TÉRMINOS Y CONCEPTOS DE POO

La P.O.O. es un paradigma de programación que se fundamenta en los conceptos de **objeto** y **clase**. En primer lugar, definamos que entendemos por objeto y clase:

1. ¿Qué Significa POO?

La filosofía de la POO (*Object Oriented Programming, Programación Orientada a Objetos)* da lugar a una nueva idea, *el objeto*.

1.1. **Objeto:** Una entidad autónoma con una funcionalidad concreta y bien definida.

El objeto es una abstracción en la que se unen sentencias y datos, de tal forma que a un objeto sólo lo van a poder tratar los métodos definidos para él, y estos métodos están preparados para trabajar específicamente con él. Este grado de compenetración evita que un método pueda tratar datos no apropiados, o bien que unos datos puedan ser tratados por un método no adecuado, ya que la llamada a cualquier método ha de ir siempre precedida del objeto sobre el que se quiere actuar, y éste sabe si ese método se ha definido o no para él. En terminología POO, cuando se quiere ejecutar un método sobre un objeto, se utiliza un mensaje que se envía al objeto, de tal forma que el objeto llame al método y éste sepa qué objeto lo ha llamado.

1.2. **Clase:** Es la especificación de las características de un conjunto de objetos.

Una clase es una definición de operaciones que se define una vez en algún punto del programa, pero normalmente se define en un archivo cabecera, asignándole un nombre que se podrá utilizar más tarde para definir objetos de dicha clase. Las clases se crean usando la palabra clave ***class***, y su sintaxis es la siguiente.

Con el ejemplo sólo conseguimos definir la clase, pero no se crea ningún objeto.

```
// Definición de la clase Persona

class Persona {

private:   // Declaración de variables privadas de la clase (Datos)
……

public:   // Declaración de los métodos de la clase y Prototipos
…….
};
```

Nota: *Muy importante no olvidar el punto y coma final.* Un objeto es una *instancia* de una clase.

El cuerpo de un método, puede o bien incluirse en la definición de una clase, en cuyo caso lo llamaremos método inline, o bien en un archivo fuera de la clase. Para que ***C++*** sepa cuándo una función es simplemente una función, y cuándo es un método, en éste último caso siempre debemos poner al método como prefijo el nombre de la clase a la que pertenece seguido de :: (operador de ámbito).

Por ejemplo, el método inline podría ser:

```
class Celdilla
{
        Private:
                char Caracter, Atributo; //Miembros privados

        public:

                void FijaCeldilla(char C, char A)
                {
                Caracter=C; Atributo=A;
                }
```

```
            void ObtenCeldilla(char &C, char &A)
            {
            C=Caracter; A=Atributo;
            }
            //&C es una referencia a la variable C
};
```

La otra forma sería:

```
class Celdilla
{

 private:

      char Caracter, Atributo;
  public:

      void FijaCeldilla(char C, char A);
      void ObtenCeldilla(char &C, char &A);

};

      void Celdilla::FijaCeldilla(char C, char A)
      {

      Caracter=C;Atributo=A;

      }

      void Celdilla::ObtenCeldilla(char &C, char &A)
      {

      C=Caracter;A=Atributo;

      }
```

2. Constructores y Destructores (Inicialización de Clases I)

Son métodos que permiten establecer el estado inicial y final de un objeto. Los constructores se pueden definir con un conjunto de argumentos arbitrario, pero no pueden devolver nada. Y los destructores no pueden recibir ni devolver ningún valor.

El constructor debe llamarse igual que la clase, y el destructor el nombre de la clase precedido del carácter ~

```
class Persona {

  private:
        …….
  public:

  // Constructor de objetos Alumno

  Persona (variables privadas del objeto …….)

  // El destructor sería: ~Persona (void);
  …….

};
```

2.1. Creación y Destrucción de Objetos

Debido a que una clase es únicamente una **especificación, p**ara poder utilizar la funcionalidad contenida en la misma, se deben **instanciar** las clases.

Un constructor se ejecuta cuando se crea un nuevo objeto: 1) por declaración, ó 2) cuando se crea dinámicamente con el operador *new*.

Cuando un objeto deja de ser útil hay que eliminarlo. De esta manera la aplicación recupera los recursos (memoria) que ese objeto había acaparado cuando se creó.

La destrucción de objetos creados en tiempo de ejecución con *new* se realiza mediante el operador ***delete***.

Un destructor se ejecuta cuando el objeto deja de existir: 1) porque su ámbito acaba, ó 2) cuando se libera explícitamente con el operador ***delete***.

2.1.1. Creación por Declaración.

Un objeto se puede instanciar de una forma simple, declarando una variable del tipo de la clase.

```
Persona Objetopepe;
```

2.1.2. Creación Dinámica

Es la forma habitual de crear objetos en *C++* y se realiza mediante el operador ***new***.

```
Persona * Objetopepe; // Variable Global.

        // Objetopepe es un puntero a objetos de tipo Alumno

Objetopepe = new Persona;
```

Luego, para destruir el objeto:

```
delete Objetopepe;
```

2.2. Miembros Públicos, Protegidos y Privados

Un miembro *público* es accesible en cualquier lugar en el que exista un objeto de la clase. Un miembro *protegido* sólo es accesible desde las clases que se hereden de la clase que lo contiene.

Un miembro *privado* sólo es accesible por los métodos de la clase a la que pertenece. También se puede crear una clase usando la palabra clave **struct**, la diferencia es que con **struct** los miembros se hacen públicos por defecto, mientras que con **class** se hacen privados por defecto.

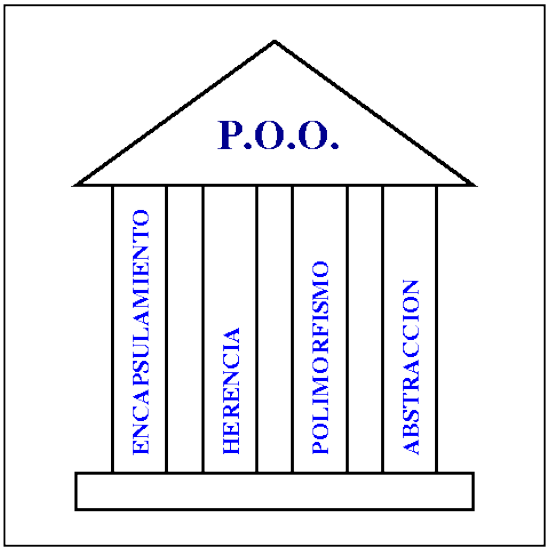
Para hacer un miembro *público, privado o protegido* usamos respectivamente:

public, **private** o **protected**

3. El Paradigma de la POO en C++

Existen cuatro principios básicos que cualquier sistema orientado a objetos debe incorporar, que se esquematizan en la figura 1.

Figura 1. Pilares de la POO.



3.1 Encapsulación

Este concepto permite tener un control de acceso selectivo tanto a los miembros como a los métodos, de tal forma que desde fuera del objeto sólo se pueda acceder a los métodos e identificadores que permita el creador del objeto.

