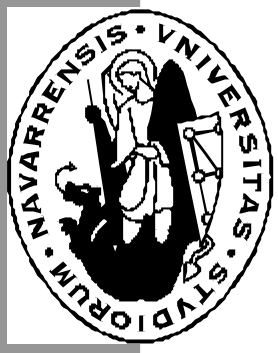


tecnun

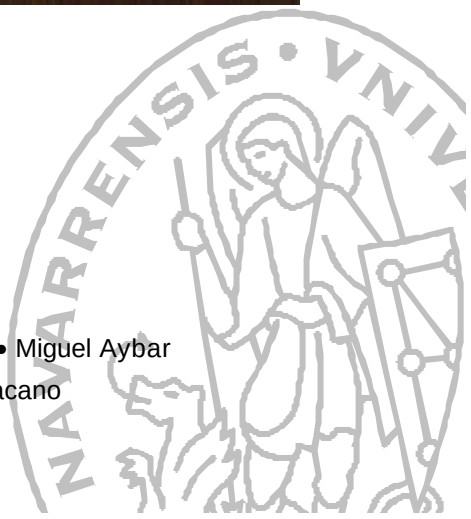
CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA
NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Escuela Superior de Ingenieros · Ingeniarien Goi Mailako Eskola

Aprenda Informática ...

