2 - Clases y objetos en C++

Introducción

Es sabido que en programación existen dos entes fundamentales: los **datos** y las **operaciones** que se realizan con los datos. En la programación estructurada, definir funciones está indicado para encapsular operaciones y hacerlas más sencillas, pero se definen independientes de los datos, aunque las funciones se apliquen sobre los mismos.

Si nos fijamos en el mundo real, si tenemos una lista, se nos ocurrirá crearla, recorrerla, borrarla, etc. Si se tiene una pantalla, podremos dibujar, borrar, colorear, etc. Esto nos encamina a replantearnos nuestro antiguo paradigma de programación, que era hacer pequeñas operaciones que vamos combinando para hacer grandes programas. Ahora nos podemos plantear otro paradigma: vemos unos datos sobre los que vamos a trabajar y definimos esos datos por las operaciones que se pueden realizar sobre ellos. Por ejemplo, una pila ya no es una pila porque ocupe posiciones consecutivas decrecientes en memoria, sino porque se han definido sobre ella unas operaciones ***push***, ***pop*** y ***top*** que hacen que sea una pila.

Todo lo anterior nos conduce a la Programación Orientada a Objetos (POO). El conjunto de datos y operaciones forman un conjunto que en C++ se conoce como **clase**. Una clase es un nuevo tipo de dato.

Siguiendo con la similitud entre clase y tipo, al igual que de un tipo podemos definir variables, de una clase podemos crear (***instanciar*** en terminología POO) objetos. El **objeto** es a la clase lo que la variable es al tipo.

Pero una clase es más que datos y operaciones. Una clase también **permite ocultar** estos datos y operaciones al resto del programa, como se verá en este tema.

Definición de clases y objetos en C++

Un **Tipo Abstracto de Dato** (**TAD**) es un tipo de datos que incluye datos, funciones que manipulan dichos datos, y un método de encapsular los detalles. Se puede afirmar que es el centro de la POO.

La principal forma de crear TAD en **C++** es mediante un nuevo tipo de dato definido por el usuario, **la clase**. Una clase contiene tanto datos como funciones para manejarlos. Realizando un símil matemático, una clase se puede representar por la siguiente ecuación:

clase = datos + funciones

En el momento que se crea una clase se está creando un **nuevo tipo de dato**, así como las operaciones que pueden desarrollarse con esos datos. Pero además es una forma de **empaquetar datos y las funciones** que utilizan los datos, pudiendo ocultar los datos al resto del programa[1].

El concepto de clase en C++ ofrece al programador una herramienta para crear nuevos tipos abstractos de datos.

Estos tipos se comportan de forma similar a los tipos estándar del lenguaje, ya que sólo se diferencian de los tipos elementales en la forma de crearlos, no en la de usarlos.

[1] Por eso se dice que una clase distingue interfaz de implementación. Para usarla será necesario conocer su interfaz, pero se podrá obviar su forma de implementar las funciones en ella contempladas.

Una **clase** es sintácticamente similar a una **estructura**, formada por

- Uno o más elementos dato –miembros dato o **atributos**–
- Ninguna, una o más funciones que manipulan dichos datos – funciones miembro o **métodos**–

Cuando hablemos simplemente de "miembros" estaremos haciendo referencia tanto a los atributos como a los métodos de la clase.

Nomenclatura POO	Nomenclatura C++
atributo	miembro dato
método	función miembro

Control de acceso a una clase

Los miembros pueden ser visibles fuera de la clase si se definen como **public**, o no visibles si se definen como **private**. De esto se deduce que dentro de una clase se van a tener dos secciones diferenciadas:

- La **sección privada**: Es la sección de miembros de la clase a la que no tiene acceso el exterior de la clase. Está oculta para los accesos desde fuera de la clase. Es como si se construyera un muro alrededor de los miembros de la clase para protegerlos de errores accidentales del resto del programa. Esta sección se marca con la palabra reservada **private**.

- La **sección pública**: Es la sección de miembros de la clase a la que se tiene acceso desde el exterior de la clase. Esta sección se marca con la palabra reservada **public**.

Es correcto incluir datos y funciones en la sección privada, y más datos adicionales y funciones en la parte pública. Pero en la mayoría de los casos prácticos, los datos se incluyen sólo en la sección privada, y en la sección pública se declaren funciones para acceder a ellos.

Existe una **tercera sección** denominada **sección protegida**. Los miembros (datos y funciones) que aquí se declaran son privados para el exterior, excepto para las clases que se derivan de una clase. Se verá con detalle en capítulos posteriores cuando se trate el tema de la herencia.

Declaración de una clase - Interfaz de la clase

Una clase puede ser declarada de tres formas:

1ª) Mediante la palabra **struct**. Por defecto todos los ***miembros*** son ***públicos***.

```
struct  cuadrado {
    double Lado1;
    double Lado2;
    double CalcularArea();
    void Leerdatos (double, double);
};
```

2ª) Mediante la palabra **union**. Por defecto los ***miembros*** son ***públicos*** y los datos comparten espacio de memoria.

```
union  Nombre_Persona {
    char Nombre_Completo[30];
    char Nombre_y_Apellido[2][15];
    void MuestraNombre();
    void MuestraApellido();
};
```

3ª) Mediante la palabra **class**. Esta es la forma usual de declarar clases. Los ***miembros*** son ***privados*** por defecto.

```
class vehículo {
   int Numero_Ruedas;
   int Numero_Ocupantes;
   void MostrarNumeroOcupantes();
};
```

Con las palabras **public**, **private** y **protected** pueden declararse respectivamente como públicos, privados o protegidos los miembros de una clase, por lo que la sintaxis general de la definición de una clase, centrándonos únicamente en el uso de **class**, es:

```
class <identificador> {
     [ public:
          [ <miembros dato> ]
          [ <funciones miembro> ] ]
     [ protected:
          [ <miembros dato> ]
          [ <funciones miembro> ] ]
     [ private:
          [ <miembros dato> ]
          [ <funciones miembro> ] ]
};
```

Las partes **public**, **protected** y **private** en la definición de una clase pueden aparecer o no, y no importa su orden.
Para una clase existen tres tipos de usuarios:

- la propia clase
- usuarios genéricos
- clases derivadas (de las cuales todavía no se hablado)

Cada nivel de privilegio tiene asociada una palabra reservada:

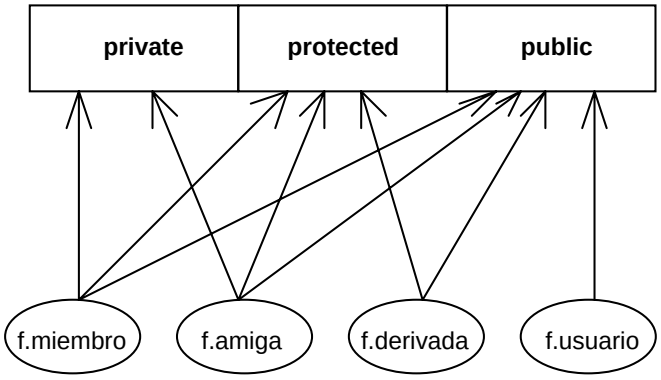
- ***private***: sólo la propia clase
- ***public***: cualquier usuario
- ***protected***: clases derivadas

Las funciones miembro de la clase tienen acceso a todo lo declarado dentro de ésta. También tienen este privilegio un tipo de funciones que no pertenecen a la clase pero que se declaran como amigas (***friends***). Éstas se verán más adelante.

Desde el exterior se puede acceder a los miembros públicos de la misma manera que se accede a los campos de una estructura en **ANSI C**.

Las funciones miembro de clases derivadas (se verán en el tema de herencia) sólo pueden acceder a los miembros declarados como públicos o protegidos de la clase base. Vistos desde el exterior los miembros *protegidos* son como *privados.*
Dicho de otra forma:

- Los **miembros privados** de la clase sólo pueden ser accedidos por:
  - Las funciones miembro de la propia clase
  - Las funciones amigas de la propia clase
- Los **miembros protegidos** de la clase sólo pueden ser accedidos por:
  - Las funciones miembro de la propia clase
  - Las funciones amigas de la propia clase
  - Las funciones miembro de clases derivadas
- Los **miembros públicos** de la clase pueden ser accedidos por:
  - Las funciones miembro de la propia clase
  - Las funciones amigas de la propia clase
  - Las funciones miembro de clases derivadas
  - Cualquier usuario de la clase



Resumen de los modos de acceso a miembros

Mediante el uso correcto de las tres secciones de la clase, se consigue un concepto de suma importancia en la POO, el ***encapsulamiento***:

- Una clase debe ocultar su estructura y la implementación de sus servicios, haciendo públicas las interfaces de sus servicios.
- Sus atributos deberán ser también privados, y el acceso a éstos para el resto del programa debe producirse exclusivamente a través de los métodos públicos.
- Los métodos que sean usados para la manipulación interna de datos o como auxiliares a otros métodos de la propia clase deben ser privados.
- Los atributos de un objeto deben ser privados, y el acceso a éstos debe producirse exclusivamente a través de las funciones miembro públicas.

Definición de la clase - Implementación de la clase

Tal como se ha plasmado en los ejemplos, normalmente dentro de la declaración de la clase se incluye además de los atributos, sólo la declaración o prototipo de los métodos. La definición de los métodos se realiza fuera de la definición de la clase.

El siguiente sería un ejemplo completo en el que se incluye la declaración de una **clase fecha** junto con la definición de sus métodos.

```
// Declaración de la clase
class fecha {
   int dia, mes, año; // Por omisión, sección privada
public: // Sección pública
   void Iniciar(void);
   void Fijar(int, int, int);
   void Mostrar(void);
};

// Definición de los métodos
void fecha::Iniciar(void) {
       dia = 0;
       mes = 0;
       año = 0;
     }

void fecha::Fijar(int d, int m, int a) {
       dia = d;
       mes = m;
       año = a;
     }

void fecha::Mostrar(void) {
       cout << d << "/" << m << "/" << a << \n";
}
```

En este ejemplo se observa como dentro de la declaración de la clase (dentro de la estructura **class**) sólo se especifican los prototipos de los métodos. Luego, fuera de la clase se definen los métodos, pero se utiliza el operador de resolución de ámbito (**::**) para indicar al compilador que el método pertenece a una clase concreta. Gracias a esto pueden existir métodos con el mismo nombre en distintas clases.

Creación de objetos de una clase - Instanciación

Desde el punto de vista de C++ **un objeto es una instancia de una clase**.

Si las clases las consideramos como nuevos tipos definidos por el usuario, los objetos serían como las "variables" que definimos de estos nuevos tipos.

Una vez que se tiene declarada la clase, se pueden definir una o más instancias de ella, de la misma forma que se haría con los tipos estándar cuando definimos variables de los mismos.

Cuando se instancian varios objetos de la misma clase, los atributos de los distintos objetos son independientes.

El **empleo de una función miembro** se realiza poniendo el nombre del objeto, que se ha instanciado de la clase, seguido por el operador punto y el nombre de la función miembro.