En la programación clásica (lenguaje **C**, p.e.) existen datos y procedimientos que actúan sobre esos datos. No hay una relación aparente entre datos y procedimientos/funciones y esta relación se establece de manera más o menos precisa de acuerdo a la profesionalidad del programador.

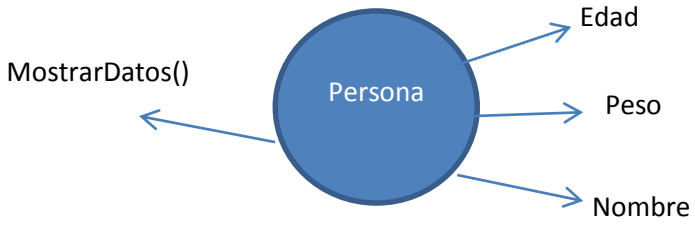
En un objeto podemos distinguir dos aspectos bien diferenciados:

- Estado ----------> **Propiedades**
- Comportamiento ---> **Métodos**

En P.O.O. los datos y los procedimientos que los gestionan están relacionados explícitamente y se "encapsulan" en un objeto. La especificación de las propiedades de un objeto y los métodos de acceso se realiza en la declaración de la clase de la que se instancia el objeto.

En la figura 2 esquematizamos las propiedades y métodos que se van a asociar a los objetos de la clase **Persona**:

Figura 2: Propiedades y métodos de los objetos de la clase **Persona**



La declaración de propiedades y métodos de los objetos de la clase **Persona** se realiza de la siguiente manera (en un fichero de cabecera .h) tal como se muestra en el siguiente código ejemplo que ilustramos a continuación:

En el fichero *.h*:

```
//-----------------------------------------------

class Persona {

 private:
   int      edad;       // Propiedades
   int      peso;
   string    Nombre;

 public:

 void MostrarDatos (void); // Métodos

};
```

3.2. Acceso a Miembros de un Objeto

Para acceder a los miembros de un objeto se usan los operadores típicos de acceso a miembros: el operador **.** para referencia directa al objeto y el operador -> para acceso a través de un puntero. Si creamos los objetos con *new*, y los referenciamos mediante un puntero, el operador de acceso que se utilizará es el operador ->

En el fichero *.cpp*:

```
//-----------------------------------------------
   ...

   int ValorY;
   ...
   Objetopepe->edad = 25;
   ValorY = Objetopepe->peso;
   Objetopepe->MostrarDatos();   //Equivalente a
(*Objetopepe).MostrarDatos();
}
```

3.3. Abstracción

Es la ocultación de detalles irrelevantes o que no se desean mostrar. Podemos distinguir en una clase dos aspectos desde el punto de vista de la abstracción:

- Interfaz: lo que se puede ver/usar externamente de un objeto.
- Implementación: cómo lleva a cabo su cometido.

Resumiendo: ***nos interesa saber qué nos ofrece un objeto, pero no cómo lo lleva a cabo.***

3.4. Herencia

La herencia es una de las características fundamentales de la *Programación Orientada a Objetos* por la que, tomando como base una clase ya existente, es posible derivar una nueva, que heredará todos sus miembros, teniendo posibilidad de sobrecargarlos, crear unos nuevos o utilizarlos. La idea básica por tanto, es reutilizar las clases existentes y extender o modificar su semántica. De esta forma, se permite la reutilización y la extensión del código. De esta forma, se permite diseñar nuevas clases a partir de otras ya existentes, pudiéndose además extender sus métodos (cambiar su semántica en la clase que hereda).

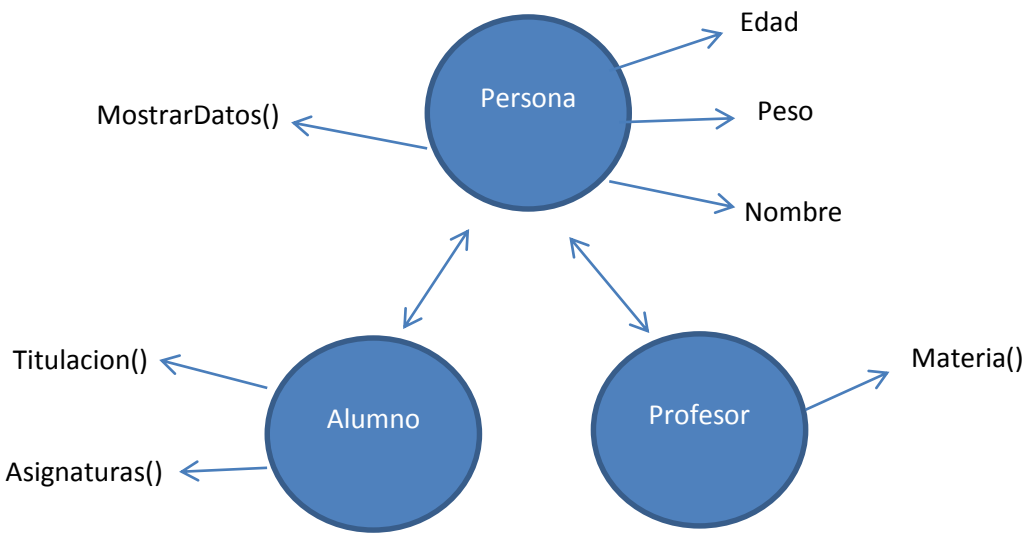
Cuando una clase hereda de otra, la **clase derivada** incorpora todos los miembros de la **clase base** además de los suyos propios.

La herencia es una herramienta muy importante en muchos aspectos del desarrollo de aplicaciones:

- Organización del diseño.
- Reusabilidad de clases (propias o no).
- Mejora del mantenimiento.

Tomando como base la clase **Persona** se van a construir dos nuevas clases, **Alumno** y **Profesor**, que *derivan* de **Persona**. Esto significa que los objetos de estas clases tienen asociados las propiedades y métodos de la clase base, **Persona**, además de los suyos propios. En la figura 3 esquematizamos el mecanismo de herencia para las nuevas clases y las nuevas propiedades que se asocian a los objetos de las clases derivadas.

Fig 3. Las clases **Alumno** y **Profesor** heredan las propiedades y métodos de la clase base



En el fichero *..h*:

```
//************************************************/
// Definicion de la clase derivada Alumno
// Deriva de la clase base Persona
//************************************************/

class Alumno : public Persona {

public:

  Titulacion();  // Propiedad exclusiva de Alumno
  Asignatura();  // Idem

};
```

```
//************************************************/
// Definicion de la clase derivada Profesor.
// Deriva de la clase base Persona
//************************************************/

class Profesor : public Persona {

public:

  Materia(); // Propiedad exclusiva de Profesor

};
```

Antes del nombre de la clase base hay que poner *public*, esto es así porque C++ permite también la herencia *private*.

3.5. Clases Abstractas

Clase abstracta: es una clase que no está completamente especificada (posee métodos sin implementar), por lo tanto ***no se pueden crear instancias de la misma***. Una clase abstracta se usa para servir de clase base a otras clases. En terminología C++ se dice que una clase abstracta es aquella que posee al menos un **método virtual puro**.

- **Virtual:** obliga a las clases derivadas a implementar ese método.
- **Puro:** no pueden crearse instancias de esa clase.