Aprenda C++ Avanzado

como si estuviera en primero

San Sebastián, Febrero 2004

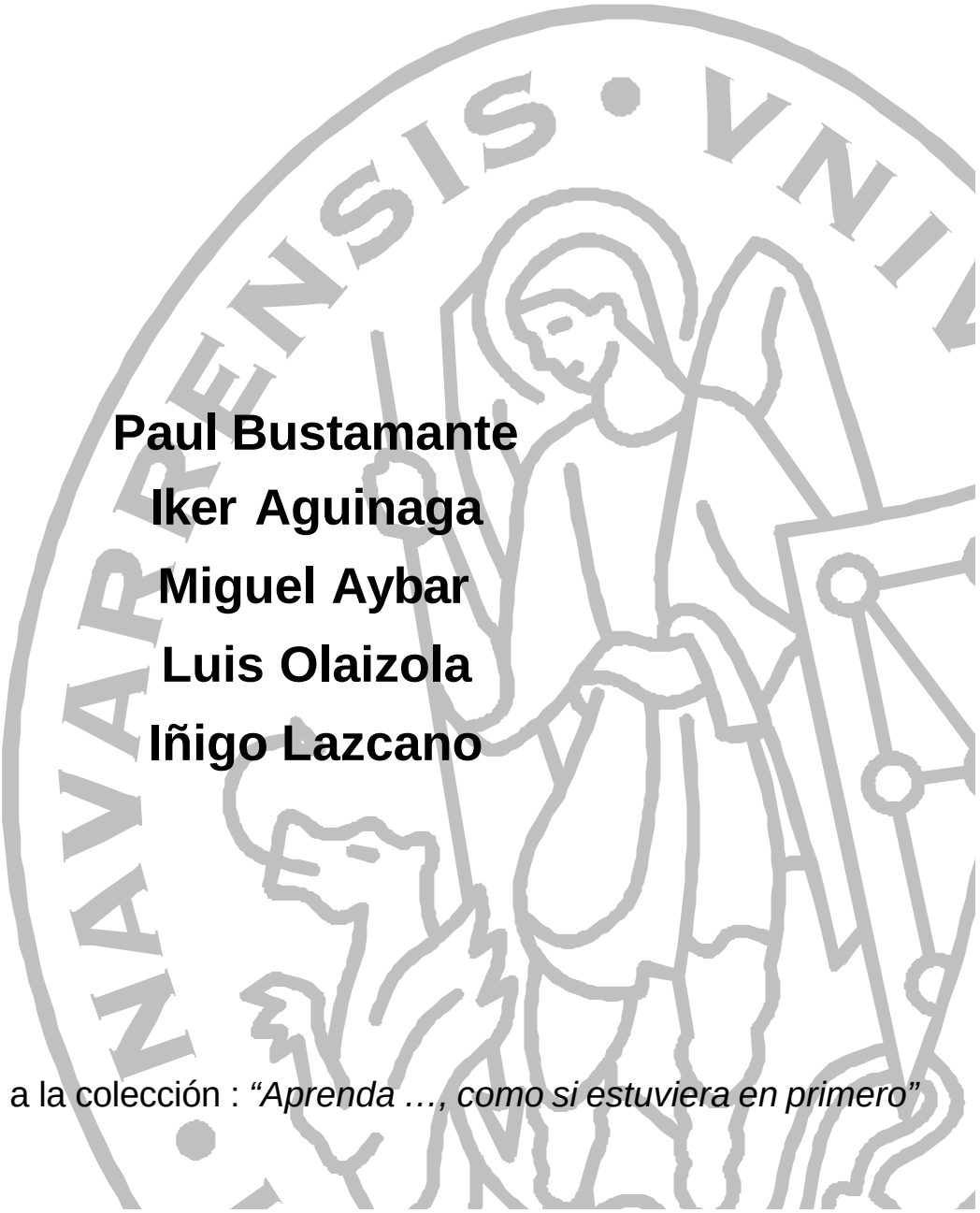


Paul Bustamante • Iker Aguinaga • Miguel Aybar
Luis Olaizola • Iñigo Lazacano

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

Aprenda C++ Avanzado

como si estuviera en primero



Paul Bustamante
Iker Aguinaga
Miguel Aybar
Luis Olaizola
Iñigo Lazcano

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

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

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

ÍNDICE

1. 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***.

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***.

1.1. Clases, Objetos y Métodos

Hasta ahora las ***funciones*** eran algo relativamente independiente de las ***variables***, y constituían el centro del lenguaje. Cualquier ***función*** se podía 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 hacía que se pudieran 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 su funcionamiento interno. 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++.

1.2. 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. 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.
- 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: 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++.
- 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***. Se verán mejor en el punto 10.9.
- 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 (::)***. Por ejemplo, la notación ***void complejo::SetData(void)*** indica que se está haciendo referencia a la función ***SetData*** definida como ***función miembro*** en la clase ***complejo***. 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.
- *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***. 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 son explícitos.
- 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, 5+2i, 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++.

1.3. 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***.

Ya se ha dicho que la principal característica de las ***clases de C++*** era que agrupan datos y funciones. En C++ las ***estructuras*** son verdaderas ***clases***. Se pueden añadir algunas funciones a las estructuras para que sean casos más reales. Estas funciones serán los ***métodos*** que permitirán interactuar con las variables de esa clase.

Así crearíamos la clase ***C_Cuenta*** de esta manera:

```
struct C_Cuenta {
    // Variables miembro
    double Saldo;          // Saldo Actual de la cuenta
    double Interes;        // Interés aplicado
    // Métodos[1]
    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***.

[1] 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.

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:

```
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 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.

1.4. 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*** por la palabra ***class***. 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 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. 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.*

1.5. Expansión *Inline*

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.

Por eso conviene recordar el uso de la expansión ***inline***, pues ofrece la solución a este problema (apartado 7.4).

Hay que mencionar, por otra parte, que la definición de las funciones ***inline*** debe hacerse en los ficheros de encabezamiento (extensión ****.h***), y no en los ficheros fuente (extensión ****.cpp***).

Cuando se declaran las funciones incluyendo su definición en la declaración, estas funciones son ***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 ); }
};
```

1.6. 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***. 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 (con 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];
```

Los operadores ***new*** y ***delete*** crean y destruyen ***objetos***, no se limitan a reservar o liberar espacio 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***, 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***).

1.7. 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++ también permite 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 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;
}
```

1.7.1. 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.

1.7.2. 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.

1.7.3. 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.

1.7.4. 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.

1.7.5. CONSTRUCTOR DE COPIA

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.***

1.7.6. 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***.

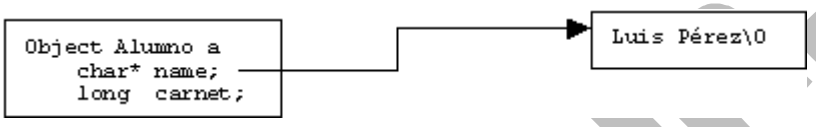


Figura 1. Objeto con reserva dinámica de memoria.

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.

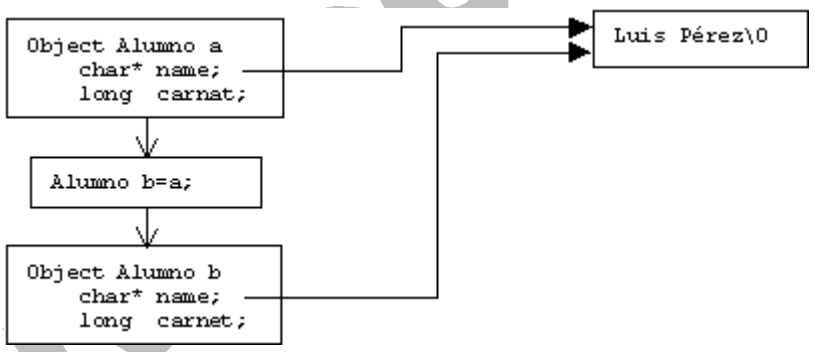


Figura 2. Copia bit a bit del objeto de la figura 1.

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 lo 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.

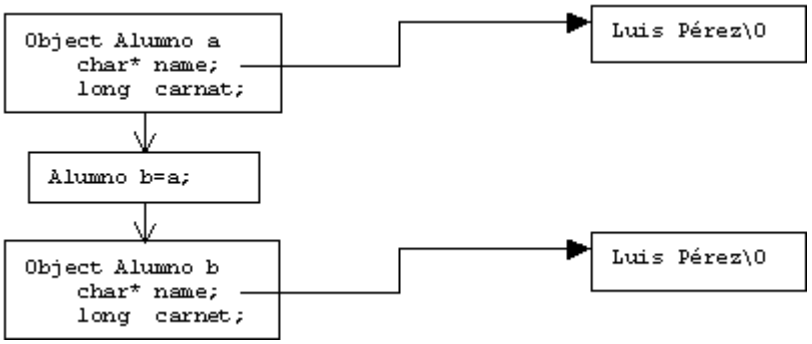


Figura 3. Copia correcta del objeto de la figura 1.

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.

1.7.7. 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++, 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.

1.7.8. 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);
    ...
}
```