Observemos el siguiente programa ejemplo **AGENDA.CPP**. En el mismo hay dos instancias de la clase **Registro**. La primera de ellas justo después de la declaración de la clase, y la otra en la función **main**. Estas son las dos maneras de instanciar objetos.

```
// Curso Lenguaje C++
// Programa: Demostración del uso de una clase
// Fichero:  AGENDA.CPP
#include <iostream.h>
#include <string.h>

class Registro {
    // Por omisión esto es la sección privada
    char nombre[50];
    char telefono[15];
public:     // Sección pública
    void Iniciar (void);
    void EntradaDatos (void);
    void SalidaDatos (void);
} registro1;

// Programa principal
void main(void) {
    Registro registro2;

    registro1.Iniciar();
    registro2.Iniciar();

    registro1.EntradaDatos();
    registro2.EntradaDatos();

    registro1.SalidaDatos();
    registro2.SalidaDatos();
}

// Funciones miembro de la clase Registro
void Registro::Iniciar (void) {
    strcpy(nombre, "");
    strcpy(telefono, "");
}

void Registro::EntradaDatos (void) {
    cout << "\nNombre: ";
    cin  >> nombre;
    cout << "Teléfono: ";
    cin  >> telefono;
}

void Registro::SalidaDatos (void) {
    cout << "\n\nNombre:\t\t" << nombre << "\n";
    cout << "Teléfono:\t" << telefono << "\n";
}
```

Se presenta otro ejemplo en **FECHA.CPP**.

```
// Curso Lenguaje C++
// Programa: Demostración del uso de una clase
// Fichero:  FECHA.CPP
#include <iostream.h>

// Declaración de la clase
class Fecha {
   int dia, mes, anio; // Por omisión, sección privada
public: // Sección pública
   void Iniciar(void);
   void Fijar(int, int, int);
   void Mostrar(void);
};

// Programa principal
void main(void) {
    Fecha fecha;
    fecha.Iniciar();
    fecha.Fijar(1,3,2003);
    fecha.Mostrar();
}

// Definición de los métodos
void Fecha::Iniciar(void) {
       dia = 0;
       mes = 0;
       anio = 0;
     }

void Fecha::Fijar(int d, int m, int a) {
       dia = d;
       mes = m;
       anio = a;
     }

void Fecha::Mostrar(void) {
       cout << dia << "/" << mes << "/" << anio << "\n";
     }
```

Los atributos ***día***, ***mes*** y ***anio*** de la clase ***Fecha*** se han declarado privados. La única manera de asignar valores al objeto ***fecha*** es a través de la función pública ***Fijar*** (interfaz del objeto).

Clases vacías

Las clases pueden tener declaraciones vacías, aunque este no es el cometido principal.

```
class Vector {};
```

Por supuesto si se pueden crear objeto de este tipo de clases.

El motivo de crear clases vacías no es otro que cuando se está desarrollando un proyecto de programación grande, se puede tener la necesidad de comprobar el funcionamiento adecuado de las primeras versiones, en las que algunas clases puede que no estén completamente acabadas, o incluso no estén empezadas. En estos casos se

utilizan estas clases vacías para que el código compile sin errores y se pueda probar parte del código. A estas clases vacías se las suele conocer con el nombre de resguardos o *stubs*.

Clases anidadas

Dentro de una declaración de una clase se puede tener declaraciones de otras clases. Una clase que se declara en el interior de otra clase se denomina **clase anidada**, y se puede considerar como una clase miembro.

Un identificador de una clase anidada está sujeto a las mismas reglas de acceso que los restantes miembros. Si una clase anidada se declara en la sección privada de la clase *externa*, la clase anidada será visible exclusivamente para las funciones miembro y para las funciones amigas.

La clase *externa* puede acceder a la clase anidada sin el operador de resolución de ámbito, pero si una clase anidada es accesible desde una clase o función que no la contiene, se debe aplicar el operador de resolución de ámbito.

```
// Curso Lenguaje C++
// Programa: Clases anidadas
// Fichero:  ANIDA1.CPP
#include <iostream.h>

class Externa {
public:
   int i;
   class Interna {
   public:
      int x;
   };
};
void main (void) {
   Externa E1;

   E1.i=0;
   cout << E1.i << "\n";  // 0 por pantalla

   //E1.x=9;   // Error 'x' no es un miembro de Externa
   Externa::Interna I1;
   I1.x=9;
   cout << I1.x << "\n";  // 9 por pantalla
}
```

Miembros dato o atributos

Algunas consideraciones sobre los miembros datos o atributos:

- No pueden ser declarados como **auto**, **register**, o **extern.**
- Pueden ser enumeraciones, campos de bits, estructuras y cualquier otro tipo de dato intrínseco o definido por el usuario.
- Pueden contener punteros a cualquier tipo válido.
- Pueden ser objetos, siempre y cuando hayan sido previamente declaradas o definidas las clases a las que pertenecen.
- Una clase puede contener definiciones de clase (clases anidadas).
- Una clase no puede definir atributos dato que sean instancias de ella misma.
- Una clase puede contener referencias y punteros a instancias de ella misma.
- Los miembros dato de una clase no se pueden iniciar de forma explícita. Una clase no es un objeto, y por lo tanto no se asigna memoria hasta que no se produce una instancia de la clase. Por este motivo los miembros datos declaradas en una clase se pueden considerar como si fueran campos de una estructura, y no como variables.

Funciones miembro o métodos

Después de haber introducido las funciones miembro mediante el programa **AGENDA.CPP**, se está en disposición de estudiarlas más profundamente. Su cometido es llevar a cabo las acciones requeridas sobre los miembros dato de la clase.

En la declaración de las clases aparecen los prototipos de sus funciones miembro. Estos prototipos siguen la misma sintaxis que los prototipos de las funciones que no son miembro de una clase.

Las definiciones de las funciones miembro pueden estar en cualquier parte, en línea dentro de la clase (como se verá más adelante), en el mismo fichero pero fuera de la clase, o en otro fichero diferente. Es un hecho común definir las funciones miembro en un archivo de código separado al de la declaración de la clase. Al ser un módulo independiente mejora la documentación, el mantenimiento, y la ampliación.

La referencia de una función miembro dentro de la clase se hace utilizando simplemente el nombre de la misma. La referencia a una función miembro fuera de la clase, se realiza con el operador resolución de ámbito `::`. La forma de hacerlo es poner el nombre de la clase, con la que el método está asociado, como prefijo del nombre del método, y ambos nombres van separados por el operador de resolución de ámbito.

```
NombreClase::NombreMetodo
```

El operador de resolución de ámbito se puede usar también para seleccionar otra definición que no sea la que se toma por omisión. Esta es la manera de redefinir las funciones implícitas del lenguaje. Esto puede llegar a ser interesante pues las funciones implícitas se pueden seguir utilizando.

```
// Curso Lenguaje C++
// Programa: Redefinición de funciones implícitas
// Fichero:  PUTS.CPP
#include <stdio.h>
#include <iostream.h>

class Ejemplo {
public:
   void puts(char *texto);
   void escribir(char *texto);
};

void Ejemplo::puts (char *texto) {
     cout << "&&&" << texto << "&&&\n";
}

void Ejemplo::escribir (char *texto) {
   ::puts (texto);  // Se usa función puts() de stdio.h
   puts (texto);    // Se usa función Ejemplo::puts()
}

void main(void) {
   Ejemplo A;
   A.escribir("Texto de ejemplo");
   A.puts("Segundo texto de ejemplo"); // Se usa función Ejemplo::puts()
   puts("Tercer texto de ejemplo");    // Se usa función puts() de stdio.h
}
```

Otro ejemplo

Consideremos que dentro de un método de una clase hay un conflicto de nombres entre tres variables: global, el miembro de la clase y una variable local al método. Cuando coinciden identificadores, la forma de escoger entre ellos es la siguiente:

```
int a;
class clase {
   int a;
   void f(int a) {
        a = 1;        // Acceso a variable local del método f
        clase::a = 2; // Acceso al miembro de la clase
        ::a = 3;      // Acceso a la variable global
        ...
   }
};
```

De todas maneras no es aconsejable abusar de estos operadores `::`, salvo que queramos un enunciado difícil para un examen de universidad. En serio, hay miles de identificadores diferentes como para no poder usarlos sin que se repitan.

Funciones de acceso contra miembros dato públicos

Crear funciones miembro públicas para acceder a los datos privados de la clase puede parecer, especialmente cuando se empieza a programar bajo las normas de la POO, un trabajo innecesario. Puede parecer mucho más sencillo declarar los datos miembro como públicos, y manipularlos directamente.

Las ventajas de tener funciones miembro son dos:

- Aseguran que los objetos nunca tendrán valores sin sentido.
- Estas funciones permiten que los cambios en la realización de la clase puedan hacerse de una forma más sencilla. Un cambio en la estructura del programa en **C** puede significar la modificar de miles de líneas, todos los fuentes que utilizaran la estructura modificada. En **C++**, lo que hay que hacer es modificar las funciones miembro de la clase, lo cual no significará demasiado trabajo. Esto significa que el uso de funciones miembro en lugar de datos miembro públicos disminuye la cantidad de código a reescribir en el momento de cambiar o ampliar una aplicación.

Usando funciones miembro para controlar el acceso a los datos privados, se está ocultando la representación de la clase. Las funciones de acceso permiten cambiar la realización de una clase sin afectar al resto del programa. Este concepto se conoce como **encapsulamiento**, y es uno de los principios más importantes de la **POO**.

Métodos inline

Como cualquier otra función en **C++**, las funciones miembro pueden ser ***inline***. Hay dos formas para hacer que una función miembro sea ***inline.***

1. Definiendo la función dentro de la definición de la clase.

```
class ejemplo {
     <...>
public:
     tipo funcion(parametros) { <cuerpo>; }
}
```

2. Utilizando la palabra reservada **inline** en la definición del método.

```
class ejemplo {
     <...>
public:
     tipo funcion(parametros);
}

//Restricción: la implementación debe ponerse en
//el mismo archivo donde se define la clase

inline tipo ejemplo::funcion(parametros)
{
     <cuerpo>;
}
```

La declaración de una función en línea es una solicitud al compilador. Si el compilador considera que la función es demasiado complicada, bien por su tamaño o por contar con una estructura compleja, la tratará como si fuera una función ordinaria. Estas funciones alcanzan su mayor funcionalidad y sentido cuando se trata de funciones muy cortas.

Sólo se puede utilizar una función en línea en el archivo en que está definida. La razón es que el compilador necesita la definición completa de la función para poder insertar el cuerpo de la misma siempre que se llame. Por

este motivo si se utiliza un fichero cabecera para declarar una clase, y dicha clase contiene funciones miembro **inline**, éstas deben definirse en el mismo archivo cabecera.

El siguiente es un ejemplo de uso de métodos **inline**.

```
// Curso Lenguaje C++
// Programa: Métodos inline
// Fichero:  INLINE.CPP
#include <iostream.h>
#include <math.h>

class Complejo {
   float real,  // Parte real
         imag;  // Parte imaginaria
public:
   void MuestraComplejo(void) {
        // inline por definirla en la clase
            cout << real ;
           (imag<0) ? cout << " - " : cout << " + ";
           cout << fabs(imag) << "i\n";
   }

   void PideComplejo(void);
};


inline void Complejo::PideComplejo(void) {
  // inline porque se ha forzado
     cout << "Parte real:  ";
     cin  >> real;
     cout << "\nParte imaginaria:  ";
     cin  >> imag;
}

void main(void) {
   Complejo I;
   I.PideComplejo();
   I.MuestraComplejo();
}
```

Constructores y destructores

Constructores

Es normal que una parte de un objeto necesite una inicialización antes de poder usarse. Un ***constructor*** no es más que una **función miembro de la clase** que aglutinará un conjunto de instrucciones que permiten inicializar memoria asociada a los objetos de la clase. Normalmente se utilizan para iniciar el objeto cuando éste se crea.

Sus características son:

- Tiene en mismo nombre que la clase.
- Puede definirse **inline** o fuera de la declaración de la clase.
- Pueden aceptar parámetros, los cuales puede definirse con valores por defecto.
- Puede estar sobrecargado. Se puede declarar más de un constructor para una misma clase si todos ellos tienen diferente lista de argumentos. Esto es útil si se quiere iniciar los objetos de varias formas diferentes.
- No devuelven valores. No se puede especificar un valor de retorno cuando se declara un constructor, ni siquiera **void**. En consecuencia, un constructor no puede tener ninguna sentencia **return**.
- El constructor se ejecuta automáticamente cuando se crea un objeto de tipo clase. No se invoca nunca de forma explícita.
- El constructor se suele declarar en la sección pública. Si se hiciera en la sección privada sólo podría ser utilizado por los objetos que se creasen en las funciones miembro y en las funciones amigas.