En el fichero *.h*:

```
class Persona {

public:
  ...

  // Otros metodos

  virtual void MostrarOtros (void) = 0; // Metodo virtual puro
};
```

```
class Alumno : public Persona {

public:
  ...

  // Instanciacion del metodo virtual puro de la clase Persona

  void MostrarOtros (void);

};



class Profesor : public Alumno {

public:
  ...

  // Instanciacion del metodo virtual puro de la clase Persona

  void MostrarOtros (void);

};
```

En el fichero *.cpp*: hay que implementar el método

¿Por qué se especifica el método MostrarOtros en Persona, como virtual puro, en lugar de omitirlo? Fundamentalmente podemos considerar dos razones para usar métodos virtuales puros:

- Para obligar a que las clases descendientes los implementen. De esta forma estamos seguros de que todas las clases descendientes no abstractas de Persona poseen el método, y se podrá invocar con seguridad.
- Para evitar que se puedan crear instancias de la clase abstracta.

3.6. Herencia Pública o Privada

¿Qué estado tendrán en una clase derivada los miembros y métodos heredados de su clase base?. Todo va a depender del cualificador que se utilice delante del nombre de la clase base: **public** o **private**. Los miembros de la clase base que sean private no serán accesibles en la clase derivada en ningún momento.

Los miembros de la clase base que sean declarados **protected** serán accesibles en la clase derivada, aunque no fuera de ella, es decir, que si una clase deriva de forma privada miembros protegidos de la base, en la derivada van a ser accesibles, pero en una futura derivada de la actual derivada van a ser privados. Si por el contrario, se derivan como **public**, en la clase derivada van a seguir siendo públicos, pero si se deriva de forma privada, serán privados. Tal como mostramos a continuación:

BASE		DERIVADA public		DERIVADA private
Miembro	*private*	*private*	Pasa a	*private*
	protected	*protected*		*private*
	public	*public*		*private*

A continuación, mostramos un ejemplo de una clase derivada a una clase base en el código siguiente:

```
#include <iostream>
class Base
{
                    int Entero;     // Miembro privado por defecto
      protected:
                    void FijaEntero( int N)
                    {
                          Entero=N;
                    }
      public:
                    void ObtenEntero(void)
                    {
                          return Entero;
                    }
};
class Derivada : public Base
{
      public:
                    void ImprimeEntero(void)
                    {
                          cout<<Entero;  // Error: Entero no está
                                   // accesible aquí
                    }
                    void ActivaBase(int N)
                    {
                          FijaEntero(N);  // Correcto: acceso a
                                    // miembro protegido
                    }
                    void MuestraEntero(void)
                    {
                          cout << ObtenEntero();
                          // Correcto: acceso a miembro público
                    }
};


void main(void)
{
      Derivada A;
      A.ActivaBase(5);          // Se accede a un miembro público
      cout << A.ObtenEntero();  // Correcto, al ser un miembro
                                //  público
      A.FijaEntero(10);         //Error, FijaEntero está protegido,
}                               //y por tanto, no accesible desde
                                //fuera
```

3.7. Problemas de Accesibilidad

Vamos a ver lo que ocurre cuando tenemos una clase derivada de otra, en la cual (en la derivada), hay miembros que coinciden en nombre con miembros de la clase base. Como regla general, una clase siempre utilizará por defecto sus miembros antes que cualquier otro que se llame igual, aunque esté en la clase de donde se ha derivado. Para hacer uso de la clase base es necesario usar el operador **::** precedido del nombre de dicha clase.

En el siguiente código se muestra un ejemplo de la complejidad en la accesibilidad.

```
#include <iostream.h>
int Entero; // Variable global

class Base
{
      protected:
                  int Entero;  // Miembro protegido
};
class Derivada : public Base
{
                  int Entero;
      public:
                  void ImprimeValores(void);
};
void Derivada::ImprimeValores(void)
{
      cout << Entero;  // Imprime el Entero de la clase derivada
      cout << Base::Entero; // Imprime el Entero de la clase Base
      cout << ::Entero;  // Imprime el Entero varible global
}
```

3.8. Conversión de la Clase Derivada a Base

Un objeto de una clase derivada tendrá más información que uno de la clase base. Así, esto nos va a permitir hacer asignaciones de un objeto de la clase derivada a otro de la clase base, aunque no al revés, es decir, BASE=DERIVADA, pero no al revés. En el siguiente código mostramos un ejemplo de algunos problemas que pueden surgir con la conversión de una clase derivada a base.

```
class Base
{
            int Entero1;
};
class Derivada : public Base
{
            int Entero2;  // Se hereda Entero1
};
void main(void)
{
      Base        A;
      Derivada    B;

      A = B;  // Ningún problema, Entero1 se asigna de B a A
      B = A; // Error, en A existe Entero1 para asignar a B, pero
             // no Entero2
}
```

En este ejemplo, el objeto A está compuesto de un miembro -un entero-, mientras que B está compuesto de dos miembros -el entero heredado y el suyo propio-. Por tanto, B tiene información suficiente para "llenar" el objeto A, pero no así en sentido contrario.

3.9. Herencia de Constructores y Destructores

Los constructores y destructores de una clase **no** son heredadas automáticamente por sus descendientes. Deberemos crear en las clases hijas sus propios constructores y destructores. Es posible, no obstante, emplear los constructores de la clase base pero hay que indicarlo **explícitamente**. De ser así, es necesario saber:

- que los constructores y destructores de las clases base son invocados automáticamente **antes** que los constructores de las clases derivadas, y
- que los destructores de las clases derivadas se invocan **antes** que los de las clases base.

Por ello, Si se tiene una clase derivada se comprobará que parte de los miembros pertenecen a la *clase base*, y será ella quien los cree, y que parte pertenecen a *la derivada*. Por tanto, en la creación de una clase derivada intervendrán el constructor de la clase base y el de la derivada. Por tanto, el constructor de la clase derivada, normalmente, llamará al de la clase base. En el caso de que el constructor de la clase base no tenga parámetros, no va haber problemas, puesto que se llamará de forma automática, pero si por el contrario, tiene parámetros, la clase derivada tendrá que llamar al constructor de la clase base. Por tanto, el constructor de la clase derivada deberá recibir dos conjuntos de parámetros, el suyo y el de la clase base, al cual deberá de llamar con el conjunto de parámetros correspondiente.

