, después de cerrar el paréntesis de los argumentos y antes del cuerpo de la función.

A continuación se incluye la declaración de la clase ***persona***, que incluye también tres funciones miembro ***inline*** para acceder a las variables miembro privadas. Obsérvese que la fecha de nacimiento se ha definido como ***const*** (resaltado con negrita en el listado de la definición),

```
// fichero persona.h

#include <iostream.h>
#include "date.h"

class persona {
private :
    char  nombre[25];
    long  DNI;
    const Date fechaNac;
public:
    // constructor
    persona(char*, long, Date);
    char* getNombre(void)
         { return nombre; }
    long getDNI(void)
         { return DNI; }
    Date getFechaNac(void)
         { return fechaNac; }
};
```

El fichero ***persona.cpp*** que se muestra a continuación contiene la definición del ***constructor*** de la clase ***persona***. Dicho constructor tiene valores por defecto para los tres argumentos. Se puede observar que tanto el ***DNI*** como la ***fecha de nacimiento*** toman valor por medio de inicializadores y no de sentencias de asignación. En el caso del ***DNI*** esto es opcional, pero en la ***fecha de nacimiento*** es obligatorio ya que es un objeto ***const*** que no puede figurar a la izquierda en una sentencia de asignación. Nótese también que el tercer argumento se puede inicializar a partir de tres valores ya que existe un constructor ***Date*** que puede tomar estos parámetros. Es importante notar la forma en la que C++ hace definir el tercer argumento y del segundo inicializador,

```
// Fichero persona.cpp

#include "persona.h"
#include <string.h>

// definición del constructor con valores por defecto
persona::persona(char* name = "", long dni = 0, Date birth=(1,1,1970) )
        // inicializadores
       : DNI(dni), fechaNac(birth)
{
    strcpy(nombre, name);
}
```

Finalmente se incluye el fichero ***lista2.cpp***, que contiene un programa principal que hace uso de las clases ***Date*** y ***persona***. Se definen dos personas, una definiendo directamente un objeto y la otra mediante un puntero y reserva dinámica de memoria. Obsérvese cómo se pasa al ***constructor*** de ***persona*** la fecha de nacimiento. En el mismo fichero que el programa principal está la función ***ElapsedYears()*** que calcula los ***años completos*** transcurridos entre dos objetos ***Date*** que se le pasan como argumentos. Como esta función se ha declarado como función ***friend***, su definición se ha incluido en el mismo fichero que el programa principal.

```
// fichero lista2.cpp

#include <iostream.h>
#include "persona.h"
int ElapsedYears(const Date& hoy, const Date& birth);

void main(void)
{

    // ver cómo se inicializa un objeto Date que es const
    persona* m = new persona("Maria",   26429764, Date(1, 2, 1980));
    persona  i("Ignacio", 25190578, Date(13, 4, 1992));

    Date hoy(13,4,1998);
    cout << m->getNombre() << " tiene " <<
        ElapsedYears(hoy, m->getFechaNac()) <<
            " agnos" << endl;
    cout << i.getNombre() << " tiene " <<
            ElapsedYears(hoy, i.getFechaNac()) <<
            " agnos" << endl;

    cout << "Ya he terminado." << endl;
}

// Función para calcular los años completos entre dos fechas
int ElapsedYears(const Date& hoy, const Date& birth)
{
    int years = hoy.year - birth.year;
    if ((birth.month <= hoy.month) && (birth.day <= hoy.day))
        return years;
    else
        return (years-1);
}
```

3.14 Variables miembro *static*

Cada ***objeto*** de una clase tiene su ***propia copia*** de cada una de las variables miembro de esa clase. Sin embargo, a veces puede interesar que ***una variable miembro sea común*** para todos los objetos de la clase, de modo que todos compartan el mismo valor. Por ejemplo, puede resultar útil que el interés ofrecido por un banco sea igual para todas las cuentas bancarias de un mismo tipo. En tal caso se puede utilizar una variable miembro ***static*** como en el ejemplo siguiente:

```
class C_Cuenta {
    // Variables miembro
    private:
        char *Nombre;        // Nombre de la persona
        double Saldo;        // Saldo Actual de la cuenta
    public:
        static double Interes;    // Esta variable es la misma en todos los
                                  //    objetos creados de C_Cuenta

// ... Definición de los métodos
};

// Las variables static necesitan inicializarse. Si no se especifica un
//     valor inicial las variables static se inicializan a cero.
//     double C_Cuenta::Interes;
//     para inicializaras con otro valor: double C_Cuenta::Interes=1.0;
```

```
void main()
{
    // Las variables estáticas tienen un scope global.
    // Por lo tanto se pueden usar aunque no esté creado
    //    ningún objeto de dicha clase.
    C_Cuenta::Interes = 2.0;
    // ...
}
```

Como se ve en el ejemplo anterior, para conseguir que el interés sea el mismo en todas las cuentas que se creen, C++ permite declarar una variable miembro como ***static***. A continuación se describen las características más importantes de las variables miembro ***static***:

- Sólo existe una copia de cada una de las variables miembro ***static***. Es decir, todos los objetos declarados de esa clase hacen referencia a la misma variable, esto es, a la misma posición de memoria.
- Las variables ***static*** pueden ser ***public*** o ***private***, del mismo modo que el resto de las variables miembro.
- Las variables ***static*** de una clase existen aunque no se haya declarado ningún objeto de esa clase. Esto quiere decir que la memoria reservada para este tipo de variable se ocupa en el momento de la definición de la clase, no en el momento de la declaración de los objetos.
- Para referirse a una variable ***static*** se puede utilizar el nombre de un objeto y el operador punto (.), pero esta notación es confusa ya que se está haciendo referencia a una variable común a todos los objetos de una clase mediante el nombre de uno sólo de ellos. Por eso es mejor utilizar el nombre de la clase y el ***scope resolution operator*** (::):
  - `Nombre_de_la_clase::variable_static`
- Las variables ***static*** no se pueden inicializar en un ***constructor***, porque se inicializarían muchas veces. Si se desea inicializarlas debe hacerse *en el fichero que contiene las definiciones de las funciones miembro de esa clase*:
  - `float Nombre_de_la_clase::variable_static = valor_inicial;`

  Obsérvese que en este caso hay que incluir el ***tipo de variable*** porque se trata de una ***inicialización*** y no de una sentencia de ***asignación***.

Las variables ***static*** tienen una cierta similitud con las variables ***global***, pero difieren en que el ***scope*** de las variables miembro ***static*** puede ser controlado por el programador definiéndolo como ***private***, ***public*** o ***protected***.

Un caso habitual en el que las variables tipo ***static*** son de gran utilidad es en el caso de que se desee llevar un ***contador del número de objetos creados*** de una clase dada:

```
class C_Cuenta {
    // Variables miembro
    private:
        char *Nombre;       // Nombre de la persona
        double Saldo;       // Saldo Actual de la cuenta
        double Interes;     // Interés calculado hasta el momento
        static int Cuantas; // Número de cuentas abiertas
    public:
```

```
        // Constructor
        C_Cuenta(const char *unNombre, double unSaldo=0.0, double unInteres=0.0)
        {
            Nombre = new char[strlen(unNombre)+1];
            strcpy(Nombre, unNombre);
            SetSaldo(unSaldo);
            SetInteres(unInteres);
            Cuantas++;
        }
        // Destructor
        ~Cuenta()
            { delete []Nombre;
              Cuantas--; }
        // Métodos
        inline int getCuantasCuentas()
            { return Cuantas; }
        inline char *GetNombre()
            { return Nombre; }
        inline double GetSaldo()
            { return Saldo; }
        inline double GetInteres()
            { return Interes; }
        inline void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        inline void SetInteres(double unInteres)
            { Interes = unInteres; }
        ivoid Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
        friend ostream& operator<<(ostream& os, C_Cuenta& unaCuenta)
        {
            os << "Nombre=" << unaCuenta.GetNombre() << endl;
            os << "Saldo=" << unaCuenta.GetSaldo() << endl;
            return os;
        }
};

// definición e inicialización de variable static
int C_Cuenta::Cuantas = 0;

void main(void)
{
    C_Cuenta c1("Igor");
    // Imprime 1
    cout << c1.getCuantasCuentas() << endl;
    C_Cuenta c2("Juan");
    // Imprime 2
    cout << c2.getCuantasCuentas() << endl;
    // ...
}
```

Se ve en el ejemplo anterior cómo el número total de cuentas creadas hasta el momento se almacena en una variable miembro ***static*** llamada ***cuantas***. Esta variable se incrementa en una unidad cada vez que se llama al ***constructor*** y se decrementa en uno cada vez que se llama al ***destructor***. Para conocer en un momento dado el valor de esta variable se utiliza la función ***getCuantasCuentas()***. En el ejemplo anterior, se ha añadido también a la clase ***C_Cuenta*** un ***operador de inserción en ostream*** sobrecargado (<<) capaz de escribir todos los datos de una cuenta corriente.

3.15 Funciones miembro *static*

Las funciones miembro que ***sólo acceden*** a variables miembro ***static*** pueden, a su vez, ser declaradas como ***static*** en la definición de la clase. Estas funciones pueden ser llamadas indiferentemente con el ***operador punto*** (.) y con el ***scope resolution operator*** (::).

Algunas de las características más importantes de este tipo de funciones se comentan a continuación:

- Son ***funciones genéricas*** que no actúan sobre ningún objeto concreto de la clase.
- Como ya se dicho anteriormente, no pueden utilizar variables miembro que no sean ***static***.
- No pueden utilizar el puntero ***this***, ya que éste hace referencia al objeto concreto de la clase sobre la que se está trabajando.