Se denomina ***constructor por defecto de la clase*** al constructor que no tiene argumentos.

No es obligatorio definir constructores cuando se define una clase, pero es una buena técnica de programación hacerlo. **Si no se define ninguno**, **el compilador** de forma automática **genera** un constructor que no tiene ningún parámetro, un **constructor por defecto**, que no hace nada, no inicia los datos miembro.

A continuación se muestran unos sencillos ejemplos de constructores:

Ejemplo 1

```
//definición de la clase cola
class Cola {
   int c[100];
   int ppio, fin;
   public :
      Cola (void)       // constructor
      void put (int i);
      int get (void);
   };


// Constructor
Cola::Cola (void) {
   ppio=fin=0;
}

...
// resto de métodos
```

Ejemplo 2

```
class Circulo {
   float x_centro, y_centro;
   float radio;
public:
   Circulo(float x, float y, float r) {
      //constructor inline
        x_centro = x;
        y_centro = y;
        radio = r;
   }
};
```

En el ejemplo de la clase **Cola**, el compilador se encarga de forma automática de ejecutar el constructor al definir una instancia de la clase, como podía ser:

```
Cola cola1;
```

En el ejemplo de la clase **Circulo**, cuando se defina un objeto de dicha clase, debe tener tres argumentos. Así por ejemplo:

```
Circulo c1(10, 10, 5.5);    // Correcto
Circulo c1 = Circulo(10, 10, 5.5);  // Equivalente a lo anterior. No se usa
Circulo c2;                 // Error: Necesita argumentos
Circulo c3(25, 20);         // Error: Pocos argumentos
```

La tercera definición (**`Circulo c2;`**), da error debido a que no se ha definido el constructor por defecto (**`Circulo()`**). Se recuerda que el compilador lo crea automáticamente solamente si no hay otro constructor definido. Generalmente, cualquier clase se define con uno o varios constructores, por lo que no se creará automáticamente el constructor por defecto. De todas formas, se podría haber definido de forma explícita el constructor por defecto de la siguiente forma, para que hubiera sido válida la definición.

```
class Circulo {
   float x_centro, y_centro;
   float radio;
public:
   Circulo(void) { }; // Constructor por defecto
   Circulo(float x, float y, float r) {
        x_centro = x;
        y_centro = y;
        radio = r;
   }
};
```

Ahora ya sería correcta la definición

```
Circulo c2;                 // Usa constructor por defecto
Circulo c2 = Circulo();     // Equivalente a lo anterior. No se usa
```

La cuarta definición (**`Circulo c3(25, 20);`**) sería correcta si se hubiese definido un argumento por defecto para el parámetro **r** en el constructor. En algunos casos, el crear argumentos por defecto elimina la necesidad de tener que crear varios constructores para una misma clase.

Un ejemplo de constructor con argumentos por defecto puede ser el siguiente:

```
class Prueba {
   int a, b;
public:
   Prueba(int i, int j=20);
};


Prueba::Prueba(int i, int j) {
   a=i;
   b=j;
}
```

Otro mecanismo alternativo de paso de parámetros en los constructores es en el que existe una **lista de iniciación** que sigue a la lista de parámetros.

```
class P {
   dato d1, d2, d3;
public:
   P(dato x1, dato x2, dato x3):d1(x1), d2(x2), d3(x3) { }
...
```

El carácter **:** separa la lista de iniciación que debe estar en el mismo orden que los parámetros. El método constructor **P** anterior que incluye la lista de iniciación es equivalente a este otro:

```
P(dato x1, dato x2, dato x3) {
      d1 = x1;
      d2 = x2;
      d3 = x3;
}
```

El constructor se suele declarar en la sección pública. Si se hiciera en la sección privada sólo podría ser utilizado por los objetos que se creasen en las funciones miembro y en las funciones amigas. Un ejemplo de un constructor declarado en la sección privada puede ser el siguiente:

```
#include <iostream.h>

class Prueba {
     int a1, a2, a3;

     Prueba(int x, int y, int z) {
         a1=x;
         a2=y;
         a3=z;
     }
     // Se ha declarado un constructor privado, sólo accesible
     // desde objetos declarados en funciones miembro o por
     // los objetos declarados en las funciones amigas
```

```
public:
     Prueba(void) {
         a1=0;
         a2=0;
         a3=0;
     } // Constructor público

     void Muestra(void) {cout << a1 << " " << a2
                              << " " << a3 << "\n";}
     void Priv (int t0, int t1, int t2) {
           Prueba O1(t0, t1, t2); // Usa constructor privado
           O1.Muestra();
    }

     friend void Fr1(void);
};

void Fr1(void) {
     Prueba O2(4, 5, 6); // Usa constructor privado
     O2.Muestra();
}

void main (void) {
     Prueba O1;          // Usa constructor por defecto (público)
     O1.Muestra();       // Saca 0 0 0
     O1.Priv(1,2,3);     // Saca 1 2 3
     Fr1();              // Saca 4 5 6
}
```

Constructores de copia

Se denomina constructor copia a aquel constructor que crea un nuevo objeto, iniciándolo con los datos tomados de otro objeto ya existente. Este tipo de constructor tiene solamente **un argumento**: una **referencia constante** a un **objeto** de la **misma clase**.

Se invocan implícitamente cuando se hace una de estas cosas:

- Asignaciones entre objetos.
- Paso de parámetros-objeto por valor.

Ejemplo

```
#include <iostream.h>

class MC {
      float m, n, o, p;
public:
      MC () { m=n=o=p=0.0;}          // Constructor 1
      MC (float i) {m=n=o=p=i;}      // Constructor 2
      MC (const MC &fuente) {        // Constructor copia
            m=fuente.m;
            n=fuente.n;
            o=fuente.o;
            p=fuente.p;
      }
```

```
      void Cambiar(float a=0.0, float b=0.0, float c=0.0,
                   float d=0.0);
      void Escribir(void);
};

// Funciones miembro
void MC::Cambiar(float a, float b, float c, float d) {
      m=a;  n=b;  o=c;  p=d;
}

void MC::Escribir(void) {
      cout << "\nValor de m:  " << m;
      cout << "\nValor de n:  " << n;
      cout << "\nValor de o:  " << o;
      cout << "\nValor de p:  " << p << "\n";
}

// Programa Principal
void main (void) {
    MC o1(3),  // La primera instancia utiliza el constructor 2
       o2,     // La segunda instancia utiliza el constructor 1
       o3(o1); // La tercera utilizará el constructor copia

    o1.Escribir();       // Saca m=n=o=p=3
    o2.Escribir();       // Saca m=n=o=p=0
    o3.Escribir();       // Saca m=n=o=p=3
    o3.Cambiar(5,6,7,8); // Cambiamos los valores del objeto 3
    MC o4(o3);           // Nueva instancia de MC
                         // utiliza el constructor copia

    o4.Escribir();       // Saca por pantalla m=5, n=6, o=7, p=8
    MC o5 = o4;          // Nueva instancia de MC
                         // utiliza también el constructor copia
    o5.Escribir();       // Saca por pantalla m=5, n=6, o=7, p=8
}
```

Un constructor copia por defecto es creado por el compilador si no se declara un constructor para la clase. Este constructor por defecto copia el objeto fuente bit a bit al nuevo objeto. Esto funciona de forma correcta en la mayoría de los casos, aunque hay unos casos en los que este constructor copia por defecto dará problemas: cuando no hay punteros invocados, no hay ningún tipo de problemas, y se puede utilizar el constructor copia por defecto. Sin embargo, cuando se ven involucrados punteros, el constructor copia por defecto duplica los punteros, pero no la memoria que apuntan, con lo que se va a tener dos punteros de objetos distintos apuntando a la misma dirección. Esta situación, a no ser que se busque de forma consciente, es una fuente de errores.

Los constructores copia tienen gran importancia debido a:

- Son el único medio de hacer una copia de un objeto.
- Se emplea cuando se pasan objetos por valor a una función.
- Se emplea cuando una función devuelve un objeto.

Los dos últimos casos se verán con detalle en próximos capítulos.

Destructores

Un ***destructor*** no es más que una ***función miembro de la clase*** que aglutinará un conjunto de instrucciones que permiten realizar alguna acción cuando se destruye un objeto. Hay muchas razones por las que se puede necesitar un destructor. Por ejemplo, un objeto puede tener que liberar la memoria apuntada por un puntero, que previamente se ha reservado.

Sus características son:

- Tiene en mismo nombre que la clase precedido de tilde[2] "~".
- El destructor de un objeto se invoca automáticamente cuando el programa sale del alcance del objeto, es decir, para objetos globales, cuando termina el programa, y para objetos locales cuando llegamos al final del segmento de código correspondiente.
- El destructor también se llama automáticamente cuando se aplica el operador **delete** a un puntero que apuntaba a una clase.
- No pueden tener parámetros, por lo que no puede estar sobrecargado. En definitiva, cada clase tiene como máximo un destructor.
- No devuelve valores.
- Puede definirse **inline** o fuera de la declaración de la clase.
- El destructor se suele declarar en la sección pública.
- Normalmente se utiliza para realizar alguna acción sobre el objeto antes de que éste desaparezca.
- Si no se define un destructor para una clase, el compilador genera un destructor por defecto para la misma.
- Si no se ha definido un destructor para una clase, el compilador generará un destructor que no haga nada.

Un sencillo ejemplo de destructor:

```
class BJ {
      void *p;
public:
      BJ (unsigned int dim) { p = new char [dim]; }
      ~BJ { delete [] p; }
};
```

El siguiente ejemplo es de la clase **Cola** y sus funciones constructora y destructora (tenga en cuenta que la clase **Cola**, siendo precisos, no necesita un destructor).

```
class Cola {
     int c[100];
     int ppio, fin;
public :
     Cola (void);           // constructor
     ~Cola (void);          // destructor
     void meter (int i);
     int sacar (void);
 }
```

[2] Alt – 126.

```
//Función constructora
Cola::cola (void) {
   ppio=fin=0;
}

//Función destructora
Cola::~cola (void) {
}
```

Otro ejemplo

```
class pila_char {
      int   tam;
      char *pila;
      char *cima;
public:
      pila_char(int t) {
        cima = pila = new char[tam = t];
     }

      ~pila_char(void) {
        delete []pila;
     }

      void meter(char c) {
        *cima++ = c;    // Primero se almacena el valor de c el la posición de memoria apuntada por
                        // cima y posteriormente se incrementa la dirección contenida en cima.
          // La anterior expresión se divide en dos: primero se evalúa *cima y posteriormente cima++
     }

     char sacar(void) {
         return *--cima;   // Primero se decrementa la dirección contenida en cima y posteriormente
                           // se obtiene el valor  apuntado por la nueva dirección de cima.
          // La anterior expresión se divide en dos: primero se evalúa --cima y posteriormente *cima
     }
};

void f() {
      pila_char p(3);
}
//en este punto se invocará el destructor de la pila

void g() {
      pila_char p(3);
      //...
      delete p;
      // en este punto se invocará el destructor de la pila
}
```

Un ejemplo más

A continuación se presenta el programa **VECTORS.CPP**, en el que se ha definido una clase de vectores de enteros. Este sencillo ejemplo sirve para mostrar el uso de los constructores y destructores.

```
// Curso Lenguaje C++
// Programa: Clase de vectores. Demo de constructores y destructores
// Fichero:  VECTORS.CPP
#include <iostream.h>
#include <stdlib.h>

void error(char *msg);  // Función auxiliar

class Vector {
   int *vector;
   int sz;
public:
   Vector(int);   // Constructor
   ~Vector();     // Destructor
   int Size(void) {return sz;}
   void IntroduceValor(int, int);
   int  LeeValor(int);
};

Vector::Vector(int s) { // Constructor
   if (s<=0)
      error("\nTamaño de vector erróneo\n");
   sz=s;
   vector=new int[s];
   cout << "\nConstructor ejecutado con éxito\n";
}

Vector::~Vector() { // Destructor
   delete [] vector;
   cout << "\nDestructor ejecutado con éxito\n";
}

void Vector::IntroduceValor(int indice, int valor) {
   vector[indice]=valor;
}

inline int Vector::LeeValor(int indice) {
   return vector[indice];
}

void error(char *msg) {
   cerr << msg;
   exit(-1);
}
```

```
void main(void) {
   Vector v(25);        // Instancia de la clase vector

   for (register int i=0; i < v.Size(); i++)
      v.IntroduceValor(i, i*2);
   for (i=0; i < v.Size(); i++)
      cout << v.LeeValor(i) << " ";

   cout << "\n";

   {
      Vector x(7);  // Se instancia objeto x de la clase vector
   }               // Se destruye el objeto x
}    // Se destruye el objeto v
```

En algunos de los ejemplos anteriores, tanto en el constructor como en el destructor se han incluido unas instrucciones que sacan mensajes por pantalla. Esto en la práctica **nunca** se hace, pero en este caso se ha hecho así con el fin de que quede claro en que momento se ejecuta el constructor y en que momento se ejecuta el destructor.