El siguiente código muestra un ejemplo de herencia en constructoras.

```
#include <iostream.h>
class Base
{
                    int Entero1;
      public:
                    Base()
                    {
                          Entero1=0;
                          cout <<("Constructor Base \n");
                    }
                    Base(int N)
                    {
                          Entero1=N;
                    }
};
class Derivada : public Base
{
                    // Se hereda Entero1
                    int Entero2;
      public:
                    Derivada()
                    {
                          Entero2=0;
                          cout <<("Constructor Derivado\n");
                    }
                    Derivada(int N1, int N2) : Base (N1)
                    {
                          Entero2=N2;
                    }
};
void main(void)
{
      Base A(5);
      Derivada B(3,6);
}
```

En cambio, con los destructores ocurre todo lo contrario, puesto que se destruirá primero el objeto de la clase heredada y después se irán llamando a los destructores de objetos de las clases bases. Además, una ventaja que tiene es que como los destructores no tienen parámetros, no es necesario llamar al **destructor** de la clase base, sino que la llamada se hará de forma automática. En el siguiente código mostramos un ejemplo de herencia en destructoras.

```
#include <iostream>
class Base
{
      protected:
                  int Entero1;
      public:
                  Base()
                  {
                        Entero1=0;
                        cout <<("Constructor Base\n");
                  }
                  Base(int N)
                  {
                        Entero1=N;
                  }
                  ~Base()
                  {
                        cout <<("Destructor Base\n");
                  }
};
class Derivada : public Base
{
                  int Entero2;  // Se hereda Entero1
      public:
                  Derivada()
                  {
                        Entero2=0;
                        cout <<("Constructor Derivado\n");
                  }
                  Derivada(int N1, int N2) : Base(N1)
                  {
                        Entero2=N2;
                  }
                  ~Derivada()
                  {
                        cout << "Destructor Derivado\n";
                  }
}
void main(void)
{
      Base A(5);
      Derivada B(3,6);
}
```

3.10. Herencia Múltiple

La herencia múltiple es el hecho de que una clase derivada se genere a partir de varias clases base. Una misma clase puede heredar de varias clases base, e incorporar por tanto los distintos miembros de éstas, uniéndolos todos en una sola clase.

Para construir una clase derivada tomando varias como base, se pondrán los nombres de cada una de las clases base separadas por coma,

antecedidos, cada una de la palabra **public** o **private**. Siempre que sea posible evite usar herencia múltiple.

Ejemplo:

En un concesionario de coches podríamos considerar la siguiente jerarquía de clases:

```
class TProducto
 {
   long Precio;
   ...
};

class TVehiculo
 {
   int NumRuedas;
   ...
};

class TCocheEnVenta : public TProducto, public TVehiculo
{
   ...
};
```

Observar que los objetos de la clase TCocheEnVenta derivan de las clases TProducto y TVehiculo.

Es importante considerar, que existen dos formas para que una clase saque partido de las ventajas de otra, una es la herencia, y la otra es que una clase contenga un objeto de la otra clase. Ninguna de las dos posibilidades es mejor que la otra, en cada caso particular habrá que estudiar cual es la mejor opción.

Por ejemplo, si quisiéramos diseñar una clase (*TUniversidad*) que represente un marco (representado por un Alumno y un Profesor), podemos decidir distintas estrategias a la hora de llevarlo a cabo:

- Que herede de la clase Alumno y la clase Profesor.
- Que herede de la clase Persona y contenga un objeto de la clase Alumno y otro de la clase Profesor.
- Que herede de la clase Alumno y contenga un objeto de la clase Profesor.
- Que herede de la clase Profesor y contenga un objeto de la clase Alumno.

3.11. Restricciones de acceso en C++

En C++ se puede especificar el acceso a los miembros de una clase utilizando los siguientes especificadores de acceso:

- **public**: Interfaz de la clase.
- **private**: Implementación de la clase.
- **protected**: Implementación de la familia.

Estos especificadores no modifican ni la forma de acceso ni el comportamiento, únicamente controlan desde dónde se pueden usar los miembros de la clase:

- **public**: desde cualquier sitio.
- **private**: desde los métodos de la clase.
- **protected**: desde los métodos de la clase y desde los métodos de las clases derivadas.

Ejemplo:

```
//*************************************************/
// Definicion de la clase base TObjGraf
//*************************************************/

class TObjGraf {

private:    // Puede acceder SOLO los objetos de esta clase.
```

```
  int X;
  int Y;

protected:  // Pueden acceder los objetos de esta clase y sus
descendientes.

  TColor      Color;
  TPaintBox * PaintBox;

public:     // Pueden usarlas todas.

  // Constructor de objetos TObjGraf

  TObjGraf (TPaintBox *_PaintBox, TColor _Color=clBlack,
          int _X=0, int _Y=0);

  // Otros metodos

  virtual void Mostrar (void) = 0; // Metodo virtual puro

};
```

Es una buena técnica de programación no permitir el acceso público a las propiedades de un objeto, ya que si esto ocurriera podría peligrar la integridad del objeto. ¿Entonces cómo se puede cambiar el estado de un objeto desde el exterior?

- Ofreciendo métodos (públicos) que se encarguen de modificar las propiedades (privadas) que se desee. De esta manera son los métodos los que acceden a las propiedades y el usuario de la clase sólo accede a través de ellos. Esta es la técnica clásica que se emplea en C++
- A través de los métodos y de las propiedades "virtuales".

3.12. SOBRECARGA

Consiste en la redefinición de un método. Por tanto, un *método sobrecargado* es un método que consta de varias definiciones distintas, aunque todas ellas llamadas de igual forma. Lo que va a permitir distinguir un método de otro que se llame de igual forma van a ser el tipo y el número de parámetros que recibe.

```
#include <iostream.h>

class Celdilla

{

  char Caracter, Atributo;

  public:

        void FijaCeldilla(char C, char A)
        {
        Caracter=C;
        Atributo=A;

        }
        void FijaCeldilla(unsigned CA)
        {
        Caracter=CA & 0xff;
        Atributo=CA >> 8;
        }
        void ObtenCeldilla(char &C, char &A)
        {
        C=Caracter;
        A=Atributo;
        }
        unsigned ObtenCeldilla()
        {
        return Atributo<<8 | Caracter;
        }
};

void main()
{

    Celdilla      X, Y;
   unsigned         Casilla;
     char          C, A;
```

```
X.FijaCeldilla('A',112);          // Fija los valores del objeto X

Casilla = X.ObtenCeldilla();      //Obtiene el valor de X en forma
                                  // de un entero
Y.FijaCeldilla(Casilla);          //Toma Y en dos caracteres
cout << "Caracter= " << C << ", Atributo= " << (int) A;
}
```

3.13. POLIMORFISMO

Esta característica de C++ permite que un objeto tome distintas formas, gracias al enlace en tiempo de ejecución (vinculación dinámica). Una clase se puede comportar como cualquiera de sus antecesoras (en la asignación por ejemplo). Como tenemos variables (punteros) que pueden contener objetos de distintas clases, el compilador no sabe qué tipo de objeto es al que realmente apunta la variable (en tiempo de compilación) por lo tanto hay retrasar el enlace a tiempo de ejecución.

El **enlace dinámico** es retrasar el enlace de una llamada a un método (función) al tiempo de ejecución

3.13.1. Funciones Virtuales

Estas funciones nos van a ayudar a usar un método de una clase o de otra, llamado igual en las dos clases, dependiendo de la clase a la que estemos refiriéndonos. Para ello existe la palabra reservada **virtual**, que se pone en la clase base y se antepone al prototipo. A partir de este momento, todas las funciones llamadas igual con los mismos parámetros en las clases derivadas serán virtuales, sin que sea necesario indicarlo. Esto hará necesario que en el momento de ejecución se distinga cual es la función que hay que llamar.