1.8. 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 la ***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. 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.

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***).

Con las funciones hay que tener en cuenta que para que una 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.

1.9. 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***.

1.10. 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* (?:), el operador *sizeof*, 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 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).

1.10.1. 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 directo e intuitivo. 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>
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&); */
};
```

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***.

1.10.2. 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.

1.10.3. 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
```

1.10.4. 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 (--).

1.11. 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))
       : DNI(dni), fechaNac(birth)          // inicializadores

{
    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);
}
```

1.12. 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.
Para inicializaras con otro valor: 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*:

  ```
  tipo_de_variable 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:

```
#include <iostream.h>
#include <string.h>
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
        ~C_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; }
        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;
        }
};
```

```
//fichero main.cpp
#include <iostream.h>
#include "C_Cuenta.h"
// 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.

1.13. Funciones miembro *static*

De la misma manera que las variables,también podemos declarar funciones miembro ***static***. 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.
- No pueden utilizar el puntero ***this***, ya que éste hace referencia al objeto concreto de la clase sobre la que se está trabajando. A los miembros no estáticos sólo puede acceder empleando el operador (.) o (->).
- No pueden ser declaradas ***virtual***.

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.

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];
    //Constructor
    persona(long dni, char *name) {
        DNI = dni;
        strcpy(nombre, name);
    }
    // funcion static con argumento persona
    static void comparaDNI(persona P1, persona P2) {
        if (P1.DNI > P2.DNI ){
          cout << "El DNI de " << P2.nombre << " es anterior al de " <<
P1.nombre <<endl;
        }else{
          cout << "El DNI de " << P1.nombre << " es anterior al de " <<
P2.nombre <<endl;
        }
    }
};

// fichero main.cpp
#include "clase.h"
void main (void){
     persona p1(47126578, "Pepe Perez");
     persona p2(17684592, "Pedro Conde");
    // se llama a la función static con el nombre de la clase
     persona::comparaDNI (p1,p2);
}
```

2. HERENCIA

2.1. 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***). En C++ se puede resolver el problema de la ***pertenencia*** mediante las ***estructuras***, que pueden ser todo lo complejas que se quiera. Pero 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.

2.2. 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 ***vehículo*** 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.

2.2.1. 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>
#include <string.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
        ~C_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; }
};

//fichero main.cpp
#include "C_Cuenta.h"
#include <iostream.h>
void main(){
    C_CuentaJoven c1("Igor", 18, 10000.0, 1.0);
     C_CuentaEmpresarial c2("Juan", "MicroComputers", 10000000.0);
    cout << c1 << 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 y destructores
Funciones ***friend***
Funciones y datos estáticos de la clase
Operador de asignación (=) sobrecargado

2.3. 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.