En resumen, hay que decir que las ***variables*** y ***funciones*** miembro ***static*** resultan útiles en el caso de que se quieran establecer variables y métodos ***comunes a todos los objetos*** de una clase. Se puede decir que se necesita una variable de tipo ***static*** cuando se quiere almacenar el valor de una característica que pertenece más a la "fábrica" que crea los objetos que a cada uno de los objetos creados. Sirva como ejemplo el caso ya citado de que se quieran contar el número de objetos de una clase que existen en un momento determinado. Para ello, como ya se ha visto, se puede utilizar el operador incremento (++) en el ***constructor*** y el operador decremento (--) en el ***destructor*** de esa clase.

Las funciones ***static*** pueden recibir objetos de su clase como argumentos explícitos, aunque no como argumento implícito. Esto implica que cuando se desea que una función actúe sobre dos objetos de una clase (por ejemplo para hacer una trasferencia entre dos cuentas bancarias), las funciones ***static*** son una alternativa a las funciones ***friend*** para conseguir simetría en la forma de tratar a los dos objetos de la clase (que ambos pasen como argumentos explícitos). Considérese el siguiente ejemplo, válido en C++:

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

class persona {
public:
    long DNI;
    char nombre[41];

    persona(long dni, char *name) {
        DNI = dni;
        strcpy(nombre, name);
    }

    // funcion static con argumento persona
    static long getDNI(persona P) {
        return P.DNI;
    }
};


void main (void)
{
    persona p1(47126578, "Pepe Perez");
    // se llama a la función static con el nombre de la clase
    cout << p1.nombre << " DNI: " << persona::getDNI(p1) << endl;
    cout << "Ya he terminado." << endl;
}
```

4 HERENCIA

4.1 Necesidad de la herencia

La mente humana clasifica los *conceptos* de acuerdo a dos dimensiones: ***pertenencia*** y ***variedad***. Se puede decir que el Ford Fiesta es un tipo de coche (*variedad* o, en inglés, una relación del tipo ***is a***) y que una rueda es parte de un coche (*pertenencia* o una relación del tipo ***has a***). Antes de la llegada de la ***herencia***, en C ya se había resuelto el problema de la ***pertenencia*** mediante las ***estructuras***, que podían ser todo lo complejas que se quisiera. Con la ***herencia***, como se va a ver en este capítulo, se consigue clasificar los tipos de datos (*abstracciones*) por ***variedad***, acercando así un paso más la programación al modo de razonar humano.

4.2 Definición de herencia

La herencia, entendida como una característica de la programación orientada a objetos y más concretamente del C++, permite ***definir una clase modificando una o más clases*** ya existentes. Estas modificaciones consisten habitualmente en ***añadir nuevos miembros*** (variables o funciones), a la clase que se está definiendo, aunque también se puede ***redefinir*** variables o funciones miembro ya existentes.

La clase de la que se parte en este proceso recibe el nombre de ***clase base***, y la nueva clase que se obtiene se denomina ***clase derivada***. Ésta a su vez puede ser ***clase base*** en un nuevo proceso de derivación, iniciando de esta manera una ***jerarquía de clases***. De ordinario las ***clases base*** suelen ser ***más generales*** que las ***clases derivadas***. Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian concretan y particularizan.

En algunos casos una clase no tiene otra utilidad que la de ser ***clase base*** para otras clases que se deriven de ella. A este tipo de ***clases base***, de las que no se declara ningún objeto, se les denomina ***clases base abstractas*** (***ABC, Abstract Base Class***) y su función es la de agrupar miembros comunes de otras clases que se derivarán de ellas. Por ejemplo, se puede definir la clase ***vehiculo*** para después derivar de ella ***coche***, ***bicicleta***, ***patinete***, etc., pero todos los objetos que se declaren pertenecerán a alguna de estas últimas clases; no habrá vehículos que sean sólo ***vehículos***. Las características comunes de estas clases (como una variable que indique si está parado o en marcha, otra que indique su velocidad, la función de arrancar y la de frenar, etc.), pertenecerán a la ***clase base*** y las que sean particulares de alguna de ellas pertenecerán sólo a la ***clase derivada*** (por ejemplo el número de platos y piñones, que sólo tiene sentido para una ***bicicleta***, o la función ***embragar*** que sólo se aplicará a los vehículos de motor con varias marchas).

Este mecanismo de ***herencia*** presenta múltiples ventajas evidentes a primera vista, como la ***posibilidad de reutilizar código*** sin tener que escribirlo de nuevo. Esto es posible porque todas las ***clases derivadas*** pueden utilizar el código de la ***clase base*** sin tener que volver a definirlo en cada una de ellas.

4.2.1 VARIABLES Y FUNCIONES MIEMBRO PROTECTED

Uno de los problemas que aparece con la ***herencia*** es el del control del ***acceso a los datos***. ¿Puede una función de una ***clase derivada*** acceder a los datos ***privados*** de su ***clase base***? En principio una clase no puede acceder a los datos privados de otra, pero podría ser muy conveniente que una ***clase derivada*** accediera a todos los datos de su ***clase base***. Para hacer posible esto, existe el tipo de dato

protected. Este tipo de datos es ***privado*** para todas aquellas clases que no son derivadas, pero ***público*** para una ***clase derivada*** de la clase en la que se ha definido la variable como ***protected***.

Por otra parte, el proceso de herencia puede efectuarse de dos formas distintas: siendo la clase base ***public*** o ***private*** para la clase derivada. En el caso de que la clase base sea ***public*** para la clase derivada, ésta hereda los miembros ***public*** y ***protected*** de la clase base como miembros ***public*** y ***protected,*** respectivamente. Por el contrario, si la clase base es ***private*** para la clase derivada, ésta hereda todos los datos de la clase base como ***private***. La siguiente tabla puede resumir lo explicado en los dos últimos párrafos.

Tipo de dato de la clase base	Clase derivada de una clase base **public**	Clase derivada de una clase base **private**	Otras clases sin relación de herencia con la clase base
Private	*No accesible directamente*	*No accesible directamente*	*No accesible directamente*
Protected	*Protected*	*Private*	*No accesible directamente*
Public	*Public*	*Private*	*Accesible mediante operador (.) o (->)*

Tabla 1: Herencia pública y privada.

Como ejemplo, se puede pensar en dos tipos de cuentas bancarias que comparten algunas características y que también tienen algunas diferencias. Ambas cuentas tienen un ***saldo***, un ***interés*** y el ***nombre*** del titular de la cuenta. La ***cuenta joven*** es un tipo de cuenta que requiere la ***edad del propietario***, mientras que la ***cuenta empresarial*** necesita el ***nombre de la empresa***. El problema podría resolverse estableciendo una clase base llamada ***C_Cuenta*** y creando dos tipos de cuenta derivados de dicha clase base.

Para indicar que una ***clase deriva de otra*** es necesario indicarlo en la ***definición de la clase derivada***, especificando el modo -***public*** o ***private***- en que deriva de su ***clase base***:

```
class Clase_Derivada : public o private Clase_Base
```

De esta forma el código necesario para crear esas tres clases mencionadas quedaría de la siguiente forma:

```
#include <iostream.h>
class C_Cuenta {
    // Variables miembro
    private:
        char   *Nombre;    // Nombre de la persona
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado
    public:
        // Constructor
        C_Cuenta(const char *unNombre, double unSaldo=0.0, double unInteres=0.0)
        {
            Nombre = new char[strlen(unNombre)+1];
            strcpy(Nombre, unNombre);
            SetSaldo(unSaldo);
            SetInteres(unInteres);
        }
        // Destructor
        ~Cuenta()
            { delete [] Nombre; }
        // Métodos
        inline char *GetNombre()
            { return Nombre; }
        inline double GetSaldo()
            { return Saldo; }
```

```
        inline double GetInteres()
            { return Interes; }
        inline void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        inline void SetInteres(double unInteres)
            { Interes = unInteres; }
        inline void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
        friend ostream& operator<<(ostream& os, C_Cuenta& unaCuenta)
        {
            os << "Nombre=" << unaCuenta.GetNombre() << endl;
            os << "Saldo=" << unaCuenta.GetSaldo() << endl;
            return os;
        }
};

class C_CuentaJoven : public C_Cuenta {
    private:
        int Edad;
    public:
        C_CuentaJoven(     // argumentos del constructor
            const char *unNombre,
            int        laEdad,
            double     unSaldo=0.0,
            double     unInteres=0.0)
                 : C_Cuenta(unNombre, unSaldo, unInteres)
            // se llama al constructor de la clase base en la línea previa.
        {
            Edad = laEdad;
        }
};
class C_CuentaEmpresarial : public C_Cuenta {
    private:
        char *NomEmpresa;
    public:
        C_CuentaEmpresarial(    // argumentos del constructor
            const char *unNombre,
            const char *laEmpresa,
            double     unSaldo=0.0,
            double     unInteres=0.0)
                 : C_Cuenta(unNombre, unSaldo, unInteres)
        // se llama al constructor de la clase base en la línea previa.
        {
            NomEmpresa = new char[strlen(laEmpresa)+1];
            strcpy(NomEmpresa, laEmpresa);
        }
        // Cuando una variable de este tipo se destruye se llamará
        //   primero el destructor de CuentaEmpresarial y posteriormente se
        //   llama automáticamente el destructor de la clase base.
        ~C_CuentaEmpresarial()
             { delete [] NomEmpresa; }
};