Esto permite que un objeto unas veces tome una forma y otras veces tome otra. Para ilustrar el polimorfismo, crearemos una nueva clase, **Coche**, que derive de la clase **Alumno**. Un Coche (un objeto de tipo Coche,

para ser más precisos) es un auto que tiene la capacidad de movimiento. Para implementar el movimiento de un auto necesitamos incorporar nuevas propiedades y métodos propios a la clase Coche. Sin embargo, en este momento nos interesa poner de manifiesto el polimorfismo, lo que conseguimos a través del método **MostrarDatos()** asociado a la clase Coche. Antes, modificamos la declaración del método **MostrarDatos()** de la clase Alumno para obligar a sus descendientes a implementar su propio método **MostrarDatos()**: basta con indicar que en el método **MostrarDatos()** de la clase Alumno es virtual.

En un fichero de cabecera *.h*:

```
//************************************************/
// Definicion de la clase derivada Alumno.
// Deriva de la clase base Persona
//************************************************/
class Alumno : public Persona {
private:
  ...
public:
  ...
  // Ahora, el metodo MostrarDatos() se declara virtual, aunque no es
puro:
  // 1) Por ser virtual: cualquier clase que derive de Alumno debe
  //    tener su propio metodo MostrarDatos(),
  // 2) Por no ser puro: puede llamarse a este metodo con objetos
Alumno.

  virtual void MostrarDatos (void);
  ...
};
```

Ahora nos centramos en la nueva clase **Coche**. Antes, por comodidad y claridad, definimos un tipo por enumeración para la dirección del movimiento:

```
// Tipo definido por enumeracion para la direccion de Coche. Codificacion:
/*

        N
        2
        |
  O 4 ---  *  --- 3 E
        |
        1
        S
*/

enum TDireccion {S=1, N=2, E=3, O=4};
```

La declaración de la clase **Coche** se hará :

```
//************************************************/
// Definicion de la clase derivada Coche
// Coche deriva de la clase Alumno, que a su
// vez deriva de la clase base Persona
//************************************************/

class Coche: public Alumno{

private:

  int FDirX;     // Dir. en el eje X
  int FDirY;     // Dir. en el eje Y
  int FVelocidad; // Velocidad del movimiento

  void       SetDireccion (TDireccion _Direccion);
  TDireccion  GetDireccion (void);


public:

  // Constructores
```

```
  Coche (……,    // declaraciones de la clase base
        TDireccion _Direccion=E, int _Velocidad=5);

  // Otros metodos

  void MostrarDatos (void);
  void Borrar  (void);
  void Mover   (void);

};
```

Finalmente, para ilustrar el polimorfismo nos basaremos en la existencia de diferentes métodos con el mismo nombre: MostrarDatos(), que provocan diferentes acciones dependiendo del tipo de objetos al que se aplican. Vamos a crear dinámicamente cuatro objetos de diferentes clases. Estos objetos van a ser referenciados (mediante punteros) desde un vector de objetos de tipo Persona *. El polimorfismo se va a manifestar invocando a la función MostrarDatos() para cada uno de esos objetos.

3.14. AMISTAD

Esta característica permite que una función que no pertenezca a una clase pueda hacer uso de todos sus miembros.

3.14.1. Funciones Amigas

Para hacer que una función sea amiga de una clase y tenga acceso a todos sus miembros, habrá que definir un prototipo de esa función dentro de la clase, precediendo a la definición con la palabra **friend**. Ejemplo:

```
class  Clase
{
      int EnteroPrivado;

      public:
            void FijaEntero(int N);
            friend void FuncionAmiga(Clase &X,int N);
};
void FuncionAmiga(Clase &X,int N)
{
      X.EnteroPrivado=N;//Acceso al miembro privado
}
// Si no fuese friend EnteroPrivado no estaría accesible desde
// fuera de la clase
```

Es necesario, como se ve en el ejemplo, que a la función amiga se le pase como parámetro un objeto de la clase de la cual es amiga.

3.14.2. Clases Amigas

De forma análoga, puede ocurrir que sea necesario que todos los miembros de una clase tengan que hacer uso de los miembros de otra. Se podrían declarar como funciones amigas todos los métodos de la primera. Sin embargo, es más fácil declarar como amiga la primera clase. Para ello es necesario que antes de declarar la clase de la cual es amiga la otra, se ponga como aviso de que hay una clase amiga. Ejemplo:

```
class ClaseAmiga;
class Clase
{
      int EnteroPrivado;
      friend ClaseAmiga;
};
```

3.15. PLANTILLAS

Un problema con el que nos podemos encontrar es que tengamos dos clases que hacen uso de los mismos métodos pero estos están aplicados a distintos tipos de datos; Por ejemplo, listas con enteros, listas con cadenas, etc...

Para solucionar esto hay varias opciones:

a.- *Que la clase almacene todos sus datos como punteros a* **void** *realizando moldeados (castings) en las operaciones.*

b.- *Que todos los objetos que vayan a usar esa clase se creen a partir de una misma base.*

c.- *Usar plantillas.*

3.15.1. Plantillas de Funciones

Con esto lo que vamos a hacer es permitir al compilador ir creando las funciones apropiadas dependiendo de los parámetros utilizados.

La sintaxis es:

```
template <class X,class Y,.....>
tipo_resultado función(parámetros)
.
.
```

Con la palabra template estamos indicando al compilador que estamos creando una función general, (una plantilla) a partir de la cual deberá posteriormente ir creando las funciones adecuadas. Los identificadores X e Y, representan dos clases con las que al usar la función se debe instanciar la plantilla, y pueden usarse para indicar el tipo del valor devuelto, el tipo de los parámetros que recibe la función o para declarar variables dentro de ella.

3.15.2. Plantillas de Clases

Lo que diferencia una plantilla de clase de una plantilla de función es que en lugar de tener la definición de una función detrás de **template** y los argumentos, vamos a tener la definición de una clase. A la hora de implementar los métodos de la clase es obligatorio encabezar cada método de la misma forma que la clase, con la palabra **template** y los parámetros adecuados.

3.16. OTROS CONCEPTOS

3.16.1. Miembros Static

Los miembros y métodos de una clase pueden declararse estáticos anteponiendo la palabra static. Un dato declarado estático en una clase es un dato de una sóla instancia, se crea al definir la clase y sólo existirá uno, independientemente del número de objetos que se creen. Esto es bueno cuando todos los objetos van a compartir una misma variable. Para inicializar la variable, lo que haremos será preceder a la variable del nombre de la clase.

En cuanto a los métodos declarados como estáticos, debemos decir que no han de estar relacionados con objeto alguno de dicha clase. Un método estático no puede ser nunca virtual.

```
class Clase
{
      static int Dato;
      .....
};

int Clase::Dato=ValorInicial;      //Inicialización de la
                                   // variable estática
```

3.16.2. El modificador const

Un miembro declarado con const, no puede sufrir modificaciones, tan sólo una inicialización en el momento en que se crea. También un miembro de una clase puede declararse constante, impidiendo que cualquier otro método que no sea el constructor y el destructor lo trate. Ni siquiera el constructor puede asignar un valor al campo directamente, sino que tendrá que llamar al constructor del miembro que se va a inicializar. Sin embargo, si queremos que algunos métodos puedan hacer uso de estos miembros, lo que hacemos es poner detrás el modificador const. Ejemplo:

```
#include <iostream.h>
class Clase
{                   const int Objeto;   // Un miembro const privado
      public:
                    Clase(int N=0) : Objeto(N){ }// Inicializa el
                                                 // objeto
              // El método Imprime es capaz de manejar objeto const
                    void Imprime(void)
                    const { const cout << Objeto;}
};
void main(void)
{     const Clase X(15);   // Un objeto const de valor inicial 15
      X.Imprime();
}
```

3.16.3. El Objeto this

Permite acceder al objeto actual cuyo comportamiento se está definiendo. Es un apuntador a una clase usado dentro de cualquier método de dicha clase, por ejemplo:

this->Imprime().