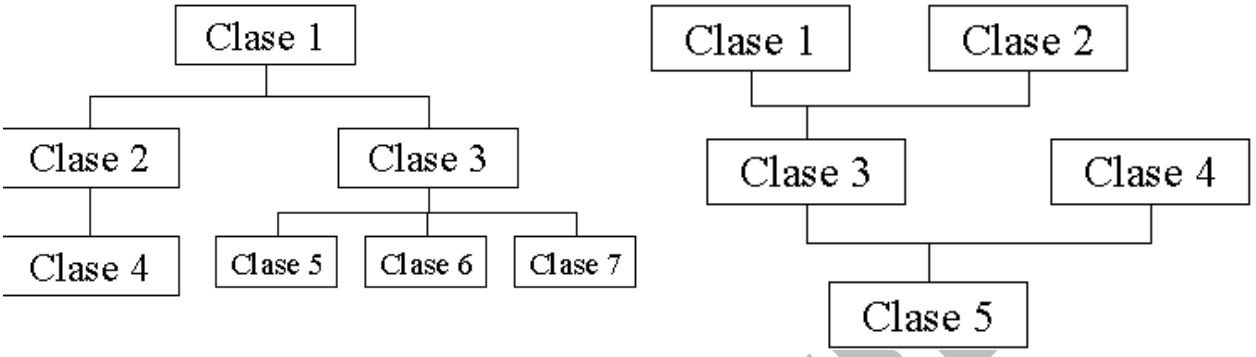
El ***inicializador base*** puede ser omitido en el caso de que la clase base tenga un ***constructor por defecto***. En el caso de que el constructor de la clase base exista, al declarar un objeto de la clase derivada se ejecuta primero el constructor de la clase base.

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){...}
```

2.4. 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, que en este caso no hay que añadir más
             código ya que los constructores de las clases base
             inicializan todas las variables miembro */
        }
```

```
        // Otros métodos
};
```

2.5. 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.

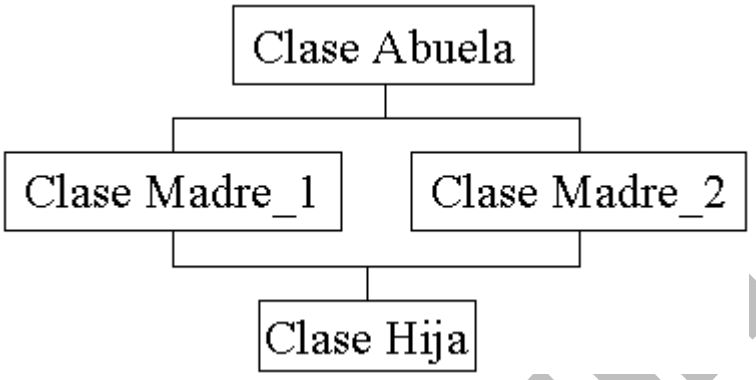


Figura 5: Necesidad de las clases base virtuales.

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 {
...
}
```

2.6. 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. En el siguiente punto se verá cómo poder evitar este problema.

3. 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:

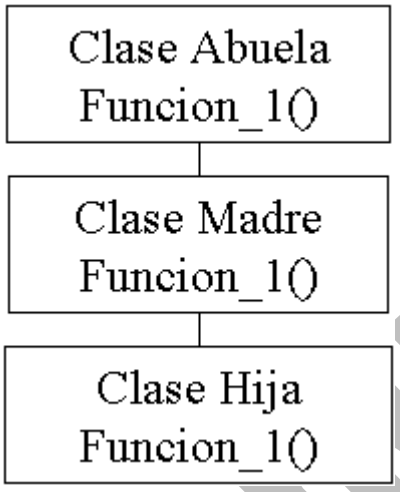


Figura 6: Funciones virtuales

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 al que apunta el puntero lo que determina la función que se llama. Esta es la esencia del ***polimorfismo***.

3.1. 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***.

3.2. 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( ) =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***.

3.3. 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***.

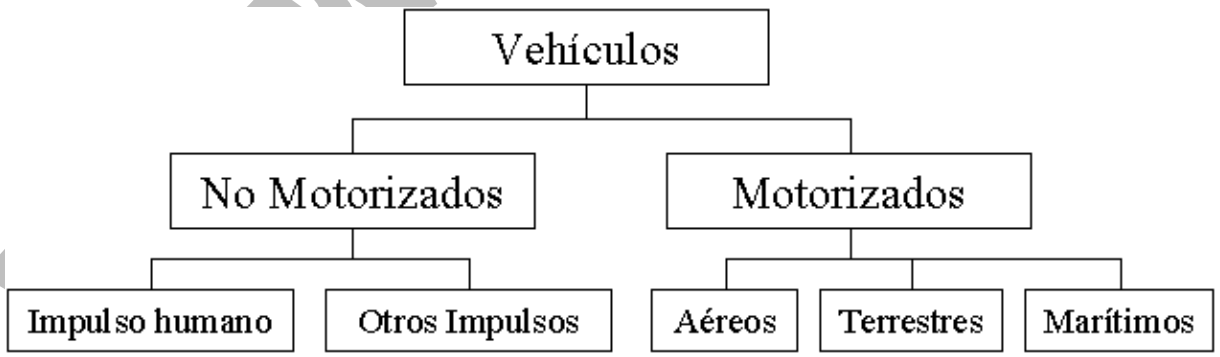


Figura 7. Clases base abstractas.

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 dases que heredan de ellas definen sus propias ***funciones virtuales***, convirtiéndose así en ***clases concretas***.

3.4. 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 eso 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.

4. 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).

4.1. 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 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***.

4.2. 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
```

4.3. 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.