Otro aspecto de interés es la función **error.** Ya que el constructor no puede devolver valor alguno, se utiliza esta función para que en caso de error ofrezca un mensaje, y aborte la ejecución del programa.

Funciones miembro que retornan una referencia

En ocasiones se pueden ver clases en C++ que declaran funciones miembro que retornan referencias a miembros dato privados, por lo que los están convirtiendo en públicos. Esto se considera una mala técnica de programación porque viola el encapsulamiento.

Como ejemplo

```
// Curso Lenguaje C++
// Programa: Funciones miembro que retornan una referencia
// Fichero:  REF4.CPP
#include <iostream.h>

int max(int, int);
int min(int, int);

class Fecha {
   int mes, dia, agno;
public:
   Fecha(int, int, int); // Constructor
   int &Mes(void);
   int ver_mes(void);
};

Fecha::Fecha(int d, int m, int a) {
   dia=d;  mes=m;  agno=a;
}
```

```
int &Fecha::Mes(void) {
   mes=max(1,mes);
   mes=min(mes, 12);
   return mes;
}

int Fecha::ver_mes(void) {
   return mes;
}

inline int max(int a, int b) {
   if (a > b)
        return a;
   else return b;
}

inline int min(int a, int b) {
   if (a < b)
        return a;
   else return b;
}

void main(void) {
   Fecha nacimiento(9, 7, 1971);

   cout << nacimiento.ver_mes()<< " ";   // Muestra 7
   cout << nacimiento.Mes()    << "\n";  // Muestra 7

   nacimiento.Mes()=4;
   cout << nacimiento.ver_mes()<< " ";   // Muestra 4
   cout << nacimiento.Mes()    << "\n";  // Muestra 4

   nacimiento.Mes()=13;
   cout << nacimiento.ver_mes()<< " ";   // Muestra 13
   cout << nacimiento.Mes()    << "\n";  // Muestra 1

   nacimiento.Mes()=-1;
   cout << nacimiento.ver_mes()<< " ";   // Muestra -1
   cout << nacimiento.Mes()    << "\n";  // Muestra 1

   cout << ++nacimiento.Mes()  << " ";   // Muestra 2
   cout << nacimiento.ver_mes()<< "\n";  // Muestra 2
}
```

Como se puede apreciar la función miembro se comporta como un dato miembro público. En consecuencia la llamada a la función **nacimiento.Mes()** puede aparecer en la parte izquierda de una asignación, de la misma manera que se fuera una variable pública. Así, por ejemplo, se puede incrementar su valor con el operador ++.
Pero lo más grave es que mediante esta técnica se puede asignar un valor erróneo a un dato miembro privado, aunque en este caso particular no ocurre porque hay una comprobación en la función **Fecha::Mes** que lo impide. En realidad, una sentencia como **nacimiento.Mes()=13** consigue introducir un valor erróneo en el atributo mes, pero la función **nacimiento.Mes()** cuando es llamada modifica el valor erróneo de **mes** antes de devolverlo (si es menor que 1 lo modifica a 1 y si es mayor de 12 lo modifica a 12).

Como conclusión no se aconseja esta técnica por las siguientes razones:

- La sintaxis es confusa para las personas que lean el programa.
- La comprobación del rango se lleva a cabo cada vez que el dato miembro es leído, y esto no es eficiente.
- Esta técnica **convierte el miembro dato privado en público**, por lo que **viola el encapsulamiento**.

Objetos y funciones miembro constantes

Objetos const y funciones miembro const

De la misma manera que se utiliza la palabra reservada **const** cuando se declara una variable, se puede usar para la declaración de un objeto. Tal declaración significa que **el objeto es una constante y ninguno de los datos miembro podrá ser modificado**. Por ejemplo:

```
const Fecha nacimiento(9, 7, 1971);
```

La declaración anterior significa que los atributos del objeto **nacimiento** de la clase **Fecha** no pueden ser modificados.

Los compiladores de C++ deshabilitan (no permiten) las llamadas a funciones miembro de los objetos **const**. Esto es así incluso en el caso de funciones miembro "**tipo get**" que no modifican el objeto.

De todas formas los métodos que no modifiquen ningún dato miembro, puede hacerse que llamen a objetos constantes. Para poder hacer esto, se debe declarar la función miembro como **const** poniendo la anterior palabra reservada después de la lista de parámetros, tanto en la declaración como en la definición de la función miembro. De esta forma se está declarando la función miembro de sólo lectura y el compilador permitirá su uso sobre objetos constantes, siempre que no detecte algún intento de modificación de sus atributos.

En el siguiente ejemplo se muestra el uso de los objetos constantes.

```
// Curso Lenguaje C++
// Programa: Objetos constantes
// Fichero:  FECHA.CPP

#include <iostream.h>

class Fecha {
   int mes, dia, agno;
public:
   Fecha(int m, int d, int a); // Constructor
   int LeeDia(void);
   int LeeMes(void) const;
// void EscribeDia(int m) const;
   void EscribeMes(int m);
   void MuestraFecha(void) const;
};

Fecha::Fecha(int d, int m, int a) { dia=d;  mes=m;  agno=a; }

int Fecha::LeeDia(void) { return dia; }

int Fecha::LeeMes(void) const { return mes; }

void Fecha::EscribeMes(int m) { mes=m; }

// void Fecha::EscribeDia(int d) const { dia=d; }
// Error: se ha definido como const y, sin embargo, modifica un tributo
```

```
void Fecha::MuestraFecha(void) const {
   cout << dia << "/" << mes << "/" << agno << "\n";
}

void main(void) {
   const Fecha nacimiento(9, 7, 1971);

// cout << nacimiento.LeeDia(); // Error. LeeDia() no es const
                                // No puede llamar a un objeto const
                                // aunque no modifique sus atributos

   cout << nacimiento.LeeMes(); // Correcto. LeeMes() es función const

// nacimiento.EscribeMes(10);   // Error. EscribeMes() no es const
                                // No puede llamar a un objeto const

   nacimiento.MuestraFecha();   // Correcto. MestraFecha() es función const
}
```

El compilador permite que se llame a las funciones miembro que son constantes **MuestraFecha** y **LeeMes** para el objeto **nacimiento**.
Pero no permite que se llame a la función **LeeDia** ya que aunque ésta no modifica ningún atributo, no ha sido declarada como función **const.**
Tampoco permite llamar a la función **EscribeMes**[3], ya que no está declarada como constante, y no se puede declarar como constante ya que modifica un dato miembro. Si se intentara declarar como constante, como se ha intentado la función miembro ***EscribeDia***, el compilador genera un error al detectar que modifica un atributo.

Como conclusión, podemos resumir:

- Definir cono **const** una función miembro que modifica un dato miembro es un error de sintaxis.
- Definir como **const** una función miembro que llama a otra no **const** de la clase es un error de sintaxis
- Llamar a una función miembro no **const** en un objeto **const** es un error de sintaxis
- Un objeto **const** no puede ser modificado por asignación, por lo que debe ser inicializado.
- Es posible sobrecargar una función miembro const con una versión no const. El compilador elige automáticamente cuál función miembro sobrecargada debe utilizarse, dependiendo de si el objeto es const o no.
- Es una buena costumbre declarar las funciones miembro como constantes siempre que sea posible. De esta forma se permite declarar instancias constantes de la clase.

[3] Este programa se ha compilado con los siguientes compiladores **Visual C/C++ 1.0 for Windows**, **Borland C/C++ 3.1 y 4.0, DJGPP 2.6.0 (DJ's GCC port to DOS), GNU C++ Compiler 2.5.8 for Linux.** Todos ellos, a excepción de los compiladores de **Borland,** generaron el correspondiente y esperado error. Pero los de **Borland** se limitaron a dar un aviso y modificaron un objeto que se había declarado como constante, lo cual es un detalle importante, que además de ir en contra de la definición de **constante**, deja toda la responsabilidad al programador.

Funciones miembro que retornan tipos const

Una función miembro que retorna un ***const*** se puede utilizar como cualquier otra constante, aunque con alguna salvedad. Un sencillo ejemplo de esto:

```
// Curso Lenguaje C++
// Programa: Funciones miembro const
// Fichero:  MCONST.CPP
#include <iostream.h>

class Prueba {
      int v;
public:
      void Definir(int z=0) {v=z;}
      const int Get(void);
};

const int Prueba::Get(void) {
      return v;
}

void main (void) {
   Prueba P;
   int a=3, b=2;
   P.Definir(a+b);
   const int indice = P.Get();
   cout << indice << "\n";    // 5 por pantalla
}
```

Un ejemplo de un uso incorrecto es la siguiente función **main**.

```
void main (void) {
   Prueba P;
   int a=3, b=2;
   P.Definir(a+b);

// int Matriz[P.Get()];     // Error: expected constant expression

   const int dimension = P.Get();
// int Matriz[dimension];   // Error: expected constant expression

   const int tamanno = 32;
   int Matriz[tamanno];     // Correcto
   for (int i=0; i<a+b; Matriz[i++]=i);
   for (i=0; i<a+b; cout << Matriz[i++] << " ");
}
```

Miembros estáticos

Atributos estáticos

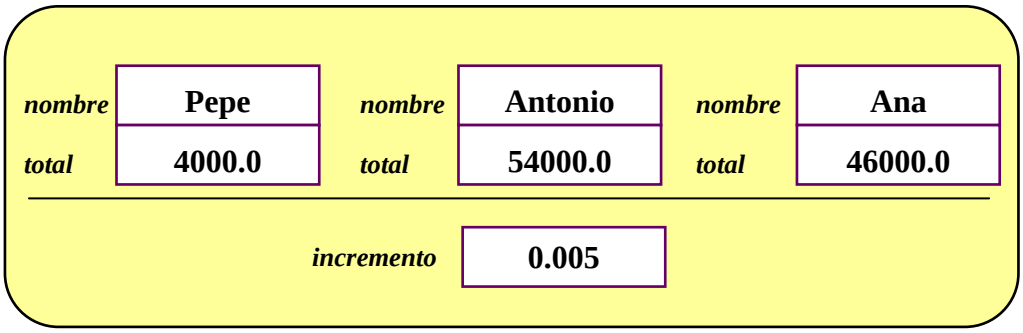
Cada vez que se define un objeto a partir de una clase concreta los elementos de datos se duplican. Es decir, los atributos de diferentes instancias de objetos de la misma clase son independientes y ocupan posiciones diferentes de memoria. Si se desea que todos los objetos instanciados de una clase compartan una sola copia de un atributo, se puede definir el atributo como ***static***. Esto se puede utilizar como una forma de comunicación entre los objetos.