void main()
{
    C_CuentaJoven c1("Igor", 18, 10000.0, 1.0);
    C_CuentaEmpresarial c2("Juan", "MicroComputers Corp." ,10000000.0);
    // Ambas cuentas pueden llamar métodos definidos previamente
    cout << c1;
    cout << c2;
}
```

Si un miembro heredado se ***redefine*** en la clase derivada, el nombre redefinido oculta el nombre heredado que ya queda invisible para los objetos de la clase derivada.

Hay algunos elementos de la clase base que ***no pueden ser heredados***:

- Constructores
- Destructores
- Funciones ***friend***
- Funciones y datos estáticos de la clase
- Operador de asignación (=) sobrecargado

4.3 Constructores de las clases derivadas: inicializador base

Un objeto de la ***clase derivada*** contiene todos los miembros de la ***clase base*** y todos esos miembros deben ser inicializados. Por esta razón el ***constructor de la clase derivada*** debe llamar al ***constructor de la clase base***. Al definir el ***constructor de la clase derivada*** se debe especificar un ***inicializador base***.

Como ya se ha dicho las clases derivadas ***no heredan los constructores*** de sus clases base. El ***inicializador base*** es la forma de llamar a los ***constructores de las clases base*** y poder así inicializar las variables miembro heredadas. Este ***inicilizador base*** se especifica poniendo, a continuación de los argumentos del constructor de la clase derivada, el carácter dos puntos (:) y el nombre del constructor de la clase o las clases base, seguido de una lista de argumentos entre paréntesis.

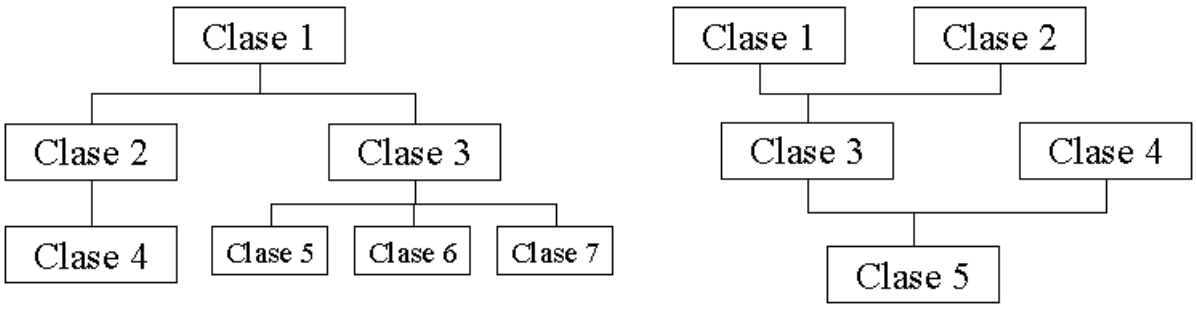
El ***inicializador base*** puede ser omitido en el caso de que la clase base tenga un ***constructor por defecto***. En el caso de que el constructor de la clase base exista, al declarar un objeto de la clase derivada se ejecuta primero el constructor de la clase base.

En el ejemplo del apartado anterior ya se puede ver como se llama al ***constructor de la clase base*** desde el ***constructor de la clase derivada***:

```
C_CuentaJoven(const char *unNombre, int laEdad, double unSaldo=0.0,
    double unInteres=0.0) : C_Cuenta(unNombre, unSaldo, unInteres)
```

4.4 Herencia simple y herencia múltiple

Una clase puede heredar variables y funciones miembro de una o más clases base. En el caso de que herede los miembros de una única clase se habla de ***herencia simple*** y en el caso de que herede miembros de varias clases base se trata de un caso de ***herencia múltiple***. Esto se ilustra en la siguiente figura:



Herencia Simple: Todas las clases derivadas tienen una única clase base

Herencia Múltiple: Las clases derivadas tienen varias clases base

Figura 4: Herencia simple y herencia múltiple.

Como ejemplo se puede presentar el caso de que se tenga una clase para el manejo de los datos de la empresa. Se podría definir la clase ***C_CuentaEmpresarial*** como la herencia múltiple de

dos clases base: la ya bien conocida clase ***C_Cuenta*** y nueva clase llamada ***C_Empresa***, que se muestra a continuación:

```
class C_Empresa {
    private:
        char *NomEmpresa;
    public:
        C_Empresa(const char*laEmpresa)
        {
            NomEmpresa = new char[strlen(laEmpresa)+1];
            strcpy(NomEmpresa, laEmpresa);
        }
        ~C_Empresa()
            { delete [] NomEmpresa; }
        // Otros métodos ...
};

class C_CuentaEmpresarial : public C_Cuenta, public C_Empresa {
    public:
        C_CuentaEmpresarial(
            const char *unNombre,
            const char *laEmpresa,
            double     unSaldo=0.0,
            double     unInteres=0.0
        ) : C_Cuenta(unNombre, unSaldo, unInteres), C_Empresa(laEmpresa)
        // se llama a los constructores de las clases base en la línea previa
        {
            // Constructor
        }
        // Otros métodos
};
```

4.5 Clases base virtuales

Al utilizar la herencia múltiple puede suceder que, indirectamente, una clase ***herede varias veces*** los miembros de otra clase, tal como se ve en la figura 5.

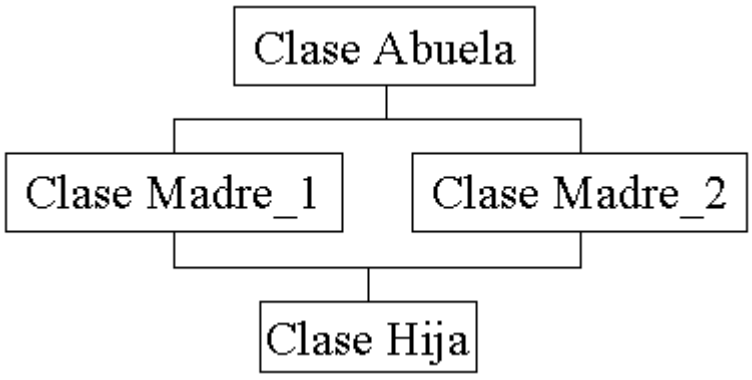


Figura 5: Necesidad de las clases base virtuales.

Si la clase ***Madre_1*** y la clase ***Madre_2*** heredan los miembros de la clase ***Abuela*** y la clase ***Hija*** hereda, a su vez, los miembros de las clases ***Madre_1*** y ***Madre_2***, los miembros de la clase ***Abuela*** se encontrarán duplicados en la clase ***Hija***. Para evitar este problema las clases ***Madre_1*** y ***Madre_2*** deben derivar de la clase ***Abuela*** declarándola ***clase base virtual***. Esto hace que los miembros de una clase de ese tipo se hereden tan sólo una vez. Un ejemplo de declaración de una ***clase base virtual*** es el que se presenta a continuación:

```
class Madre_1 : virtual public Abuela {
...
}
```

4.6 Conversiones entre objetos de clases base y clases derivadas

Es posible realizar ***conversiones*** o ***asignaciones*** de un ***objeto de una clase derivada*** a un ***objeto de la clase base***. Es decir se puede ir de lo más particular a lo más general, aunque en esa operación se pierda información, pues haya variables que no tengan a qué asignarse (el número de variables miembro de una clase derivada es mayor o igual que el de la clase de la que deriva).

Por el contrario las ***conversiones*** o ***asignaciones en el otro sentido***, es decir de lo más general a lo más particular, ***no son posibles***, porque puede suceder que no se disponga de valores para todas las variables miembro de la clase derivada.

Así pues, la siguiente asignación sería correcta:

```
Objeto_clase_base = Objeto_clase_derivada          // Asignación válida
```

mientras que esta otra sería incorrecta:

```
Objeto_clase_derivada = Objeto_clase_base          // Asignación incorrecta
```

En el siguiente ejemplo se pueden ver las distintas posibilidades de asignación (más bien de ***inicialización***, en este caso), que se presentan en la clase ***C_CuentaEmpresarial.***

```
void main()
{
    // Válido
    C_CuentaEmpresarial *c1 = new C_CuentaEmpresarial("Juan",
          "Jugos SA", 100000.0, 10.0);

    // Válido. Se utilizan los valores por defecto
    C_Cuenta *c2 = new C_CuentaEmpresarial("Igor", "Patata CORP");