3.16.4. Problemas de Ámbito

Es bastante normal crear dentro de una clase tipos enumerados o estructuras. En el caso de los tipos enumerados podemos acceder a sus elementos por medio del nombre de la clase y el operador de resolución de ámbito. Aunque también por medio de un objeto perteneciente a la clase se puede hacer, esta notación no se suele usar. En cambio, en el caso de las estructuras el acceso a los datos siempre hay que hacerlo a través de un objeto de la clase. Si tenemos privados los elementos de una enumeración entonces sólo van a poder ser accedidos desde el interior de la clase. En el siguiente ejemplo, mostramos una clase con problemas de ámbito.

```
class Clase
{                   enum Enumeracion1 {Negro, Blanco};
      public:
                    enum Enumeracion2 {Abierto, Cerrado};

                    struct {
                                char  Caracter;
                                int     Numero;
                           } Datos;
};
void main(void)
{     int         N;
      Clase Objeto;

      N=Clase::Abierto;  // Dos formas de acceder a lo mismo
      N=Objeto.Abierto;
      N=Objeto.Datos.Caracter;
}
```

3.17. CREACIÓN DE BIBLIOTECAS CON CLASES

3.17.1. Archivos de Cabecera

Normalmente, las definiciones de las clases están incluidas en ficheros cabecera, existiendo un archivo de cabecera por cada clase que definamos. En este archivo incluiremos la definición de la clase, cualquier constante, enumeración, etc..., que se pueda usar con un objeto de la clase y los métodos inline, ya que estos necesitan estar definidos antes de usarse. Mostramos un ejemplo de una macro:

```
#define Une(x,y)
                    // Macro para crear clase de manejo de matrices
#define ClaseMatriz(tipo)

class Une(Matriz, tipo)
{                   tipo *Dato;
      public:
                    Une(Matriz, tipo)(int Nelementos)
                    {
                          Dato=new tipo[NElementos];
                    }
                    Une(~Matriz, tipo)()
                    {
                          delete Dato;
                    }
                    tipo & operator[](int Elemento)
                    {
                          return Dato[Elemento];
                    }
};

ClaseMatriz(int); // Se crea una clase para manejar matrices int
ClaseMatriz(double); // Se crea una clase para manejar matrices
                     // double

#include <iostream.h>
void main(void)
{     Matrizint Objetoint(10);
      Matrizdouble Objetodouble(10);
      Objetoint[0]=10;
      Objetodouble[5]=24.5;
      cout << Objetoint[0] << '\n' << Objetodouble[5];
}
```

3.17.2. Realización

En cada uno de los archivos que necesiten hacer uso de una clase será necesario incluir los archivos cabecera en los cuales están definidas las clases que van a ser usadas. Sin embargo, normalmente estas bibliotecas son compiladas obteniendo ficheros .OBJ que serán enlazados junto a las aplicaciones.

En el caso de que tengamos varios ficheros se puede, o bien generar un código objeto por cada uno de los archivos fuente, o bien varios archivos .OBJ, o bien unirlos todos en una biblioteca .LIB.
En definitiva, el proceso de creación y utilización de una biblioteca de funciones podría simplificarse a:

1.- Definición de las clases en los archivos cabeceras.
2.- Codificación de los métodos en uno o varios archivos fuentes, incluyendo archivos de cabecera.
3.- Compilación de cada uno de los archivos fuente, con la consiguiente generación del código objeto.
4.- Unión de los códigos objetos en los distintos archivos en una sola biblioteca con cada objeto.
5.- Incluir los archivos de cabecera en la aplicación que use la biblioteca y enlazarla con ella.

3.17.3. Entrada y Salida Básica en C++

cin y **cout** son dos flujos (entrada y salida) de datos ertenecientes a la biblioteca de **C++** llamada iostream, y se utilizan para lo siguiente:

cin>>*dato* *: Almacena el carácter introducido por teclado en* ***dato.***
cout<<*dato* : *Lee el contenido de* ***dato*** *y lo muestra por pantalla.*

3.18. *STREAMS*

Un *Stream* es un flujo de datos, y para que exista es necesario que haya un origen abierto conteniendo datos, y un destino abierto y preparado para recibir datos. Entre el origen y el destino debe haber una conexión por donde puedan fluir los datos. Estas conexiones se llevan a cabo a través de operadores y métodos de C++. Los *streams* se usan de forma genérica, independientemente de cuales sean el origen y el destino.

3.18.1. Tipos de *Stream*

Los más normales son los que se establecen entre un programa y la consola: el teclado y la pantalla. También entre un programa y un

fichero o entre dos zonas de memoria. En algunos casos es necesario abrir el *stream* antes de usarlo, como es el caso de los ficheros, y otras veces no, como es el caso de las consolas.

3.18.2. Operadores de Inserción y Extracción

Todos los *streams* van a disponer de estos dos operadores. El operador de inserción, << , es el operador de desplazamiento de bits a la izquierda, que está definido en la clase ostream perteneciente a iostream.h, y **>>**, es el operador de desplazamiento a la derecha y está definido en la clase istream, definida en el mismo archivo cabecera. Estos operadores van a disponer a su izquierda un objeto *stream* que será origen o destino, dependiendo del operador usado, y a la derecha lo que se va a escribir o donde se va a almacenar lo leído. Ya se han usado *streams*: cout y cin que tienen por defecto la salida estándar (pantalla) y la entrada estándar (teclado). También existe cerr para la salida estándar de errores del programa.

3.18.3. Entrada/Salida con Formato

La E/S de datos se realiza por medio de los operadores de inserción y extracción en *streams* y puede formatearse por ejemplo para ajustar a la derecha o a la izquierda con una longitud mínima o máxima. Para esto se definen métodos y manipuladores.

Los métodos son aquellos a los que se les llama de forma habitual (*con el nombre del objeto a la izquierda*). En cambio, los manipuladores pueden actuar en cualquier punto a través de los operadores de inserción y extracción.

3.18.4. Anchura

Por defecto, los datos que entran y salen tienen una anchura que se corresponde con el espacio necesario para almacenar ese dato. Para fijar la anchura, tanto se puede usar el método width() como el manipulador setw(). Si se usa width() sin parámetro devolverá la anchura fijada, que por defecto es 0. Esto significa que se tomen los carácteres necesarios para la representación del dato en el *stream* de

salida. Si le pasamos un parámetro se fijará la anchura a ese número de carácteres.

3.18.5. Carácter de Relleno

Cuando se ajusta una salida, y quedan espacios libres, estos se rellenan con espacios en blanco. Este carácter de relleno se puede modificar con el método fill() o el manipulador setfill(). Llamando a fill() nos devuelve el carácter de relleno que se está usando en ese momento.