Se puede declarar **private**, y sólo podrán acceder a él las funciones miembro, o **public** con lo que será accesible al resto del programa.

Un ejemplo de un miembro estático privado puede ser:

```
class Cuenta {
   char nombre[50];
   float total;
   static float incremento;
public:
   Cuenta ();
   void Interes() { total += incremento*total;}
};
```

En el ejemplo anterior sólo existe una copia de **incremento**, y es accesible por todas las instancias de la clase **Cuenta**. Cada vez que se llame al método **Interes** por cualquier objeto de la clase **Cuenta**, se empleará el mismo valor de **incremento**, compartido por todas las instancias. La siguiente figura ilustra este ejemplo suponiendo que se han definido tres objetos. Se observa como **incremento** es un miembro común a los tres objetos.



Representación de un atributo estático común a todas las instancias de una clase

Por ejemplo, si **incremento** fuera un miembro público, se podría acceder a él de la siguiente manera:

```
// Si incremento fuera público
void main () {
   Cuenta micuenta;
   micuenta.incremento=0.000154;
}
```

Sin embargo, esta sintaxis no es la más apropiada, ya que parece indicar que sólo el **incremento** del objeto **micuenta** es modificado, cuando lo que realmente ocurre es que el **incremento** de todos los objetos instanciados de la clase **Cuenta** se ven afectados por la modificación. Así la mejor manera de referenciar un miembro estático es utilizando el nombre de la clase:

```
// Si incremento fuera público
void main () {
   Cuenta::incremento=0.000154;
}
```

Se puede emplear esta sintaxis aunque no se haya definido ninguna instancia de la clase: un miembro estático existe aunque no haya instancias de la clase definidas. Pero a pesar de ello **no se puede iniciar un dato miembro en la definición de la clase, incluso aunque éste sea estático**. Por ejemplo,

```
class Cuenta {
   char nombre[50];
   float total;
   static float incremento = 0.0;  // Error
public:
   Cuenta ();
   void Interes() { total += incremento*total;}
};
```

Aunque no es un error sintáctico, **no se debe iniciar un dato miembro estático desde un constructor de clase**. El motivo es que un atributo estático normalmente sólo debe ser iniciado una sola vez, y un constructor puede ser llamado varias veces, por lo que el atributo estático sería iniciado cada vez que se llamara al constructor de la clase, por ejemplo porque se instanciara un nuevo objeto.

Por el mismo motivo anterior, **la iniciación tampoco es conveniente colocarla en un fichero cabecera** porque este fichero puede ser incluido más de una vez en un mismo programa.

Un dato miembro estático suele ser iniciado en el ámbito del fichero, como si se tratase de una variable global. Esto es válido tanto para los miembros estáticos privados como para los miembros estáticos públicos. Un lugar idóneo es en el fichero fuente que contenga las definiciones de las funciones miembro de la clase.

Un ejemplo completo del uso de datos miembro estáticos es el que sigue:

```
// Curso Lenguaje C++
// Programa: Miembros estáticos
// Fichero:  STATIC1.CPP
#include <iostream.h>

class Obj{
   static int numero;  // Iniciarlo aquí sería erróneo
public:
   Obj() { numero++;}
  ~Obj() { numero--;}

void Estado(void) {
   cout << "\nHay " << numero << " objeto";
   (numero==1) ? cout << " " : cout << "s ";
   cout << "en la clase.\n";
}
};
```

```
int Obj::numero=0; // Iniciación del miembro estático

void main (void) {
   Obj A, B;       // Se crean dos objetos

   A.Estado();     // Saca 2 por pantalla
   B.Estado();     // Saca 2 por pantalla

   Obj C;          // Se crea un tercer objeto
   A.Estado();     // Saca 3 por pantalla
   B.Estado();     // Saca 3 por pantalla
   C.Estado();     // Saca 3 por pantalla

   { Obj D;        // Se crea un cuarto objeto
     D.Estado();   // Saca 4 por pantalla
   }               // Se destruye un objeto

   C.Estado();     // Saca 3 por pantalla
}
```

Como resumen, las ventajas que ofrecen los miembros dato estáticos son:

- Permiten la comunicación entre objetos, reduciendo la necesidad de variables globales.
- Ahorra en almacenamiento, ya que el atributo estático comparte la misma zona de memoria para todas las instancias de la clase.
- Hacen evidentes los datos que pueden ser compartidos dentro de una clase.

Métodos estáticos

Si se tienen funciones miembro que sólo acceden a datos miembro estáticos de una clase, se pueden declarar éstas como **static** también. Como ejemplo se va a modificar la clase **cuenta** que se introdujo en el apartado anterior.

```
class Cuenta {
   char nombre[50];
   float total;
   static float incremento;
public:
   Cuenta ();
   void Interes() { total += incremento*total;}
   static void ActualizaIncremento( float  valor) {incremento=valor;}
};
```

Las funciones miembro estáticas se llaman con la misma sintaxis que se usa para acceder a los datos miembro estáticos, es decir es utilizando directamente el nombre de la clase:

```
void main (void) {
   Cuenta micuenta;
   Cuenta::ActualizaIncremento (0.016);
   micuenta.ActualizaIncremento (0.016);  // También válida esta sintaxis
}
```

Como un función miembro no puede actuar sobre una instancia particular de la clase, no tiene puntero **this** (se verá con detalle en próximos capítulos). En consecuencia, **una función miembro estática no puede acceder a**

ningún dato miembro que no sea estático, o **llamar a funciones miembro no estáticas**, ya que ambas operaciones llevan de forma implícita el uso del puntero **this**.

Una función miembro, estática o no, puede manejar variables globales. En el siguiente ejemplo se muestra el manejo desde una función miembro estática.

```
// Curso Lenguaje C++
// Programa: Funciones miembro estáticas
// Fichero:  STATIC2.CPP
#include <iostream.h>

int v1, v2=1, v3=2;

class Prueba {
   static int incremento;
public:
   static void suma(void) { v1=v2+v3+incremento; }
};

int Prueba::incremento = 5;

void main (void) {
   Prueba::suma();
   cout << v1 << "\n";  // Saca 8 por pantalla
}
```

Se puede acceder a una función estática mediante un puntero a función, como se muestra en el siguiente ejemplo:

```
// Curso Lenguaje C++
// Programa: Funciones miembro estáticas
// Fichero:  STATIC3.CPP
#include <iostream.h>

int v1=0, v2=1, v3=2;

class Prueba {
   static int incremento;
public:
   static void suma(void) { v1 += v2+v3+incremento; }
};

int Prueba::incremento = 5;

void main (void) {
   void (*pf)();        // Se define un puntero a función
   pf = Prueba::suma;   // Se inicia
   (*pf)();             // Se llama a la función
   cout << v1 << "\n";  // Saca 8 por pantalla
   void (*pf2)() = Prueba::suma;  // Se define un puntero a función
                                  // y se inicia
   (*pf2)();                      // Se llama de nuevo
   cout << v1 << "\n";  // Saca 16 por pantalla
}
```

Objetos como miembros de una clase

El hecho de crear nuevas clases usando otras clases como componentes recibe el nombre de **composición**. Se va a estudiar esto con un ejemplo. Supóngase que se quiere realizar una clase para tener los datos personales de los amigos (nombre, dirección, teléfono, y cumpleaños).

```
// Curso Lenguaje C++
// Programa: Objetos miembros
// Fichero:  PERSOINF.CPP
#include <iostream.h>
#include <string.h>
#include <iomanip.h>

class Fecha {
   int mes, dia, agno;
public:
   Fecha(int, int, int); // Constructor
   int LeeDia (void) const;
   int LeeMes (void) const;
   int LeeAgno(void) const;

   void EscribeDia (int);
   void EscribeMes (int);
   void EscribeAgno(int);

   void MuestraFecha(void) const;
};

class Ficha {
   char  nombre[50];
   char  direccion[75];
   char  telefono[15];
   const Fecha cumple;
public:
   Ficha (char *, char *, char *, int, int, int);
   char *LeeNombre    (void) const;
   char *LeeDireccion (void) const;
   char *LeeTelefono  (void) const;

   void EscribeNombre    (char *);
   void EscribeDireccion (char *);
   void EscribeTelefono  (char *);
   void MuestraFicha     (void) const;
};

// Funciones miembro de la clase Fecha
Fecha::Fecha(int d, int m, int a)
{ dia=d;  mes=m;  agno=a; }

inline int Fecha::LeeDia(void) const
{ return dia; }

inline int Fecha::LeeMes(void) const
{ return mes; }
```

```
inline int Fecha::LeeAgno(void) const
{ return agno; }

inline void Fecha::EscribeDia(int d)
{ dia=d; }

inline void Fecha::EscribeMes(int m)
{ mes=m; }

inline void Fecha::EscribeAgno(int a)
{ agno=a; }

inline void Fecha::MuestraFecha(void) const
{ cout << dia << "/" << mes << "/" << agno << "\n"; }


// Funciones miembro de la clase Ficha
Ficha::Ficha( char *n, char *dir, char *t,
              int d, int m, int a):cumple(d, m, a) // Inicio del miembro
{
    strncpy(nombre, n, 50);
    strncpy(direccion, dir, 75);
    strncpy(telefono, t, 15);
}


inline char *Ficha::LeeNombre(void) const
{ return (char *) nombre; }

inline char *Ficha::LeeDireccion(void) const
{ return (char *) direccion; }

inline char *Ficha::LeeTelefono(void) const
{ return (char *) telefono; }

inline void Ficha::EscribeNombre(char *n)
{ strncpy(nombre, n, 50); }

inline void Ficha::EscribeDireccion(char *d)
{ strncpy(direccion, d, 75); }

inline void Ficha::EscribeTelefono(char *t) { strncpy(telefono, t, 15); }

inline void Ficha::MuestraFicha(void) const {
    cout << nombre << "\t|| ";
    cout << direccion << "\t|| ";
    cout << telefono << "\t|| ";
    cout << setw(2) << cumple.LeeDia()  << "/"
         << setw(2) << cumple.LeeMes()  << "/"
         << setw(4) << cumple.LeeAgno() << "\n";
}
```

```
void main(void) {
   Ficha A("Antonio", "Calle Van Dyck",  "986226867",  4, 8, 1968),
         C("Carlos", "Gabriel y Galán", "912235615", 15, 7, 1968),
         M("Maria", "Cabeza de Vaca",  "958259827",  3, 5, 1970);
   A.MuestraFicha();
   C.MuestraFicha();
   M.MuestraFicha();
   A.EscribeNombre("Toñito");
   C.EscribeDireccion("Plz. Gabriel y Galán");
   M.EscribeTelefono("958259837");
   A.MuestraFicha();
   C.MuestraFicha();
   M.MuestraFicha();
}
```

Algunas notas sobre el programa:

- En la sección privada del la clase **Ficha** se tiene un miembro con el nombre **cumple** que es un objeto **Fecha**. El objeto **cumple** no será construido hasta que un objeto de la clase **Ficha** sea construido.