    // NO VÁLIDO
    C_CuentaEmpresarial *c3 = new C_Cuenta("Igor", 100.0, 1.0);
    // ...
}
```

De forma análoga, se puede guardar la dirección almacenada en un ***puntero a una clase derivada*** en un ***puntero a la clase base***. Esto quiere decir que se puede hacer referencia a un ***objeto de la clase derivada*** con su dirección contenida en un ***puntero a la clase base***.

Al igual que sucede con los nombres de los objetos, en principio cuando se hace referencia a un objeto por medio de un puntero, ***el tipo de dicho puntero determina la función miembro que se aplica***, en el caso de que esa función se encuentre definida tanto en la clase base como en la derivada. En definitiva, ***un puntero a la clase base puede almacenar la dirección de un objeto perteneciente a una clase derivada***. Sin embargo, se aplicarán los métodos de la clase a la que pertenezca el puntero, no los de la clase a la que pertenece el objeto.

5 POLIMORFISMO

Polimorfismo, por definición, es la ***capacidad de adoptar formas distintas***. En el ámbito de la *Programación Orientada a Objetos* se entiende por ***polimorfismo*** la capacidad de ***llamar a funciones distintas con un mismo nombre***. Estas funciones pueden actuar sobre objetos distintos dentro de una jerarquía de clases, sin tener que especificar el tipo exacto de los objetos. Esto se puede entender mejor con el ejemplo de la figura 6:

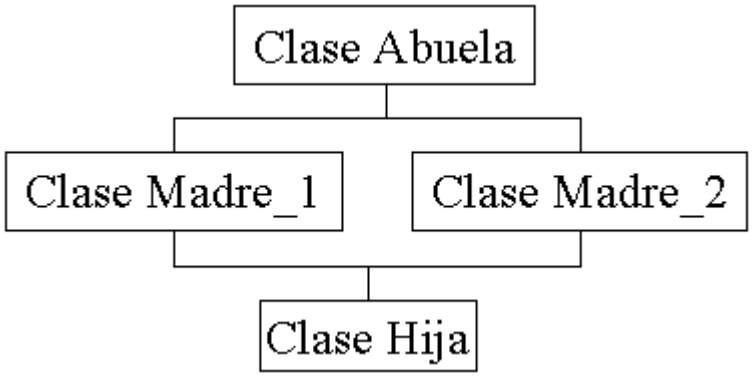


Figura 6: Funciones virtuales

En el ejemplo que se ve en la figura 6 se observa una jerarquía de clases. En todos los niveles de esta jerarquía está contenida una función llamada ***Funcion_1()***. Esta función no tiene por qué ser igual en todas las clases, aunque es habitual que sea una función que efectúe una operación muy parecida sobre distintos tipos de objetos.

Es importante comprender que ***el compilador no decide en tiempo de compilación*** cuál será la función que se debe utilizar en un momento dado del programa. Esa decisión se toma ***en tiempo de ejecución***. A este proceso de decisión en tiempo de ejecución se le denomina ***vinculación dinámica o tardía,*** en oposición a la habitual ***vinculación estática o temprana***, consistente en decidir en tiempo de compilación qué función se aplica en cada caso.

A este tipo de funciones, incluidas en varios niveles de una jerarquía de clases con ***el mismo nombre pero con distinta definición***, se les denomina ***funciones virtuales***. Hay que insistir en que la definición de la función en cada nivel es distinta.

El ***polimorfismo*** hace posible que un usuario pueda ***añadir nuevas clases*** a una jerarquía sin ***modificar o recompilar*** el código original. Esto quiere decir que si desea añadir una nueva clase derivada es suficiente con establecer la clase de la que deriva, definir sus nuevas variables y funciones miembro, y compilar esta parte del código, ensamblándolo después con lo que ya estaba compilado previamente.

Es necesario comentar que las ***funciones virtuales*** son *algo menos eficientes* que las funciones normales. A continuación se explica, sin entrar en gran detalle, el funcionamiento de las ***funciones virtuales***. Cada clase que utiliza ***funciones virtuales*** tiene un ***vector de punteros***, uno por cada ***función virtual***, llamado ***v-table***. Cada uno de los punteros contenidos en ese vector apunta a la ***función virtual*** apropiada para esa clase, que será, habitualmente, la ***función virtual*** definida en la propia clase. En el caso de que en esa clase no esté definida la ***función virtual*** en cuestión, el puntero de ***v-table*** apuntará a la ***función virtual*** de su clase base más próxima en la jerarquía, que tenga una definición propia de la ***función virtual***. Esto quiere decir que buscará primero en la propia clase, luego en la clase anterior en el orden jerárquico y se irá subiendo en ese orden hasta dar con una clase que tenga definida la función buscada.

Cada objeto creado de una clase que tenga una ***función virtual*** contiene un puntero oculto a la ***v-table*** de su clase. Mediante ese puntero accede a su ***v-table*** correspondiente y a través de esta tabla accede a la definición adecuada de la ***función virtual***. Es este trabajo extra el que hace que las funciones virtuales sean menos eficientes que las funciones normales.

Como ejemplo se puede suponer que la ***cuenta_joven*** y la ***cuenta_empresarial*** antes descritas tienen una forma distinta de abonar mensualmente el interés al saldo.

- En la ***cuenta_joven***, no se abonará el interés pactado si el saldo es inferior a un límite.
- En la ***cuenta_empresarial*** se tienen tres cantidades límite, a las cuales se aplican factores de corrección en el cálculo del interés. El cálculo de la cantidad abonada debe realizarse de la siguiente forma:
    1. Si el saldo es menor que 50000, se aplica el interés establecido previamente.
    2. Si el saldo está entre 50000 y 500.000, se aplica 1.1 veces el interés establecido previamente.
    3. Si el saldo es mayor a 500.000, se aplica 1.5 veces el interés establecido previamente.

El código correspondiente quedaría de la siguiente forma:

```
class C_Cuenta {
    // Variables miembro
    private:
        double Saldo;     // Saldo Actual de la cuenta
        double Interes;   // Interés calculado hasta el momento, anual,
                          //   en tanto por ciento %
    public:
        //Constructor
        C_Cuenta(double unSaldo=0.0, double unInteres=4.0)
        {
            SetSaldo(unSaldo);
            SetInteres(unInteres);
        }
        // Acciones Básicas
        inline double GetSaldo()
            { return Saldo; }
        inline double GetInteres()
            { return Interes; }
        inline void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        inline void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
        virtual void AbonaInteresMensual()
        {
            SetSaldo( GetSaldo() * ( 1.0 + GetInteres() / 12.0 / 100.0) );
        }
        // etc...
};

class C_CuentaJoven : public C_Cuenta {
    public:
        C_CuentaJoven(double unSaldo=0.0, double unInteres=2.0,
            double unLimite = 50.0E3) :
                 C_Cuenta(unSaldo, unInteres)
        {
            Limite = unLimite;
        }
```

```
        virtual void AbonaInteresMensual()
        {
            if (GetSaldo() > Limite)
                SetSaldo( GetSaldo() * (1.0 + GetInteres() / 12.0 / 100.0) );
            else
                SetSaldo( GetSaldo() );
        }
    private:
        double Limite;
};

class C_CuentaEmpresarial : public C_Cuenta {
    public:
        C_CuentaEmpresarial(double unSaldo=0.0, double unInteres=4.0)
            : C_Cuenta(unSaldo, unInteres)
        {
            CantMin[0] = 50.0e3;
            CantMin[1] = 500.0e3;
        }
        virtual void AbonaInteresMensual()
        {
            SetSaldo( GetSaldo() * (1.0 + GetInteres() * CalculaFactor() / 12.0 / 100.0 ));
        }
        double CalculaFactor()
        {
            if (GetSaldo() < CantMin[0])
                return 1.0;
            else if (GetSaldo() < CantMin[1])
                return 1.1;
            else return 1.5;
        }
    private:
        double CantMin[2];
};
```

La idea central del ***polimorfismo*** es la de ***poder llamar a funciones distintas aunque tengan el mismo nombre, según la clase a la que pertenece el objeto al que se aplican***. Esto es imposible utilizando ***nombres de objetos***: siempre se aplica la función miembro de la clase correspondiente al nombre del objeto, y esto se decide en tiempo de compilación.

Sin embargo, ***utilizando punteros puede conseguirse el objetivo buscado***. Recuérdese que un ***puntero a la clase base*** puede contener ***direcciones de objetos*** de cualquiera de las ***clases derivadas***. En principio, el tipo de puntero determina también la función que es llamada, pero ***si se utilizan funciones virtuales es el tipo de objeto el que apunta el puntero lo que determina la función que se llama***. Esta es la esencia del ***polimorfismo***.

5.1 Implementacion de las funciones virtuales

Una ***función virtual*** puede ser llamada como una función convencional, es decir, utilizando ***vinculación estática***. En este caso no se están aprovechando las características de las ***funciones virtuales***, pero el programa puede funcionar correctamente. A continuación se presenta un ejemplo de este tipo de implementación que no es recomendable usar, ya que utilizando una función convencional se ganaría en eficiencia:

```
Clase_1 Objeto_1;              // Se definen un objeto de una clase
Clase_1 *Puntero_1;            //   y un puntero que puede apuntarlo
float   variable;

Puntero_1 = &Objeto_1;
variable = Objeto_1.Funcion_1( );     // Utilización de vinculación estática
variable = Puntero_1->Funcion_1( );   //   con funciones virtuales. Absurdo
```

En el ejemplo anterior en las dos asignaciones a *variable*, las funciones que se van a utilizar se determinan en ***tiempo de compilación***.

A continuación se presenta un ejemplo de utilización correcta de las ***funciones virtuales***:

```
Clase_Base      Objeto_Base;
Clase_Derivada  Objeto_Derivado;
Clase_Base      *Puntero_Base_1;
Clase_Base      *Puntero_Base_2;
float           variable;

…
Puntero_Base_1 = &Objeto_Base;      // El puntero a la clase base puede
                                    //   apuntar a un objeto de la clase base
Puntero_Base_2 = &Objeto_Derivado;  //   o a un objeto de la clase derivada
…
variable = Puntero_Base_2->Funcion_1( );    // Utilización correcta
                                            //   de una función virtual
```

En este nuevo ejemplo se utiliza ***vinculación dinámica***, ya que el ***Puntero_Base_2*** puede apuntar a un ***objeto de la clase base*** o a un ***objeto de cualquiera de las clases derivadas*** en el momento de la asignación a ***variable***, en la última línea del ejemplo. Por eso, es necesariamente en tiempo de ejecución cuando el programa decide cuál es la ***Funcion_1*** concreta que tiene que utilizar.

Esa ***Funcion_1*** será la definida para la clase del ***Objeto_Derivado*** si está definida, o la de la ***clase base más próxima*** en el orden jerárquico que tenga definida esa ***Funcion_1***.

5.2 Funciones virtuales puras

Habitualmente las ***funciones virtuales*** de la clase base de la jerarquía no se utilizan porque en la mayoría de los casos ***no se declaran objetos de esa clase***, y/o porque todas las clases derivadas tienen su propia definición de la ***función virtual***. Sin embargo, incluso en el caso de que la ***función virtual*** de la clase base no vaya a ser utilizada, debe declararse.

De todos modos, ***si la función no va a ser utilizada no es necesario definirla***, y es suficiente con ***declararla como función virtual pura***. Una función virtual pura se declara así:

```
virtual funcion_1( ) const=0;    //Función virtual pura
```

La única utilidad de esta declaración es la de posibilitar la ***definición de funciones virtuales en las clases derivadas***. De alguna manera se puede decir que la definición de una función como virtual pura hace necesaria la definición de esa función en las clases derivadas, a la vez que imposibilita su utilización con objetos de la clase base.

Al definir una función como ***virtual pura*** hay que tener en cuenta que:

- No hace falta definir el código de esa función en la clase base.
- No se pueden definir objetos de la clase base, ya que no se puede llamar a las ***funciones virtuales puras***.
- Sin embargo, ***es posible definir punteros a la clase base, pues es a través de ellos como será posible manejar objetos de las clases derivadas***.

5.3 Clases abstractas

Se denomina ***clase abstracta*** a aquella que ***contiene una o más funciones virtuales puras***. El nombre proviene de que ***no puede existir ningún objeto de esa clase***. Si una clase derivada no redefine una ***función virtual pura***, la clase derivada la hereda como ***función virtual pura*** y se convierte también en ***clase abstracta***. Por el contrario, aquellas clases derivadas que redefinen todas las ***funciones virtuales puras*** de sus clases base reciben el nombre de ***clases derivadas concretas***, nomenclatura únicamente utilizada para diferenciarlas de las antes mencionadas.

Aparentemente puede parecer que carece de sentido definir una clase de la que no va a existir ningún objeto, pero se puede afirmar, sin miedo a equivocarse, que la ***abstracción*** es una herramienta imprescindible para un correcto diseño de la ***Programación Orientada a Objetos***.

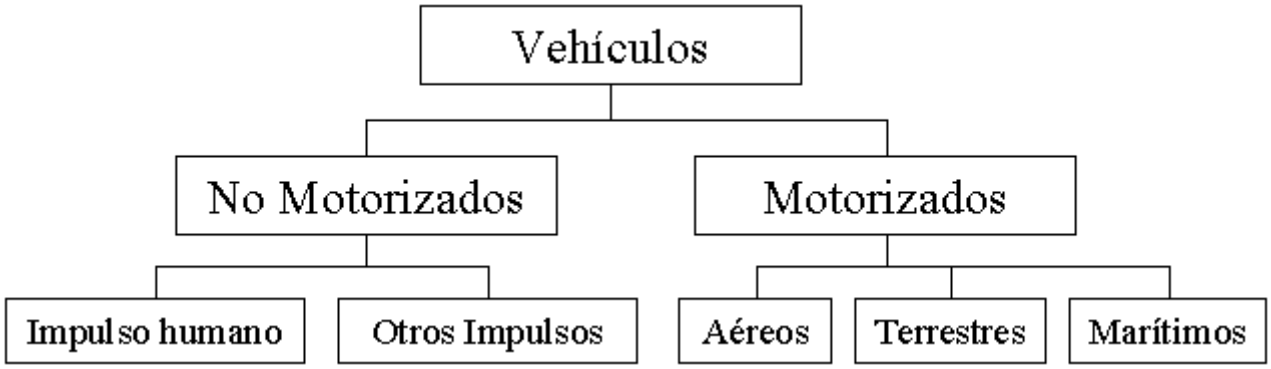


Figura 7. Clases base virtuales.

Está claro que la jerarquía que se presenta en la figura 7 no es suficiente, porque un avión y un helicóptero, o un patinete y una bicicleta, serían objetos de la misma clase. Pero lo que se pretende ilustrar es la necesidad de una clase ***vehículo*** que englobe las características comunes de todos ellos (peso, velocidad máxima, …), aunque no exista ningún objeto de esa clase, ya que cualquier vehículo en el que se piense, podrá definirse como un objeto de una clase derivada de la primera clase base.

Habitualmente las clases superiores de muchas jerarquías de clases son ***clases abstractas*** y las clases que heredan de ellas definen sus propias ***funciones virtuales***, convirtiéndose así en ***funciones concretas***.

5.4 Destructores virtuales

Como norma general, ***el constructor de la clase base se llama antes que el constructor de la clase derivada***. Con los ***destructores***, sin embargo, ***sucede al revés***: el destructor de la clase derivada se llama antes que el de la clase base.

Por esa razón, en el caso de que se borre, aplicando ***delete***, un ***puntero a un objeto de la clase base*** que apunte a un ***objeto de una clase derivada***, se llamará al ***destructor de la clase base***, en vez de al ***destructor de la clase derivada***, que sería lo adecuado. La solución a este problema consiste en ***declarar como virtual el destructor de la clase base***. Esto hace que automáticamente los ***destructores de las clases derivadas sean también virtuales***, a pesar de tener nombres distintos.

De este modo, al aplicar ***delete*** a un ***puntero de la clase base*** que puede apuntar a un objeto de ese tipo o a cualquier objeto de una clase derivada, ***se aplica el destructor adecuado*** en cada caso. Por esta razón es conveniente declarar ***un destructor virtual en todas las clases abstractas***, ya que aunque no sea necesario para esa clase, sí puede serlo para una clase que derive de ella.

Este problema no se presenta con los ***constructores*** y por eso no existe ningún tipo de constructor virtual o similar.

6 ENTRADA/SALIDA EN C++

Ya se ha visto que C++ dispone de unas herramientas propias de entrada y salida de datos basadas en ***clases*** y en la ***herencia*** que son fáciles de extender y modificar. Si este tema no se ha visto anteriormente con más extensión, es porque conviene haber visto la ***herencia*** para entenderlo correctamente.

Es necesario recordar aquí el concepto de ***stream*** o ***flujo,*** que se puede definir como dispositivo que produce o consume información. Un ***flujo*** está siempre ligado a un dispositivo físico. Todos los ***flujos***, independientemente del dispositivo físico al que se dirijan (disco, monitor…,) se comportan de forma análoga.

Al ejecutarse un programa en C++ se abren automáticamente los ***flujos*** siguientes:

cin: entrada estándar (teclado)

cout: salida estándar (pantalla)

cerr: salida de mensajes de error (pantalla)

C++ dispone de dos jerarquías de clases para las operaciones de entrada/salida: una de bajo nivel, ***streambuf,*** que no se va a explicar porque sólo es utilizada por programadores expertos, y otra de alto nivel, con las clases: ***istream***, ***ostream*** e ***iostream***, que derivan de la clase ***ios.*** Estas clases disponen de ***variables*** y ***métodos*** para controlar los ***flujos*** de entrada y salida (ver la jerarquía de clases de entrada/salida en la figura 8, un poco más adelante).

6.1 Entrada/salida con formato

Cada ***flujo*** de C++ tiene asociados unos ***indicadores***, que son unas variables miembro ***enum*** de tipo ***long*** que controlan el ***formato*** al activarse o desactivarse alguno de sus ***bits***. Su valor hexadecimal es:

```
enum {
    skipws=0x0001,      left=0x0002,       rigth=0x0004,    internal=0x0008,
    dec=0x0010,         oct=0x0020         hex=0x0040,      showbase=0x0080,
    showpoint= 0x0100   uppercase=0x0200,  showpos=0x0400,  scientific=0x800,
    fixed=0x1000,       unitbuf=0x2000
};
```

Su significado es el siguiente:

skipws:	se descartan los blancos iniciales a la entrada
left:	la salida se alinea a la izquierda
rigth:	la salida se alinea a la derecha
internal:	se alinea el signo y los caracteres indicativos de la base por la izquierda y las cifras por la derecha
dec:	salida decimal para enteros (defecto)
oct:	salida octal para enteros
hex:	salida hexadecimal al para enteros
showbase:	se muestra la base de los valores numéricos
showpoint:	se muestra el punto decimal
uppercase:	los caracteres de formato aparecen en mayúsculas
showpos:	se muestra el signo (+) en los valores positivos
scientific:	notación científica para coma flotante
fixed:	notación normal para coma flotante

unitbuf:	salida sin buffer (se vuelca cada operación)
stdio	permite compatibilizar entrada/salida al modo de C con **<*stdio.h*>** y al modo de C++ con **<*iostream.h*>**

La forma de definir las constantes anteriores permite componerlas fácilmente, guardando toda la información sobre ellas en una única variable ***long***. Existen unos ***indicadores*** adicionales (***adjustfield***, ***basefield*** y ***floatfield***) que actúan como combinaciones de los anteriores (máscaras):

adjustfield	es una combinación excluyente (sólo una en *on*) de ***left***, ***rigth*** e ***internal***
basefield	es una combinación excluyente (sólo una en *on*) de ***dec***, ***oct*** e ***hex***
floatfield	es una combinación excluyente (sólo una en *on*) de ***scientific*** y ***fixed***

Por defecto todos los indicadores anteriores están desactivados, excepto ***skipws*** y ***dec***.

6.2 Activar y desactivar indicadores

Para la activación de ***indicadores*** se pueden utilizar los métodos ***setf()*** y ***flags()*** de la clase ***ios.*** Se comenzará viendo la primera de ellas.

Los dos prototipos del método ***setf()*** son:

```
long setf(long indic);
long setf(long indic, long mask);
```

El valor de retorno de esta función es la *configuración anterior* (interesa disponer de ella al hacer algún cambio para poder volver a dicha configuración si se desea), e ***indic*** es el ***long*** que contiene los ***indicadores***. En el segundo prototipo ***mask*** es uno de los tres indicadores combinación de otros (***adjustfield***, ***basefield*** y ***floatfield***).

Se permite activar varios ***indicadores*** a la vez con el operador lógico OR binario. Ejemplo:

```
cout.setf(ios::showpoint | ios::fixed);
```

Es necesario determinar el ***flujo*** afectado (***cout***) y la ***clase*** en la que están definidos los indicadores (***ios***). Para desactivar los ***indicadores*** se utiliza la función ***unsetf()*** de modo similar a ***setf()***.

El segundo prototipo se debe utilizar para cambiar los indicadores que son exclusivos, como por ejemplo:

```
cout.setf(ios::left, ios::adjustfield);
```

que lo que hace es desactivar los tres bits que tienen que ver con la alineación y después activar el bit de alineación por la izquierda. En la forma,

```
cout.setf(0, ios::adjustfield);
```

pone los tres bits de alineación a cero.

La función ***flags()*** sin argumentos devuelve un ***long*** con la configuración de todos los ***indicadores***. Su prototipo es:

```
long flags();
```

Existe otra definición de la función ***flags()*** cuyo valor de retorno es un ***long*** con la configuración anterior, que permite cambiar todos los ***indicadores*** a la vez, pasando un ***long*** con la nueva configuración como argumento:

```
long flags(long indic);
```

donde ***indic*** contiene una descripción completa de la nueva configuración. El inconveniente de la función ***flags()*** es que establece una nueva configuración partiendo de cero, mientras que ***setf()*** simplemente modifica la configuración anterior manteniendo todo lo que no se ha cambiado explícitamente, por lo que debe ser considerada como una opción más segura.

Se presenta a continuación un ejemplo con todo lo citado hasta ahora:

```
// se mostrará el signo + para números positivos
cout.setf(ios::showpos);
// se mostrará el punto y no se utilizará notación científica
cout.setf(ios::showpoint | ios::fixed);
cout << 100.0;
```

que hace que se escriba por pantalla:

```
+100.000000
```

6.3 Funciones miembro *width()*, *precision()* y *fill()*

Estas funciones están declaradas en ***ios*** y definidas en las clases ***istream***, ***ostream*** e ***iostream***. La función miembro ***width()*** establece la anchura de campo mínima para un dato de salida. Sus prototipos son:

```
int width(int n);
int width();
```

donde el valor de retorno es la anchura anterior.

La anchura establecida con la función ***width()*** es la ***mínima*** y siempre que sea necesario el sistema la aumenta de modo automático. Esta anchura de campo sólo es válida para el siguiente dato que se imprime. Si se desea que siga siendo válida hay que llamarla cada vez.

La función miembro ***precision()*** establece el número de cifras para un dato de salida. Si no se indica nada la ***precisión por defecto*** es 6 dígitos. Los prototipos de la función ***precision()*** son:

```
int precision(int n);
int precision();
```

donde el valor de retorno es la precisión anterior.

La función miembro ***fill()*** establece el ***carácter de relleno*** para un dato de salida. Por defecto el carácter de relleno es el blanco ‘ ‘. Los prototipos de esta función son:

```
char fill(char ch);
char fill();
```

donde el valor de retorno es el carácter de relleno anterior.

En el compilador ***Visual C++*** de Microsoft sólo ***width()*** necesita ser llamada cada vez.

6.3.1 MANIPULADORES DE ENTRADA/SALIDA

Los ***manipuladores*** son constantes y/o métodos que constituyen una alternativa a los ***indicadores***. Se pueden introducir en la propia sentencia de entrada o salida. Los ***manipuladores*** pueden tener argumentos o no tenerlos. Los manipuladores sin argumentos (***endl***, ***flush***, etc.) están definidos en ***iostream.h***. Los que tienen argumentos están declarados en ***iomanip.h***. Un ***manipulador*** sólo afecta al ***flujo*** (***cin***, ***cout***, etc.) al que se aplica.

El inconveniente de los ***manipuladores*** frente a los ***indicadores*** es que no permiten guardar la configuración anterior y por tanto volver a ella de una forma general y sencilla.

Los ***manipuladores*** de entrada/salida más utilizados se citan a continuación:

dec, ***hex*** y ***oct***:	establecen base para enteros
ws:	se saltan los blancos iniciales
endl:	se imprime un **'\n'** y se vacía el buffer de salida
flush:	se vacía el buffer de salida
setw(int w):	establece la anchura mínima de campo
setprecision(int p):	establece el número de cifras
setfill(char ch):	establece el carácter de relleno
setiosflag(long i)	equivale al *indicador* ***setf()***
unsetiosflag(long i)	equivale a ***unsetf()***

Un ***manipulador*** se utiliza de la forma:

```
cout << hex << 100;
cout << setw(10) << mat[i][j] << endl;
```

El efecto de los ***manipuladores*** permanece en el ***flujo*** correspondiente hasta que se cambian por otro ***manipulador***, a excepción de ***setw()*** que hay que introducirlo en el flujo antes de cada dato al que se le quiera aplicar esa anchura de campo.

6.4 Sobrecarga de los operadores de entrada/salida (<< y >>)

Los operadores de ***extracción*** (>>) e ***inserción*** (<<) en los flujos de entrada y salida se pueden sobrecargar como otros operadores de C++. Ambos son operadores binarios (2 argumentos). Sus prototipos son los siguientes:

```
ostream& operator<< (ostream& co, const obj_type& a);
istream& operator>> (istream& ci, obj_type& a);
```

Estos ***flujos*** funcionan como cintas transportadoras que *entran* (>>) o *salen* (<<) del programa. Se recibe una ***referencia al flujo*** como primer argumento, se añade o se retira de él ***la variable que se desee***, y se devuelve como valor de retorno una ***referencia al flujo*** modificado. El valor de retorno es siempre una ***referencia*** al ***stream*** de entrada/salida correspondiente. A continuación se presenta un ejemplo de sobrecarga del operador (<<):

```
ostream& operator<<(ostream& co, const matriz& A)
{
    for (int i=0; i<=nfilas; i++)
        for (int j=0; j<=ncol; j++)
            co << A.c[i][j] << '\t';
        co << '\n';
    return(co);
}
```

Estos operadores se sobrecargan siempre como operadores ***friend*** de la clase en cuestión, ya que el primer argumento no puede ser un objeto de una clase arbitraria.

6.5 Entrada/salida de ficheros

Para poder leer desde o escribir en ficheros (lectura/escritura de datos en unidades de almacenamiento permanente como los disquetes, discos duros, etc.), se debe incluir la librería <***fstream.h***>. En esta librería se definen las clases necesarias para la utilización de ficheros, que son ***ifstream***, ***ofstream*** y ***fstream***, que derivan de ***istream*** y ***ostream***, que a su vez derivan de la clase ***ios*** (ver figura 8).

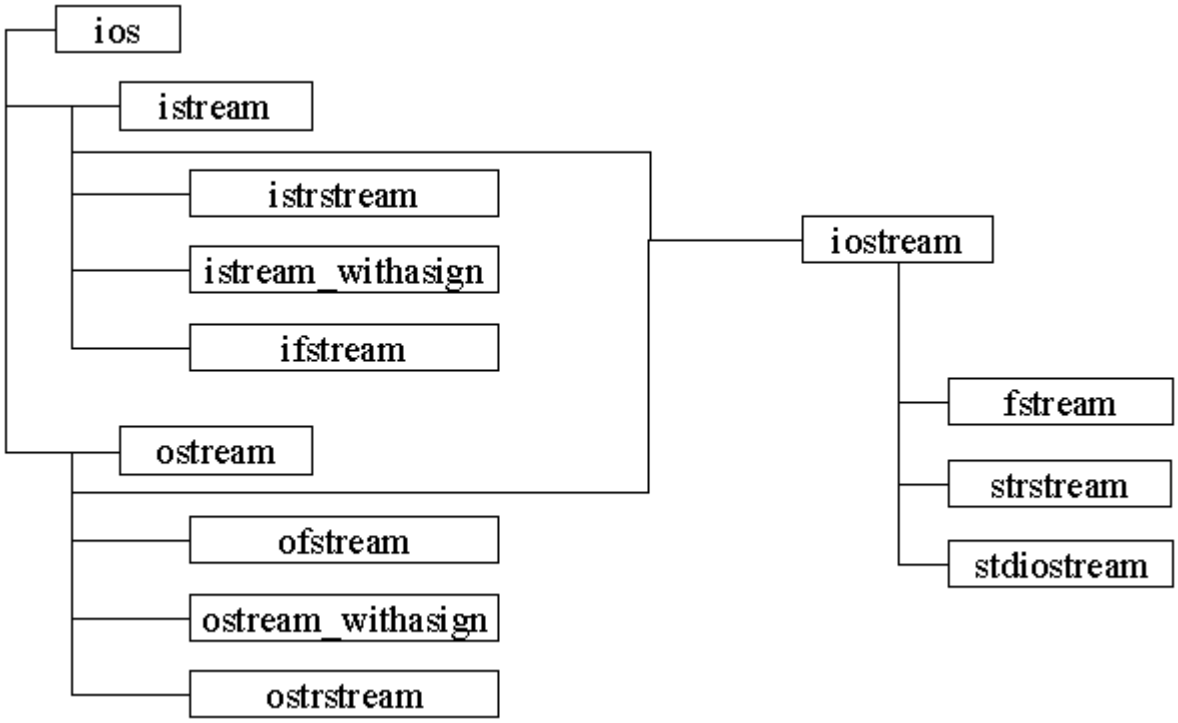


Figura 8. Jerarquía de clases de entrada/salida.

Antes de ***abrir un fichero*** hay que crear un ***flujo***, es decir un objeto de las clases ***ifstream***, ***ofstream*** o ***fstream*** e indicar el modo de apertura (lectura, escritura, lectura y escritura, …). Los modos en los que se puede abrir un fichero son:

ios::append	añadir datos al final del fichero
ios::in	abrir fichero para leer datos
ios::out	abrir fichero para escribir datos

Por ejemplo para abrir un fichero para lectura de datos creando un *fstream* ***fichero***:

```
fstream fichero;
fichero.open("datos.dat", ios::in);
```

y para escritura en ese mismo fichero:

```
fstream fichero;
fichero.open("datos.dat", ios::out);
```

Las clases ***ifstream***, ***ofstream*** y ***fstream*** tienen también constructores que permiten abrir ficheros de forma automática