3.18.6. Otros Manipuladores

Para el objeto cout existen además:

endl()
ends()
flush()

endl()**:** Inserta un retorno de carro.
ends()**:** Inserta un terminador de cadena
flush()**:** Fuerza la salida de todos los carácteres del *buffer* al *stream*.

3.18.7. Otros Métodos de E/S

Los operadores de inserción y extracción permiten realizar entradas y salidas con formato. Sin embargo, a veces puede ser necesario leer o escribir información sin formato alguno. Para ello también se dispone de algunos operadores en las clases **istream** y **ostream**.

Métodos	*Acción*
ios::Skipws	Cuando está activo ignora los espacios en blanco iniciales
ios::left	Activa la justificación izquierda en la salida de datos
ios::right	Activa la justificación derecha en la salida de datos
ios::internal	Permite que en las salidas el signo o indicador de base, se muestre a la izquierda del relleno como primer carácter
ios::dec	Activa conversión decimal
ios::oct	Activa conversión octal
ios::hex	Activa conversión hexadecimal
ios::showbase	Mostrará el indicador de base en las salidas
ios::showpoint	Muestra el punto decimal y los dígitos decimales necesarios para completar la salida
ios::uppercase	Se usan letras mayúsculas en la salida
ios::showpos	Está desactivado por defecto y lo que hace es mostrar el signo en las salidas numéricas
ios::scientific	Realiza las salidas de punto flotante con notación científica
ios::unitbuf	Vuelca los caracteres de E/S en el dispositivo origen o destino
ios::fixed	Realiza la notación decimal

3.18.8. El Polifacético Método get

get() es un método para realizar lecturas o entradas de datos desde un objeto. Sus posibles formas son:

get(): Toma el siguiente carácter y lo devuelve. En caso de que haya encontrado final del *stream*, devuelve EOF.

get(char *, int, char): El primer parámetro es la dirección donde se almacenará la cadena de caracteres, el segundo la longitud y el tercero el carácter delimitador, para que cuando lo encuentre se pare. Se va a detener cuando:

a) Encuentre el delimitador.

b) Encuentre el final del *stream*.

Si no se da el último carácter se tomará por defecto '\n'.

get(char &): Toma un carácter del *stream* y lo almacena en la variable cuya referencia se toma como parámetro.

3.18.9. Lectura de Líneas

El método **getline()** toma los mismos parámetros que **get()** y el mismo valor por defecto para el delimitador. Se almacena el delimitador antes de añadir el carácter nulo.

3.18.10. Lectura Absoluta

Mediante el método **read()** se puede leer un número de caracteres y almacenarlo en el espacio de memoria indicado. Sólo se detiene si llega al final del *stream* y no ha leído la cantidad indicada, pero no lo hace cuando se encuentre un delimitador o el carácter nulo. Como primer parámetro pasamos el puntero a donde se almacenará la cadena, y como segundo la longitud. Mediante el método **gcount()** es posible saber el número de bytes leídos (nos lo devuelve).

3.18.11. Movimiento en un Stream

Es posible desplazar el puntero de lectura en un stream mediante los métodos:

Métodos	*Parámetros y Acción*
Seekg()	Pasándole un entero largo se sitúa en una posición absoluta
ios::beg	Se sitúa en una posición relativa moviéndose desde el principio del *stream* e indicándole un desplazamiento en forma de entero largo
ios::cur	Lo mismo que antes pero parte de la posición actual
ios::end	Lo mismo que antes pero parte desde el final
tellg()	Nos devuelve la posición actual en el stream en forma de entero largo

Otras Operaciones de Lectura

Métodos	*Acción*
peek()	Método que devolverá en forma de entero el código del siguiente carácter disponible sin llegar a extraerlo
putback()	Método que toma como parámetro un carácter que se va a devolver al *stream*. Sólo se usa cuando se toma un carácter que se quiera devolver
ignore()	Método que ignora un número específico de caracteres saltándolos hasta encontrar el delimitador o hasta completar el número especificado. Toma dos parámetros: Número de caracteres a saltar y el delimitador

3.18.12. Escritura sin Formato

Existen dos métodos principales para la escritura sin formato que son:
Put(): Que toma como parámetro un carácter al que da salida por el *stream.*

Write(): Toma los mismos parámetros que read() permitiendo escribir un bloque de bytes del tamaño indicado.

También existen los métodos para desplazar el puntero de escritura en el *stream*:

seekp()=Seekg()
tellp()=tellg()

Mostramos un ejemplo del uso de stream

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

void main(void)
{
      char Cadena[20];
      int           N;

      cout << "Introduzca una cadena :";
      cin.getline(Cadena,20);  // Se toma una cadena de 20
                               // caracteres como máximo.
      cout << endl << endl;    // Dos líneas en blanco
      cout.write(Cadena,20); // Se da salida a los 20 caracteres
      cout.flush(); // Vuelca el stream al correspondiente
                    // dispositivo.
      cout << endl << "Introduzca un número :";
      cin >> N;
      cout << endl << "Su representación es :";
      cout.write(char *) &N, sizeof(N);  // Escribe su
                                         // representación en memoria
      cout.flush();
}
```

3.18.13. Los Streams y los Buffers

A cada stream se le asocia un buffer, para mejorar los tiempos de acceso. Mediante el método **rdbuf()** obtenemos un puntero al buffer correspondiente a un stream. Los métodos que nos van a permitir acceder a estos buffers podemos dividirlos en tres grupos:

Grupo 1: Lectura del *buffer*

Métodos	*Acción*
in_avail()	Devuelve un entero indicando el número de caracteres que hay actualmente en el *buffer* de entrada, tomados desde el *stream* de lectura
sbumpc()	Lee un carácter del *buffer* de entrada, lo devuelve y avanza una posición
snextc()	Avanza una posción, lee un carácter del *buffer* y lo devuelve
sgetc()	Lee el carácter actual apuntado en el *buffer* de entrada y lo devuelve, pero no avanza
stossc()	Avanza al siguiente carácter
sgetn()	Este método necesita dos parámetros, un puntero a carácter y un entero. Lee el número de caracteres indicado del *buffer* de entrada y lo almacena en la dirección indicada por el puntero
sputbackc()	Devuelve un carácter al *buffer* de entrada, especificándoselo como parámetro

Grupo 2: Escritura en el *buffer*

Métodos	*Acción*
out_waiting()	Devuelve un entero indicando el número de caracteres que hay en el *buffer* de salida esperando a escribirse en el *stream*
sputc()	Permite escribir un carácter al *buffer* de salida, carácter que habrá que pasar como parámetro
sputn()	Permite escribir en el *buffer* de salida un determinado número de caracteres. Toma dos parámetros: Puntero a carácter y un entero

Grupo 3: Posicionamiento en el *buffer*

Métodos	*Acción*
seekpos()	Posiciona el puntero en una posición absoluta dada por un entero largo, el cual se pasa como parámetro. El segundo parámetro indica el puntero que se va a fijar, el de entrada o el de salida
seekoff()	Toma tres parámetros y la posición del puntero de forma relativa según el segundo parámetro que puede ser: `ios::beg`, `ios::cur`, `ios::end`, según se desee desplazar desde el principio, desde la posición actual o desde el final. El tercer parámetro es el indicador del puntero a desplazar `ios::in`, `ios::out`. Por defecto, `ios::in/ios::out`

3.19. Miscelanea de Programas