- Para llamar al constructor del objeto miembro se debe especificar un *miembro que inicie*. Después de la lista de parámetros del constructor de la clase, se ponen **':'** y seguidamente el nombre del miembro y la lista de argumentos. Ver la definición del constructor de la clase **Ficha**.
  Esta sintaxis fuerza que el constructor de la clase **Fecha** sea invocado para el miembro **cumple,** usando los tres argumentos especificados. El constructor de la clase **Fecha** es llamado primero que el constructor de la clase **Ficha**, y en consecuencia el miembro **cumple** se inicia antes de que se ejecute el constructor de la clase **Ficha**.
  Si la clase que se define tiene más de un objeto miembro, se puede especificar una lista de miembros que inicien, separándolos con una coma.

- Si no se utiliza un miembro que inicie, el compilador utilizará el constructor por defecto, y luego se pueden asignar los valores utilizando las funciones miembro adecuadas. Por ejemplo,

```
Ficha::Ficha( char *n, char *dir, char *t,
              int d, int m, int a) // Inicio del miembro
{
    strncpy(nombre, n, 50);
    strncpy(direccion, dir, 75);
    strncpy(telefono, t, 15);
    cumple.EscribeDia (d);
    cumple.EscribeMes (m);
    cumple.EscribeAgno(a);
}
```

- Esta técnica no es eficiente pues se inician los valores de **cumple** dos veces, una con el constructor por defecto y otra con las funciones miembro.

- De todas formas, hay dos causas por las que la anterior definición del constructor de la clase **Ficha** no es correcta en nuestro programa:
  - Como no se ha definido un constructor por defecto para la clase miembro **Fecha**, el compilador generará un error. Esto se podría evitar por ejemplo definiendo en el prototipo del constructor de la clase **Fecha** parámetros por defecto, por ejemplo:

```
Fecha(int=0, int=0, int=0); // Prototipo del Constructor
```

  - Como se ha definido el miembro **cumple** de la clase **Ficha** como constante, no es posible utilizar las funciones miembro que modifican sus atributos, es decir, no es válido utilizar:

```
cumple.EscribeDia (d);
cumple.EscribeMes (m);
cumple.EscribeAgno(a);
```

- Un miembro que inicie es requerido cuando se tenga un **objeto miembro constante**, como en nuestro caso es el miembro **cumple** de la clase **Ficha**. En este caso omitir el miembro que inicie es fatal, ya que no se pueden cambiar los valores que establezca el constructor por defecto.

Clases friend

Obsérvese el siguiente ejemplo:

```
// Curso Lenguaje C++
// Programa: Clases friend
// Fichero:  CL-FR.CPP
#include <iostream.h>
#include <string.h>

class Pepi {
   char cotilleo[100];
public:
   Pepi() { strncpy(cotilleo, "A Mary Flor la plantó su novio", 100); }
   Pepi(char *not) { strncpy(cotilleo, not, 100); }
   void leer(void) { cout << cotilleo << "\n"; }

   friend class Juani; // ==> Juani es una clase amiga de Pepi
};

class Juani {
   char noticia[100];
public:
   Juani(char *not) { strncpy(noticia, not, 100); }
   void escucha (Pepi ch) { cout << ch.cotilleo << "\n"; }

   void cambia  (Pepi ch, char *not) { // Paso por valor
      strncpy(ch.cotilleo, not, 100);
      escucha(ch);
   }
```

```
   void cambia (Pepi *ch, char *not) { // Paso por referencia
      strncpy(ch->cotilleo, not, 100);
      escucha(*ch);
   }

   void cambiaR (Pepi &ch, char *not) { // Paso como una referencia
      strncpy(ch.cotilleo, not, 100);
      escucha(ch);
   }
};


void main (void) {
   Pepi P1, P3("La vecina de arriba es rubia de bote");
   Juani J1("Yo no soy nada cotilla");

   P1.leer();       // Sale lo de Mary Flor
   J1.escucha(P1);  // Sale lo de Mary Flor
                    // Desde J1 (Juani) puede accederse a los miembros
                    // privados de P1 (Pepi)

   P3.leer();       // Sale lo de rubia de bote
   J1.cambia(P3, "Es que realmente es calva");  // Sale este mensaje
   J1.escucha(P3);  // Sale lo de rubia de bote ya que P3 se ha pasado
                    // por valor: no se modifica
   P3.leer();       // Sale lo de rubia de bote

   J1.cambia(&P3, "Es porque no tienen pelo");  // Sale este mensaje
   J1.escucha(P3);  // Sale lo de que no tiene pelo ya que P3 se ha pasado
                    // por referencia: se modifica
   P3.leer();       // Sale lo de que no tiene pelo

   J1.cambiaR(P3, "O porque se le cae muy a menudo"); // Sale este mensaje
   J1.escucha(P3);  // Sale lo de que se le cae muy a menudo ya que
                    // P3 se ha pasado como una referencia: se modifica
   P3.leer();       // Sale lo de que se le cae muy a menudo
}
```

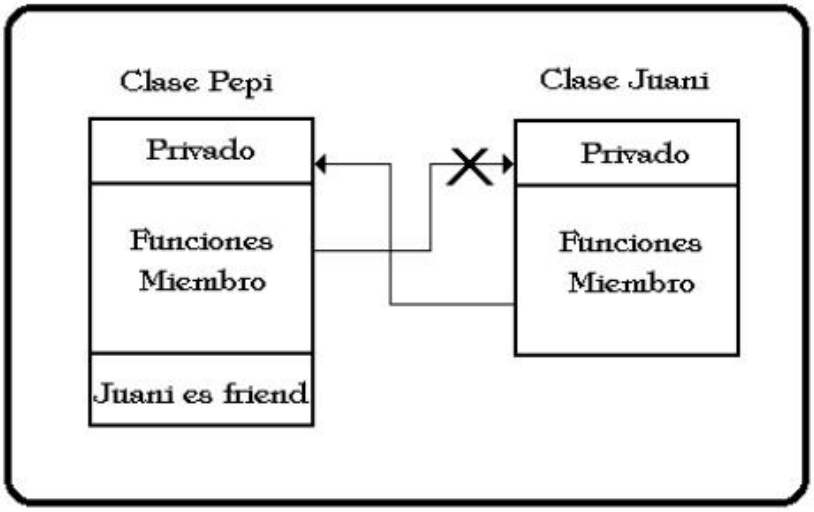
Mediante este informal ejemplo se ha querido mostrar el uso de las **clases amigas**. En él se han definido dos clases, **Pepi** y **Juani**, pero en la definición de la primera se ha declarado a **Juani** como clase amiga, empleando la palabra reservada **friend**.

```
friend class Juani;
```

Esto implica que **las funciones miembro de la clase Juani pueden leer y modificar los datos privados de la clase Pepi.**

La especificación de que clases van a ser **friend** para una determinada clase, es decir, van a tener acceso a sus datos privados, se debe hacer explícitamente en la declaración de dicha clase.
La declaración **friend** no se asocia ni a la sección **privada** ni a la sección **pública** de una clase. Se puede poner en cualquier parte de la declaración de la clase.

El definir una clase como **friend** da acceso a los datos privados en una única dirección. Así **Juani** es **friend** de **Pepi**, pero no al revés. La relación de amistad no es mutua a no ser que se declare como tal específicamente.



Representación de la relación de" amistad" definida en la clase Pepi: friend class Juani

Funciones friend

Es posible que una función que no es miembro de una clase tenga acceso a la parte privada de esa clase, declarándola como **friend** (amiga) de la clase. El prototipo estará definido dentro de la clase, pero con la palabra reservada **friend**. La definición de la función puede estar en cualquier sitio.

Las funciones **friend** no pueden ser **inline**.

El formato de la declaración de funciones **friend** es el siguiente:

Ejemplo

```
class ejemplo {
        <...>
public:
      friend tipo funcion (parametros); // Prototipo
 }
```

No es deseable abusar de las funciones amigas. Aunque técnicamente no son necesarias, la razón por la que se permiten en C++ este tipo de funciones es la de resolver situaciones en las que es necesario poder realizar operaciones con objetos de dos clases distintas. Véase el siguiente ejemplo:

```
// Programa: Funciones afines
// Fichero:  FUN-FR.CPP
#include <iostream.h>
#define SI 1
#define NO 0

enum Material { madera, cuero, aluminio, plastico, cristal};
enum Color    { negro, azul, marron, verde, rojo, blanco,
                transparente, dorado, plateado};
```

```
class Mesa;
class Silla {
   Color    color;
   Material material;
   int      patas;
public:
   Silla() {color=marron; material=madera; patas=4;}
   void PonColor(Color);
   void PonMaterial(Material);
   void PonPatas(int);
   friend int MismoColor(Silla, Mesa);
};

class Mesa {
   Color color;
   Material material;
   int patas;
public:
   Mesa() {color=transparente; material=cristal; patas=4;}
   void PonColor(Color);
   void PonMaterial(Material);
   void PonPatas(int);
   friend int MismoColor(Silla, Mesa);
};

// Funciones miembro de la clase Silla
void inline Silla::PonColor(Color c) { color=c; }
void inline Silla::PonMaterial(Material m) { material=m; }
void inline Silla::PonPatas(int num) { patas=num; }

// Funciones miembro de la clase Mesa
void inline Mesa::PonColor(Color c) { color=c; }
void inline Mesa::PonMaterial(Material m) { material=m; }
void inline Mesa::PonPatas(int num) { patas=num; }

// Función amiga a las dos clases
int MismoColor (Silla s, Mesa m) {
    if (s.color == m.color)
         return SI;
    else return NO;
}

void main (void) {
   Silla s_comedor;
   Mesa  m_salon, m_comedor;

   m_comedor.PonColor(marron);
   m_comedor.PonMaterial(madera);

   if (MismoColor(s_comedor, m_salon))
        cout << "\nLa silla y la mesa combinan\n";
   else cout << "\nLa silla y la mesa no combinan\n";
   if (MismoColor(s_comedor, m_comedor))
        cout << "\nLa silla y la mesa combinan\n";
   else cout << "\nLa silla y la mesa no combinan\n";
}
```

Notas de interés sobre el programa:

- La función **MismoColor** no es miembro de ninguna de las dos clases definidas, pero es **friend** a ambas. Proporciona un valor verdadero si el objeto **silla**, y el objeto **mesa** tienen el mismo **color**, y un valor nulo en caso contrario.
- Para realizar su cometido la función **MismoColor**, al ser amiga, tiene acceso a las partes privadas de ambas clases[4].
- Aparece una declaración vacía de la clase **Mesa** antes de la declaración de la clase **Silla.** Esto es así para que se pueda hacer la comprobación estática de tipos, ya que dentro de la clase **Silla** la función **MismoColor** hace referencia a **Mesa** antes de que ésta haya sido declarada, por lo que es preciso hacer una referencia anticipada a **Mesa**. Normalmente, las únicas ocasiones en que se necesitan hacer referencias anticipadas son aquellas en las que se ven envueltas funciones **friend**.

Comentarios sobre las funciones amigas

A) Las funciones amigas violan el encapsulamiento

Las funciones amigas pueden verse de tres formas diferentes:

- Las funciones amigas no son funciones miembro, y por su mera existencia están violando el encapsulamiento, al tener acceso a datos privados.
- Las funciones amigas están dentro de la barrera de encapsulamiento de la clase.
- Son una especie de puerta trasera que permite un acceso irregular a los datos privados de la clase.

En conclusión, las funciones amigas no son más que una variante sintáctica de las funciones métodos públicos. Usándolas de esta forma no se está violando el encapsulamiento más que lo haría un método público. Por lo tanto las funciones amigas y los métodos públicos constituyen la barrera del encapsulamiento, definida por la propia clase.

Hay programadores que abogan por la eliminación de este tipo de funciones, por considerarlas intrínsecamente peligrosas, pero siempre se podrá crear algún método para ofrecer detalles privados de una clase a otras clases o funciones del mundo externo, y que también estarán rompiendo el encapsulamiento.