```
ifstream fichero("datos.dat");
```

donde se sobreentiende que el fichero se abre para ***lectura*** por haber utilizado ***ifstream***. Si se hubiese utilizado ***ofstream*** el fichero se hubiera abierto para escritura.

6.5.1 FUNCIONES MIEMBRO DE IOSTREAM

Las clases de entrada y salida de datos de C++ disponen de algunas funciones miembro que aumentan las capacidades de estas entradas y salidas. A continuación se incluye la declaración y una breve explicación de las más importantes:

ostream& put(char c);	escribe un carácter en el flujo de salida
ostream& write(const char* s, int n);	escribe ***n*** bytes de la cadena ***s*** en el flujo de salida. Puede utilizarse para salida binaria.

istream& get(char& c);	lee un carácter del flujo de entrada y lo devuelve en el argumento pasado por referencia
istream& get(char* s, int n, char c='\n');	introduce en ***s*** a lo más ***n*** caracteres del flujo de entrada (incluyendo el '\0') o hasta que encuentra el carácter de terminación (por defecto '\n'), o el fin de fichero. No retira el carácter de terminación del flujo de entrada.
istream& getline(char* s, int n, char c='\n');	lee a lo más ***n-1*** caracteres del flujo de entrada o hasta que encuentra el carácter de terminación (por defecto '\n') o hasta el fin de fichero. Retira el carácter de terminación del flujo de entrada, pero no lo almacena.
istream& read(char* s, int n);	lee n bytes del flujo de entrada y los deposita en la cadena ***s***. Se utiliza para entrada binaria.
istream& ignore(int n=1, int delim=EOF);	ignora o descarta los ***n*** caracteres siguientes del flujo de entrada, o hasta que encuentra el carácter de terminación (por defecto el fin de fichero EOF).
istream& putback(char c);	devuelve el carácter ***c*** al flujo de entrada
int peek();	lee un carácter del flujo de entrada pero sin retirarlo de dicho flujo; lo devuelve como valor de retorno.

La mayor parte de las funciones anteriores devuelven una referencia al flujo de entrada o de salida. Como se verá un poco más adelante, esta referencia puede utilizarse para detectar errores o la condición de fin de fichero.

Considérese el siguiente ejemplo de lectura de un fichero:

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

void main()
{
    char frase[81];
    fstream fi;
    fi.open("datos.txt", ios::in);

    while(fi.getline(frase, 80) != NULL)
        cout << frase;
}
```

Para conocer con más detalle cómo se utilizan estás funciones acudir a alguno de los textos de C++ recomendados en la Bibliografía o al ***Help online***.

6.5.2 FUNCIONES MIEMBRO DE FSTREAM

La clase ***fstream*** tiene algunas funciones miembro interesantes, tales como las siguientes:

fstream(); constructor por defecto de la clase. Construye un flujo sin abrir ningún fichero. El fichero puede ser abierto más tarde con la función ***open()***.

fstream(const char* filename, int nMode, int nProt = filebuf::openprot); constructor general que crea un flujo al fichero cuyo nombre se indica, del modo indicado (***ios::in***, ***ios::out***, etc.), y con la protección indicada. La protección por defecto es **filebuf::openprot**, que equivale a **filebuf::sh_compat**, en MS-DOS. Otros posibles modos de protección son los siguientes:

- **filebuf::sh_compat** modo compatible (MS-DOS).
- **filebuf::sh_none** modo exclusivo — no se comparte.
- **filebuf::sh_read** se permite compartir para lectura.
- **filebuf::sh_write** se permite compartir para escritura.

void open(const char* filename, int nMode, int nProt = filebuf::openprot); función miembro que abre un fichero en el modo y con la protección indicados. Sus argumentos son los mismos que los de ***fstream()***.

void close(); función miembro que cierra el fichero asociado con un flujo sin destruir el flujo creado anteriormente.

Para más información sobre estas y otras funciones miembro consultar un libro de C++ o el ***Help online***.

6.5.3 EJEMPLO COMPLETO DE LECTURA Y ESCRITURA EN UN FICHERO

A continuación se muestra un programa sencillo que le y escribe en un fichero llamado ***datos5.d***, que contiene una frase y un número separados por un carácter punto y coma, tal como se muestra a continuación:

```
Estamos en la linea numero; 17
```

El programa lee la frase y el número, incrementa este último en una unidad y los vuelve a escribir en el mismo fichero. El programa es como sigue (los comentarios del código explican lo que se va haciendo):

```
// fichero a incluir para I/O en ficheros
#include <fstream.h>

void main(void)
{
    char text[81];
    long n=0;

    // prueba de escritura en disco
    // se lee un número en un fichero datos5.d
    // se crea un flujo de entrada y se asocia con un fichero
    ifstream filein;
    filein.open("datos5.d", ios::in);
    // se lee hasta el carácter (;)
    filein.getline(text, 81, ';');
    // se lee el número
    filein >> n;
    // se cierra el fichero
    filein.close();

    // se imprime el el texto y el número por pantalla
    cout << "El texto leido es: " << "\"" << text << "\""
         << "\ny el numero leido es: " << n << endl;
    // se modifica el número leído y se vuelve a escribir
    n++;
    // se crea un flujo de salida y se asocia con un fichero
    ofstream fileout;
    fileout.open("datos5.d", ios::out);
```

```
    // se escribe el mismo texto y el nuevo número
    fileout << text << "; " << n << endl;
    fileout.close();

    cout << "Ya he terminado"  << endl;
}
```

6.5.4 ERRORES DE ENTRADA/SALIDA

Al leer y escribir de ficheros es frecuente que se produzcan errores, como por ejemplo no encontrar el fichero o no poderlo abrir, o al menos situaciones de excepción, tales como el haber llegado al fin del fichero. Es importante saber cómo se pueden detectar estas situaciones con C++.

La clase ***ios*** define una variable enum llamada io_state con los siguientes valores: goodbit, eofbit, badbit y failbit. Cada flujo de entrada/salida mantiene información sobre los errores que se hayan podido producir. Esta información se puede chequear con las siguientes funciones:

int good (); devuelve un valor distinto de cero (true) si no ha habido errores (si todos los bits de error están en off).

int eof(); devuelve un valor distinto de cero si se ha llegado al fin del fichero.

int bad(); devuelve un valor distinto de cero si ha habido un error de E/S serio. No se puede continuar en esas condiciones.

int fail(); devuelve un valor distinto de cero si ha habido cualquier error de E/S distinto de EOF. Si una llamada a ***bad()*** devuelve 0 (no error de ese tipo), el error puede no ser grave y la lectura puede proseguir después de llamar a la función ***clear()***.

int clear(); se borran los bits de error que pueda haber activados.

Además, tanto los operadores sobrecargados (<< y >>) como las funciones miembro de E/S devuelven referencias al flujo correspondiente y esta referencia puede chequearse con un ***if*** o en la condición de un ***while*** para saber si se ha producido un error o una condición de fin de fichero. Por ejemplo, las siguientes construcciones pueden utilizarse en C++:

```
while (cin.get(ch)) {
    s[i++]=ch;
}
```

o bien, de una forma distinta,

```
while (cin << ch) {
    s[i++]=ch;
}
```

ya que si el valor de retorno de (***cin.get(ch)***) o de (***cin<<ch***) no es nulo, es que no se ha producido error ni se ha llegado al fin de fichero.

Si lo que se quiere comprobar es si se ha llegado al final del fichero se puede comprobar la condición,

```
if (cin.get(ch).eof()) {
    // se ha llegado al final del fichero

}
```

y si por el contrario lo que se desea es hacer algo si no se ha tenido ningún error, se puede utilizar el ***operador negación (¡)*** que devuelve un resultado distinto de cero (true) si ***failbit*** o ***badbit*** están activados.

```
if (¡cin.get(ch)) {
    // hay un error de tipo failbit o badbit.

}
```

El operador negación se puede utilizar también en la forma siguiente, para saber si un fichero se ha podido abrir correctamente:

```
ifstream filein("datos.d");
if (¡filein) {
    cerr << "no se ha podido abrir el fichero." << endl;
    exit(1);
}
```

7 OPCIONES AVENZADAS: PLANTILLAS (TEMPLATES) Y MANEJO DE EXCEPCIONES

7.1 Plantillas

La ***generalidad*** es una propiedad que permite definir una clase o una función sin tener que ***especificar el tipo*** de todos o alguno de sus miembros. Esta propiedad no es imprescindible en un lenguaje de programación orientado a objetos y ni siquiera es una de sus características. Esta característica del C++ apareció mucho más tarde que el resto del lenguaje, al final de la década de los ochenta. Esta generalidad se alcanza con las ***plantillas*** (***templates***).

La utilidad principal de este tipo de clases o funciones es la de ***agrupar variables cuyo tipo no esté predeterminado***. Así el funcionamiento de una ***pila***, una ***cola***, una ***lista***, un ***conjunto***, un ***diccionario*** o un ***array*** es el mismo independientemente del tipo de datos que almacene (*int*, *long*, *double*, *char*, u *objetos* de una clase definida por el usuario). En definitiva ***estas clases se definen independientemente del tipo de variables*** que vayan a contener y es el usuario de la clase el que ***debe indicar ese tipo*** en el momento de ***crear un objeto*** de esa clase.

7.1.1 PLANTILLAS DE FUNCIONES

Supóngase que se quiere crear una función que devolviese el ***mínimo*** entre dos valores independientemente de su tipo (se supone que ambos tienen el mismo tipo). Se podría pensar en definir la función tantas veces como tipos de datos se puedan presentar (***int***, ***long***, ***float***, ***double***, etc.). Aunque esto es posible, éste es un caso ideal para aplicar ***plantillas de funciones***. Esto se puede hacer de la siguiente manera:

```
// Declaración de la plantilla de función
template <class T> T minimo( T a, T b);
```

En ese caso con **<classT>** se está indicando que se trata de una plantilla cuyo parámetro va a ser el tipo ***T*** y que tanto el ***valor de retorno*** como cada uno de los dos ***argumentos*** va a ser de este tipo de dato ***T***. En la definición y declaración de la plantilla puede ser que se necesite utilizar mas de un tipo de dato e incluido algún otro parámetro constante que pueda ser utilizado en las declaraciones. Por ejemplo, si hubiera que pasar dos tipos a la plantilla, se podría escribir:

```
// Declaración de la plantilla de función con dos tipos de datos
template <class T1, class T2> void combinar(T1 a, T2 b);
```

Podría darse el caso también de que alguno de los argumentos o el valor de retorno fuese de un tipo de dato constante y conocido. En ese caso se indicaría explícitamente como en una función convencional.