4.3.1. 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.

4.4. 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 tipo_obj& a);
istream& operator>> (istream& ci, tipo_obj& 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.

4.5. 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).

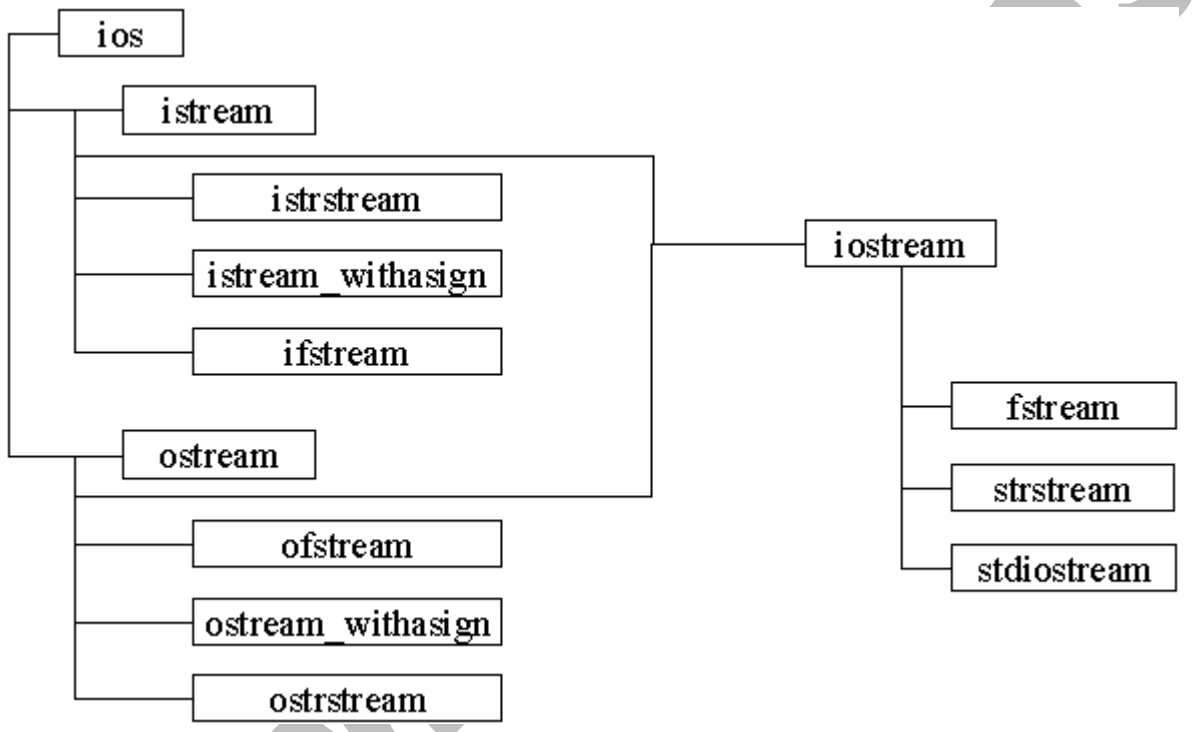


Figura 8. Jerarquía de clases de entrada/salida.

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::app	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.

4.5.1. 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***.

4.5.2. 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***.

4.5.3. EJEMPLO COMPLETO DE LECTURA Y ESCRITURA EN UN FICHERO

A continuación se muestra un programa sencillo que lee y escribe en un fichero, que tenemos que crear antes de ejecutarlo, llamado ***datos5.d***. Éste 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 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;
}
```

4.5.4. 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);
}
```

5. OPCIONES AVANZADAS: PLANTILLAS Y MANEJO DE EXCEPCIONES

5.1. Plantillas (*Templates)*

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.

5.1.1. 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& a, S& b);


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;
}
```

5.1.2. 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.

5.1.3. 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 del ***polimorfismo*** es que tiende a crear un código ejecutable grande porque se crean tantas versiones de las funciones como son necesarias.

5.2. 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 esta. ***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++.

6. LAS LIBRERÍAS DEL LENGUAJE C++

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

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

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

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

7. BIBLIOGRAFÍA

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