B) Ventajas y desventajas

La principal **ventaja** de utilizar funciones amigas es la claridad sintáctica. Funciones miembro y funciones amigas tienen los mismos privilegios, pero mientras que una función miembro debe ser invocada de la forma **objeto.funcion()** una función amiga se llama de la forma simple **funcion()**.

Incluso si no se buscan razones de claridad sintáctica, las funciones amigas se pueden emplear para diseñar dos o más clases de forma que trabajen de una forma más compenetrada, pudiendo servir incluso para establecer un mecanismo de comunicación entre clases.

En cuanto a las **desventajas**, las funciones amigas tienen básicamente las siguientes:

- Se añaden al espacio de nombres global.
- No se heredan.
- No responden al polimorfismo. No existen funciones virtuales amigas (se verá esto en capítulos posteriores).

[4] Se podía haber hecho esto creando funciones **public** que proporcionasen los colores de ambos objetos, y entonces cualquier función podía realizar la comparación. Sin embargo, una aproximación como ésta requiere llamadas adicionales a funciones, lo que puede causar en algunos casos una ineficiencia excesiva.

C) Cuando se debe emplear una función miembro y cuando una amiga
La respuesta que se puede encontrar en la bibliografía puede ser del tipo: "*Utiliza una función miembro cuando puedas, y una función amiga cuando tengas que hacerlo*", lo cual nos deja un poco como antes de plantearnos la cuestión.

Debemos tener presente que, a semejanza de la vida real, las funciones amigas tendrán una serie de privilegios al igual que los miembros de nuestra clase, pero a otra serie de ventajas o beneficios no podrán acceder.

Así, los privilegios de acceso a los datos privados están garantizados tanto por un método como por una función amiga, sin que, como se ha comentado antes, se rompa el encapsulamiento. Se debe tener en cuenta la mayor claridad sintáctica que nos ofrecen las funciones amigas, o su empleo en la sobrecarga de operadores, como se verá en capítulos posteriores. Como contrapunto se debe sopesar que las funciones amigas no admiten ni herencia ni polimorfismo.

Matrices de objetos

Se pueden declarar matrices de objetos de la misma manera que se declaran de cualquier otro tipo de datos.

Ejemplo:
```
class Hormiga {
   int patas, cuerpo;
public:
   Hormiga (void)          {...}  // Constructor por defecto
   Hormiga (int, int)      {...}  // Constructor
   void BuscaComida(void)  {...}
};

void main (void) {
   Hormiga miembros[1000];       // Matriz de 1000 Hormigas
   for (register int i=200;i<501;) {
      miembros[i++].BuscaComida();
   }
   // Sigue el programa
}
```
Cuando se declara una matriz de objetos, el constructor se llama para cada elemento de la matriz. Si se quiere declarar matrices sin tener que iniciarlas, la clase debe tener un constructor por defecto, esto es uno que se pueda llamar sin argumentos.
Se puede iniciar cada elemento de la matriz de forma explícita llamando a un constructor con argumentos. Si no se inician todos los elementos de la matriz, el constructor por defecto se llama para los restantes elementos.

Ejemplo:

```
Hormiga colonia[10] = { Hormiga(4,1), Hormiga(6,1) };
```
Sólo se han iniciado explícitamente los elementos 0 y 1 de **colonia**. Para el resto se habrá utilizado el constructor por defecto para la clase **Hormiga**.

Si la clase tiene un constructor con un solo parámetro, se puede especificar el argumento como valor inicial de un elemento. Se pueden mezclar diferentes estilos de iniciación.

Ejemplo:

```
String msg[20] =  { "Primera línea del mensaje\n",
                    "Segunda línea.\n",
                    String ("Tercera línea.\n"),
                    String ( "", 25),
                    String ()
                  };
```

Relación entre las clases, las estructuras y las uniones

Clases y estructuras

Un tipo **struct** en **C++** tiene capacidades más amplias que su equivalente en **C**. En **C++** las clases y las estructuras están íntimamente relacionadas[5]. De hecho, con una única excepción, son intercambiables, ya que una estructura en **C++**[6] puede incluir datos y el código que manipula estos datos, de igual forma que puede hacerlo una clase. La única diferencia entre una estructura **C++** y una clase es que, por defecto, los miembros de una clase son privados, mientras que los miembros de una estructura son públicos. Salvo por este detalle, las estructuras y las clases desempeñan la misma función.

Ejemplo:

```
// Curso Lenguaje C++
// Programa: Relación entre clases y estructuras
// Fichero:  CL&ST.CPP
#include <iostream.h>

struct Prueba {   // Por defecto público
   Prueba() {i=10;}
   int Lee(void);
   void Escribe(int);
private:
   int i;
};

int Prueba::Lee(void) { return i; }
void Prueba::Escribe(int valor){ i=valor; }

void main (void) {
   Prueba P;
   cout << "\n" << P.Lee() << "\n";  // Muestra 10
   P.Escribe(-10);
   cout << "\n" << P.Lee() << "\n";  // Muestra -10
}
```

[5] Algunos autores consideran el concepto de clase como algo genérico y abstracto que se puede representar por unas estructuras de datos que son **struct** y **class**. Por lo que respecta al criterio seguido por nosotros, se está asociando el concepto de clase a la nueva estructura que nos ofrece **C++** que es la **class**. Pero no hay que olvidar que todo lo que se está haciendo con **class** se podía haber realizado con **struct**, salvando una pequeña diferencia de forma.

[6] Se recuerda que las estructuras en **C** no podían contener funciones, pero sí punteros a funciones.

Clases y uniones

En **C++** existe una relación entre las clases y las uniones. Para **C++** una unión es en esencia una estructura en la que todos sus elementos se almacenan en la misma posición.

Al igual que en las estructuras los miembros de la unión son públicos por defecto. De hecho no se pueden aplicar las palabras reservadas **private** y **protected** en una unión.

Como ejemplo se utiliza la unión para mostrar los caracteres que forman el byte menos significativo y el byte más significativos de un entero (se está suponiendo que los enteros son de dos bytes en la máquina que se está trabajando).

```
// Curso Lenguaje C++
// Programa: Relación entre clases y uniones
// Fichero:  CL&UN.CPP
#include <iostream.h>

union Prueba {  // Público
   int i;
   char bytes[2];  // Los dos atributos comparten las posiciones de memoria

   Prueba(int valor) { i=valor; }  // Constructor
   void Muestra(void) {
      cout << hex << int(bytes[0]) << "|";
      cout << hex << int(bytes[1]) << "\n";
   }
};

void main (void) {
   Prueba P(16);  // Muestra 10, el 16 en hexa
   P.Muestra();   // Muestra  0
}
```

Distribución del código fuente

Se ha llegado a un punto en el que las líneas de los programas van creciendo cada vez más en cada ejemplo. Por esto es un buen momento para explicar la forma más común de organizar los ficheros fuente en **C++**.

Esto que se va a explicar es una práctica común y muy extendida, pero no implica que sea obligatorio.

En **C++** se tienen tres tipos de ficheros:

Ficheros cabecera (*header*):

Contienen la declaración de la estructura de una y sólo una clase. También se incluyen las definiciones de las funciones miembro en línea (***inline***). Esto es debido a que el compilador debe tener acceso al código fuente de la función **inline** para poder insertar el código cada vez que la función sea llamada. Estos ficheros suelen tener la extensión **.h**, y a veces **.hh, .hpp** o **.hxx** para distinguirlas de los ficheros cabecera de **C.**

Ficheros cuerpo (*body*):

Éstos van a contener la definición de los métodos de una clase. La extensión más difundida para estos ficheros es **.cpp**. Pero sin embargo los compiladores derivados del **GNU C Compiler (*g++, DJGPP*)** utilizan la extensión **.cc** para referenciar a los fuentes de **C++**. Otra extensión suele ser **.cxx.**

Fichero principal (*main*):

Es el que alberga la función de entrada al programa. La extensión que se utiliza para estos es la misma que para los ficheros *body*.

Esta separación tiene como objetivo el que las clases puedan ser reutilizadas en otras aplicaciones de forma fácil y rápida. Por decirlo de alguna forma, el fichero cabecera describe la interfaz y el fichero cuerpo describe su realización.
Si otros programadores quieren utilizar la clase, sólo necesitan el fichero cabecera y el fichero objeto para enlazarlo con sus programas. No necesitan ver el cuerpo de la aplicación.

Otra ventaja es que si se realiza algún cambio en el cuerpo de la clase, se recompila ésta, y el resto del fuente no necesita cambiar nada.

Desafortunadamente, es necesario revelar algunos aspectos de la realización de la clase en el fichero cabecera.

- Los miembros privados son visibles en el fichero cabecera, aunque no sea accesibles.
- Como ya se ha indicado, las definiciones de las funciones miembro **inline** son también visibles.
- Si se cambia los miembros privados o las funciones **inline** de la clase, esos cambios se reflejan en el fichero cabecera, y todos los programadores que usen la clase deben recompilar su código con el nuevo fichero cabecera. Pero ellos no deben tocar su código fuente si la interfaz no ha cambiado, esto es, si no se han cambiado los prototipos de los métodos públicos.
- Un punto a tener en cuenta son los ficheros **#include** que se necesita tener en la cabecera del fichero fuente. La idea es procurar poner sólo los necesarios, e intentar no repetir. Una guía a seguir es que si poniéndolos en los ficheros cuerpo, y principal es suficiente, no colocarlos también en el fichero cabecera.

- En cuanto a la inclusión de ficheros **#include** de bibliotecas estándar del **C** se sigue el siguiente procedimiento:

```
extern "C" {
      #include <fichero1.h>
      #include <fichero2.h>
      ...
};
```

La razón de esto es que las funciones de **C** y de **C++** no son compiladas de igual forma (*ya que en el **C** no existe la sobrecarga, funciones dentro de estructuras, etc...*). Si dentro de un programa **C++** se compilaran las funciones de biblioteca **C** al estilo **C**, el programa de enlace (*linker*) no sería capaz de encontrarlas. Por este motivo los programas **C++** previenen al compilador de lo que se define a continuación es al estilo **C**.
La mayor parte de los compiladores consideran esta opción por defecto, y no es necesario indicarlo de forma explícita. Pero siempre es bueno conocerlo e incluso utilizarlo por evitar incompatibilidades entre distintos compiladores.