La definición de la ***plantilla de función*** es como sigue:

```
// Definición de la plantilla de función
template <class T> T minimo(T a, T b)
{
    if(a <= b)
        return a;
    else
        return b;
}
```

A continuación se presenta un programa principal que utiliza la ***plantilla de función*** recién definida:

```
#include <iostream.h>
template <class T> T minimo(T a, T b);

void main(void)
{
    int euno=1;
    int edos=5;
    cout << minimo(euno, edos) << endl;

    long luno=1;
    long ldos=5;
    cout << minimo(luno, ldos) << endl;

    char cuno='a';
    char cdos='d';
    cout << minimo(cuno, cdos) << endl;

    double duno=1.8;
    double ddos=1.9;
    cout << minimo(duno, ddos) << endl;
}
```

La ejecución del programa anterior demuestra que el ***tipo de los argumentos*** y el ***valor de retorno*** de la función ***minimo()*** se particularizan en cada caso a los de la llamada. Es obvio también que se producirá un error si se pasan como argumentos dos variables de distinto tipo, por lo que el usuario de la ***plantilla de función*** debe ser muy cuidadoso en el paso de los argumentos.

Seguidamente se presenta un nuevo ejemplo de función para permutar el valor de dos variables:

```
#include <iostream.h>
template <class S> void permutar(S&, S&);


void main(void)
{
    int i=2, j=3;
    cout << "i=" << i << " " << "j=" << j << endl;
    permutar(i, j);
    cout << "i=" << i << " " << "j=" << j << endl;
    double x=2.5, y=3.5;
    cout << "x=" << x << " " << "y=" << y << endl;
    permutar(x, y);
    cout << "x=" << x << " " << "y=" << y << endl;
}

template <class S> void permutar(S& a, S& b)
{   S temp;
    temp = a;
    a = b;
    b = temp;
}
```

7.1.2 PLANTILLAS DE CLASES

De una manera semejante a como se hace para las ***funciones*** se puede generalizar para el caso de las clases por medio de ***plantillas de clases***. Se definirá un ***parámetro*** que indicará el tipo de datos con los que más adelante se crearán los ***objetos***. Se presenta a continuación un ejemplo completo de utilización de ***plantillas de clases*** basado en una pila muy simple (sin listas vinculadas y sin reserva dinámica de memoria):

```
// fichero Pila.h
template <class T>

// declaración de la clase
class Pila
{
public:
    Pila(int nelem=10);    // constructor
    void Poner(T);
    void Imprimir();

private:
    int nelementos;
    T*  cadena;
    int limite;
};

// definición del constructor
template <class T> Pila<T>::Pila(int nelem)
{
    nelementos = nelem;
    cadena = new T(nelementos);
    limite = 0;
};

// definición de las funciones miembro
template <class T> void Pila<T>::Poner(T elem)
{
    if (limite < nelementos)
        cadena[limite++] = elem;
};

template <class T> void Pila<T>::Imprimir()
{
    int i;
    for (i=0; i<limite; i++)
        cout << cadena[i] << endl;
};
```

El programa principal puede ser el que sigue:

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


void main()
{
    Pila <int> p1(6);

    p1.Poner(2);
    p1.Poner(4);
    p1.Imprimir();

    Pila <char> p2(6);

    p2.Poner('a');
    p2.Poner('b');
    p2.Imprimir();
}
```

En este programa principal se definen dos objetos ***p1*** y ***p2*** de la clase ***Pila***. En ***p1*** el parámetro ***T*** vale ***int*** y en ***p2*** ese parámetro vale ***char***. El funcionamiento de todas las variables y funciones miembro se particulariza en cada caso para esos tipos de variable. Es necesario recordar de nuevo que el usuario de este tipo de clases debe poner un muy especial cuidado en pasar siempre el tipo de argumento correcto.

7.1.3 PLANTILLAS VS. POLIMORFISMO.

Puede pensarse que las ***plantillas*** y el ***polimorfismo*** son dos utilidades que se excluyen mutuamente. Aunque es verdad que el parecido entre ambas es grande, hay también algunas diferencias que pueden hacer necesarias ambas características. El ***polimorfismo*** necesita ***punteros*** y su ***generalidad*** se limita a jerarquías. Recuérdese que el ***polimorfismo*** se basa en que en el momento de compilación se desconoce a qué ***clase*** de la jerarquía va a apuntar un puntero que se ha definido como puntero a la clase base. Desde este punto de vista las ***plantillas*** pueden considerarse como una ampliación del ***polimorfismo***.

Una desventaja de las ***plantillas*** es que tienden a crear un código ejecutable grande porque se crean tantas versiones de las funciones como son necesarias.

7.2 Manejo de Excepciones

En algunos programas complejos que realicen funciones críticas puede ser necesario controlar todos los detalles de su ejecución. Es relativamente normal que un programa falle durante su ejecución debido a que se haya realizado o intentado realizar una operación no permitida (por ejemplo, una división por cero), porque se haya excedido el rango de un vector, …

Si en tiempo de ejecución se produce un error de éstos, se pueden adoptar varias estrategias. Una consiste en no hacer nada, dejando que el ordenador emita un mensaje de error y finalizando la ejecución del programa bruscamente, con lo que la información referente al error producido es mínima. Otra posibilidad es obligar al programa a continuar con la ejecución arrastrando el error, lo que puede ser interesante en el caso de que se tenga la seguridad de que ese error no se va a propagar y que los resultados que se obtengan al final seguirán siendo fiables. Otra posibilidad es tratar de corregir el error y de seguir con la ejecución del programa.

C++ dispone de una herramienta adicional que permite terminar la ejecución del programa de una manera más ordenada, proporcionando la información que se requiera para detectar la fuente del error que se haya producido. Esta herramienta recibe el nombre de ***mecanismo de excepciones***. Consta básicamente de dos etapas: una inicial o de ***lanzamiento*** (***throw***) de la ***excepción***, y otra de gestión o ***captura*** (***handle***) de la misma. ***Lanzar*** una ***excepción*** es señalar que se ha producido una determinada situación de error.

Para ***lanzar una excepción*** se coloca la porción de código que se desea controlar dentro de un bloque ***try***. Si en un momento dado, dentro de ese bloque ***try*** se pasa por una instrucción ***throw*** consecuencia de un determinado error, la ejecución acude al ***gestor de excepciones*** correspondiente, que deberá haberse definido a continuación del bloque ***try***. En este ***gestor*** se habrán definido las operaciones que se deban realizar en el caso de producirse un error de cada tipo, que pueden ser emitir mensajes, almacenar información en ficheros, …

A continuación se presenta un ejemplo muy sencillo de ***manejo de excepciones***: supóngase el caso de un programa para resolución de ***ecuaciones de segundo grado*** del tipo $ax^2+bx+c=0$. Se desean controlar dos tipos de posible error: que el coeficiente del término de segundo grado (***a***) sea 0, y que la ecuación tenga soluciones imaginarias ($b^2-4ac<0$).

Para ello se realiza la llamada a la función de resolución ***raices*** dentro de un bloque ***try*** y al final de éste se define el gestor ***catch*** que se activará si el programa pasa por alguna de las sentencias ***throw*** que están incluidas en la función ***raices***. En el caso de producirse alguno de los errores controlados se imprimirá en pantalla el mensaje correspondiente:

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

void raices(const double a, const double b, const double c);
enum error{NO_REALES, PRIMERO};

void main(void)
{
    try {
        raices(1.0, 2.0, 1.0);  // dentro de raices() se lanza la excepción
        raices(2.0, 1.0, 2.0);
    }
    catch (error e) {         // e es una variable enum de tipo error
        switch(e){
            case NO_REALES:
                cout << "No Reales" << endl;
                break;
            case PRIMERO:
                cout << "Primero Nulo" << endl;
                break;
        }
    }
}

void raices(const double a, const double b, const double c)
//    throw(error);
{
    double disc, r1, r2;
    if (b*b<4*a*c)
        throw NO_REALES;      // se lanza un error
    if(a==0)
        throw PRIMERO;
    disc=sqrt(b*b-4*a*c);
    r1 = (-b-disc)/(2*a);
    r2 = (-b+disc)/(2*a);
    cout << "Las raíces son:" << r1 <<" y " << r2 << endl;
}
```

Es importante señalar que los únicos errores que se pueden controlar son los que se han producido dentro del propio programa, conocidos como ***errores síncronos***. Es imposible el manejo de errores debidos a un mal funcionamiento del sistema por cortes de luz, bloqueos del ordenador, etc.

La gestión de errores es mucho más compleja de lo aquí mostrado. El lector interesado deberá acudir a un buen manual de C++.

8 BIBLIOGRAFÍA

1. García de Jalón, J. y otros autores, Aprenda ANSI C como si estuviera en Primero, ESIISS, 1995.
2. Graham, N., LEARNING C++, McGraw Hill, 1992.
3. Schildt, H., C++: Guía de Autoenseñanza, McGraw-Hill, 2ª edición, 1995.
4. Marshall P.C. and Lomow C., C++ FAQS, Addison-Wesley Publishing Company, 1994.
5. Stroustrup, B., The C++ Programming Language, Addison-Wesley, 3ª edición, 1997.
6. Ellis, M.A. y Stroustrup, B., Manual de Referencia C++ con Anotaciones, Addison-Wesley/Díaz de Santos, Madrid, 1994.
7. Ceballos, F.J., Programación Orientada a Objetos con C++, 2ª edición, RAMA, 1997.