Como ejemplo, se ha reorganizado la clase **Fecha**, vista en este capítulo, en tres archivos: uno de cabecera, uno con el cuerpo de las definiciones y uno principal para demostrar su uso.

fecha.h

```
// Curso Lenguaje C++
// Programa: Declaración de la clase Fecha
// Fichero:  FECHA.H

#ifndef __FECHA_H__
#define __FECHA_H__

extern "C" {
     #include <conio.h>
};

#include <iostream.h>

class Fecha {
    int mes, dia, agno;
public:
    Fecha(int m, int d, int a); // Constructor
    int LeeDia(void) const;
    int LeeMes(void) const;
    int LeeAgno(void) const;
    void EscribeDia(int d);
    void EscribeMes(int m);
    void EscribeAgno(int a);
    void MuestraFecha(void) const;
};

inline void Fecha::MuestraFecha(void) const {
    clrscr();
    cout << dia << "/" << mes << "/" << agno << "\n";
}

#endif
```

fecha.cpp

```
// Curso Lenguaje C++
// Programa: Cuerpo de la clase Fecha
// Fichero:  FECHA.CPP

#include "fecha.h"

Fecha::Fecha(int d, int m, int a)
{ dia=d;  mes=m;  agno=a; }

int Fecha::LeeDia(void) const
{ return dia; }

int Fecha::LeeMes(void) const
{ return mes; }

int Fecha::LeeAgno(void) const
{ return agno; }

void Fecha::EscribeDia(int d)
{ dia=d; }

void Fecha::EscribeMes(int m)
{ mes=m; }

void Fecha::EscribeAgno(int a)
{ agno=a; }
```

demo.cpp

```
// Curso Lenguaje C++
// Programa: Objetos constantes. Usa la clase Fecha
// Fichero:  DEMO.CPP

#include "fecha.h"
//#include <iostream.h>       // En este caso no es necesario
                              // ponerlo porque ya está en el
                              // fichero cabecera fecha.h
void main(void) {
     const Fecha nacimiento(9,7,1971);

     nacimiento.MuestraFecha();
     cout << nacimiento.LeeMes();
}
```

Espacio de nombres (namespace)

Un espacio de nombres (**namespace**) es un mecanismo para expresar agrupamientos lógicos. Es decir, si algunas declaraciones se pueden agrupar lógicamente de acuerdo a algún criterio, se pueden colocar en un espacio de nombres común que exprese dicho hecho. La palabra reservada **namespace** define un espacio de nombres.

```
namespace nombre {
     cuerpo_espacio_de_nombres
}
```

Este formato crea un espacio de nombres con el cualificador **nombre**. El cuerpo_espacio_de_nombres puede incluir variables, definiciones de funciones y prototipos, estructuras, clases, enumeraciones (**enum**), definiciones de tipos (**typedef**) o bien otras definiciones de espacios de nombres. Hay que observar que tras la llave de cierre del espacio de nombres no es necesario colocar un punto y coma.

Las definiciones de espacio de nombres aparecen en archivos de cabecera y en módulos independientes con definiciones de funciones.

Por ejemplo,

```
namespace Rojo {
   int j;
   void imprimir (int i)
     { cout << "Rojo::imprimir() = " << i << endl; }
}

namespace Azul {
   int j;
   void imprimir (int i)
     { cout << "Azul::imprimir() = " << i+3 << endl; }
}

void sub1 () { ... }  // Puede acceder a espacio de nombres Azul y Rojo
void sub2 () { ... }  // Puede acceder a espacio de nombres Azul y Rojo
```

Los espacios de nombres **Rojo** y **Azul** tienen dos miembros con el mismo nombre: entero **j** y función **imprimir()**. Normalmente estas definiciones no podrían convivir en un espacio global, pero un espacio de nombres elimina ese problema.
Las definiciones de **sub1()** y **sub2()** tienen acceso a todos los miembros del espacio de nombres **Rojo** y **Azul**.

Extensiones

Los espacios de nombres son extensibles, es decir, se pueden añadir declaraciones posteriores a espacios de nombres definidos anteriormente. Las extensiones del espacio de nombres pueden aparecer también en archivos independientes de la definición original del espacio de nombres.

```
namespace Azul {  // definición original de espacio de nombres
   int j;
   void imprimir (int);
}

namespace Azul {  // extensión del espacio de nombres
   char car;
   char bufer[20];
}
```

Acceso a los miembros de un espacio de nombres

El operador de resolución de ámbito proporciona acceso a miembros de un espacio de nombres.

```
nombre_espacio_de_nombres::nombre_miembro
```

Por ejemplo, teniendo en cuenta la definición del espacio de nombres **Azul** realizada antes, la definición de la función **imprimir** fuera del espacio de nombres puede hacerse

```
void Azul::imprimir (int k) {
     cout << "Azul::imprimir() = " << k << endl; }
}
```

El operador de resolución de ámbito asocia **imprimir()** con el espacio de nombres **Azul**. Sin el mismo, el compilador define **imprimir()** como una función global.

Espacio de nombres sin nombre

En ocasiones, es muy útil envolver un conjunto de declaraciones en un espacio de nombres simplemente para proteger frente a la posibilidad de conflictos de nombres. El formato es:

```
namespace {
     cuerpo_espacio_de_nombres
}
```

Todos los miembros definidos en el **namespace** anterior están en un espacio de nombres sin nombre, que se garantiza único para cada unidad de traducción.

Directiva using

El acceso a los miembros de espacio de nombres puede tener dificultades, especialmente con calificadores de espacios de nombres largos y espacios de nombres anidados. La directiva **using** proporciona acceso a todos los miembros del espacio de nombres sin un cualificador de espacio de nombre y el operador de ámbito. El formato es:

```
using namespace nombre;
```

El cualificador **nombre** debe ser un nombre de espacio definido anteriormente. Puede aparecer la directiva **using** en ámbitos locales y globales.

Ejercicios

1. ¿Es correcta la siguiente declaración de clase? ¿Por qué?

```
class Fecha {
public:
   Fecha (int d, int m, int a);
   void Visualizar();
   ~Fecha();
private:
   int dia, mes, agno;
};
```

2. ¿Es correcta la siguiente declaración de clase? ¿Por qué?

```
class practica {
private:
   int a;
public:
   char x;
private:
   float m;
};
```

3. Se tiene la siguiente clase:

```
class Registro {
   char nombre[50];
   char telefono[15];
   register int i;
public:
   void Iniciar     (void);
   void EntradaDatos(void);
   void SalidaDatos (void);
};
```

¿Hay algún problema en su definición? En caso afirmativo señalarlo, decir la causa, y cómo solucionarlo. En caso de estar correcto señalar los miembros de datos y los métodos.

4. El siguiente programa tiene fallos. Encontrarlos, explicar cual se la causa, e indicar cómo se eliminaría el error.

```
#include <iostream.h>
class Circulo {
   int c_x, c_y;
   float radio;
public:
   void Circulo (int x, int y, float r) {
      c_x=x;
      c_y=y;
      radio=r;
   }
   void Visualizar(void) {
      cout << c_x << " " << c_y << " " << radio << "\n";
   }

   float Longitud(void) {
      return 3.14159*2*radio;
   }
};

void main (void) {
   Circulo c1(5, 4, 4);
   Circulo c2;
   Circulo c3(5);

   c1.Visualizar();
   c2.Visualizar();
   c3.Visualizar();

   cout << c1.Longitud << "\n";
}
```

5. ¿Cuál es la salida de este programa?

```
#include <iostream.h>
class P {
   int a, b, c;
public:
   void M(void) {
      cout << "\n" << a << " " << b << " " << c << "\n";
   }
};

void main(void) {
   P A;
   A.M();
}
```

Y si se añade un constructor de la forma: **P(){}**

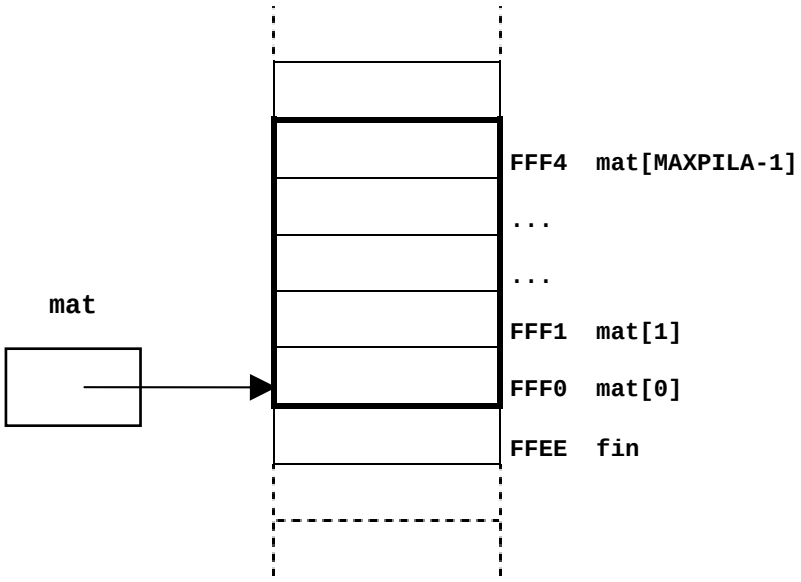
6. Crea un programa que tenga una clase que contenga una constante entera como dato miembro privado.
Define 4 instancias de esa clase, y haz que cada una de ellas tenga un valor distinto de la constante.

6.a ¿Qué método has utilizado para iniciar la constante?

6.b ¿Qué más métodos conoces para iniciar la constante?

7. Basándonos en una matriz estática, construir una clase que represente una pila de caracteres.
Pueden utilizarse las siguientes definiciones.

```
#define MAXPILA 5
int  cima;         // Marca la cima de la pila en cada momento.
char mat[MAXPILA]; // Lista de caracteres gestionada como pila.
```



Pila vacía: `cima = -1`
Pila llena: `cima = MAXPILA - 1`

La clase deberá disponer de los siguientes métodos:

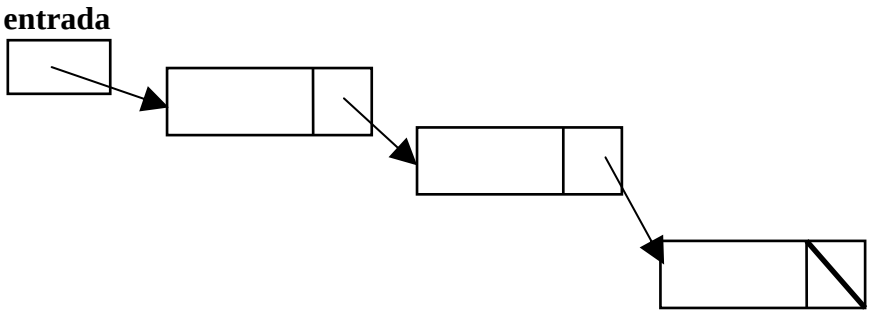
- mandatos para
  - iniciar pila
  - push
  - pop
  - mostrar contenido de toda la pila
- consultas para
  - comprobación de si la pila está vacía
  - comprobación de si la pila está llena

Construir, además, un pequeño programa principal que defina un objeto de la clase pila y comprueba el funcionamiento de todos los métodos de dicha clase.

8. Basándonos en una lista enlazada, construir una clase que represente una pila de caracteres.
Pueden utilizarse las siguientes definiciones.

```
struct nodo  {
   char dato;   /* Dato almacenado en el nodo */
   nodo *sig;   /* Puntero al siguiente nodo  */
};

nodo *entrada;           /* Puntero de entrada a la pila */
```



Se deberá disponer de los siguientes métodos:

- mandatos para
  - iniciar pila
  - push
  - pop
  - eliminar pila
  - mostrar contenido de toda la pila
- consultas para
  - comprobación de si la pila está vacía

Construir, además, un pequeño programa principal que defina un objeto de la clase pila y comprueba el funcionamiento de todos los métodos de dicha clase.

9. Modificar el ejercicio anterior incluyendo un constructor y un destructor para la clase pila definida anteriormente.
Se deberá disponer de los siguientes métodos:

- mandatos para
  - iniciar pila (constructor)
  - push
  - pop
  - eliminar pila (destructor)
  - mostrar contenido de toda la pila
- consultas para
  - comprobación de si la pila está vacía

Construir, además, un pequeño programa principal que defina un objeto de la clase pila y compruebe el comportamiento del constructor y del destructor de dicha clase.

10. Modificar de nuevo el ejercicio anterior de forma que se defina **nodo** no como una estructura sino como una clase, de forma que por composición, la *clase pila* incluya como miembro un objeto de la *clase nodo*.
Deberán definirse los métodos de acceso apropiados para la *clase nodo*, de forma que sus atributos puedan definirse privados.

Construir, además, un pequeño programa principal que defina un objeto de la clase pila y compruebe el comportamiento de dicha clase.

Realizar además una distribución del código fuente en fichero/s cabecera (*header*), fichero/s cuerpo (*body*) y el fichero principal (*main*).