```
// Ejemplo 1 de C++ para el laboratorio de POO
// * Creación de una clase
// * Public vs Private
// * Atributos vs Métodos

#include <iostream>
#include <conio.h>
using namespace std;

class Punto
{
   public:
      int LeerX() {return x;}
      int LeerY() {return y;}
      void FijarX (int valor) {x = valor;}
      void FijarY (int valor) {y = valor;}
   private:
      int x;
      int y;
};

main()
{
   int a,b;
   Punto p;

   cout << "Dame las coordenadas x e y de un punto: ";
   cin >> a >> b,
   p.FijarX(a);
   p.FijarY(b);
   cout << "El valor de x en el punto p es " << p.LeerX() << endl;
   cout << "El valor de y en el punto p es " << p.LeerY() << endl;

   cout << "\nPulsa una tecla\n";
   getch();
}
// Ejemplo 2 de C++ para el laboratorio de POO
// * Definición externa de funciones de clase
```

```
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
class Punto
{    public:
       double LeerX() {return x;}
       double LeerY() {return y;}
       void FijarX (int valor) {x = valor;}
       void FijarY (int valor) {y = valor;}
       double Distancia();
   private:
       int x;
       int y;
};

double Punto::Distancia()
{
   return sqrt(pow(LeerX(),2)+pow(LeerY(),2));
}

main()
{
   int a,b;
   Punto p;
   cout << "Dame las coordenadas x e y de un punto: ";
   cin >> a >> b,
   p.FijarX(a);
   p.FijarY(b);
   cout << "El valor de x en el punto p es " << p.LeerX() << endl;
   cout << "El valor de y en el punto p es " << p.LeerY() << endl;
   cout << "La distancia al origen del punto p es " << p.Distancia() << endl;
   cout << "\nPulsa una tecla\n";
   getch();
}
// Ejemplo 3 de C++ para el laboratorio de POO
// * Constructor de una clase
// * Sobrecarga del constructor
```

```
// * Formas de crear objetos de una clase
// * Funciones que usan clases
// * Como NO hacer métodos de clase

#include <iostream>
#include <conio.h>
using namespace std;

class Fraccion
{
   public:
      Fraccion() {num=0; den=1;}
      Fraccion(int x, int y)
      {
         if (y!=0)
         {num=x; den=y; Reducir();}
         else
         {num=0; den=1;};
      }
      void Escribir() {cout << num << "/" << den;}
      void Crear()
      {
         cout << "Numerador: ";
         cin >> num;
         cout << "Denominador: ";
         cin >> den;
      }
      int GetNum() {return num;}
      int GetDen() {return den;}
   private:
      int num,den;
      int MCD(int a, int b)
      {
         int aux;
         while (a%b!=0)
         {
            aux=a%b;
            a=b;
```

```
            b=aux;
         };
         return b;
      }
      void Reducir()
      {
         int aux=MCD(abs(num),abs(den));
         num/=aux;
         den/=aux;
         if (den<0)
         {
            num=-num;
            den=-den;
         };
         return;
      }
};

Fraccion Sumar(Fraccion x, Fraccion y)
{
   int auxden=x.GetDen()*y.GetDen();
   int auxnum=(x.GetNum()*y.GetDen())+(x.GetDen()*y.GetNum());
   Fraccion f(auxnum,auxden);
   return f;
}

main()
{
   cout << "Ejemplos con constructor" << endl;
   Fraccion f;
   f.Escribir();
   cout << endl;
   Fraccion f2(4,6);
   f2.Escribir();
   cout << endl;
   Fraccion *f3 = new Fraccion(14,8);
   (*f3).Escribir();
   cout << endl << endl;;
```

```
   cout << "Ejemplos con introduccion" << endl;
   f.Crear();
   f2.Crear();
   *f3=Sumar(f,f2);
   (*f3).Escribir();

   cout << "\nPulsa una tecla\n";
   getch();
}
```

// **Ejemplo 4 de C++ para el laboratorio de POO**
// * Destructor de una clase
// * Liberación de memoria

```
#include <iostream>
#include <conio.h>
using namespace std;

class Fraccion
{
   public:
      Fraccion() {num=0; den=1;}
      Fraccion(int x, int y)
      {
         if (y!=0)
         {num=x; den=y; Reducir();}
         else
         {num=0; den=1;};
      }
      ~Fraccion() {cout << "Se acaba de destruir "
                  << num << "/" << den << endl;}      // Destructor
      void Escribir() {cout << num << "/" << den;}
      int GetNum() {return num;}
      int GetDen() {return den;}
   private:
      int num,den;
      int MCD(int a, int b)
```

```
    {
       int aux;
       while (a%b!=0)
       {
          aux=a%b;
          a=b;
          b=aux;
       };
       return b;
    }
    void Reducir()
    {
       int aux=MCD(abs(num),abs(den));
       num/=aux;
       den/=aux;
       if (den<0)
       {
          num=-num;
          den=-den;
       };
       return;
    }
};

Fraccion Sumar(Fraccion x, Fraccion y)
{
   int auxden=x.GetDen()*y.GetDen();
   int auxnum=(x.GetNum()*y.GetDen())+(x.GetDen()*y.GetNum());
   Fraccion f(auxnum,auxden);
   return f;
}

main()
{
   Fraccion f;
   Fraccion f2(4,6);
   Fraccion *f3 = new Fraccion(14,8);
   Fraccion f4 = Fraccion(2,7);
```

```
   cout << "Vamos a sumar ";
   f2.Escribir();
   cout << " y ";
   f4.Escribir();
   cout << endl << endl;
   f=Sumar(f2,f4); // Al acabar la función se destruyen las fracciones locales
                   // y se dispara el destructor
   cout << "El resultado es ";
   f.Escribir();
   cout << endl << endl;

   cout << "Vamos a destruir ";
   (*f3).Escribir();
   cout << endl;
   delete f3; // Destrucción voluntaria

   cout << "\nPulsa una tecla\n";
   getch();
}

// Ejemplo 5 de C++ para el laboratorio de POO
// * Reserva de memoria
// * Un poquito de cadenas
// * Sobrecarga de operadores

#include <iostream>
#include <conio.h>
using namespace std;

class Fraccion
{
   public:
      Fraccion() {num=0; den=1;}

      Fraccion(int x, int y)
      {
         if (y!=0)
         {num=x; den=y; Reducir();}
```

```
      else
      {num=0; den=1;};
   }
   ~Fraccion() {cout << "Se acaba de destruir "
                << EscribirBien() << endl;}
   void Escribir() {cout << num << "/" << den;}
   char* EscribirBien()
   {
      char* dato= (char*) malloc(sizeof(char));
      sprintf(dato, "%i/%i", num, den);
      return dato;
   }
   int GetNum() {return num;}
   int GetDen() {return den;}
   Fraccion operator -() {return Fraccion(-num,den);}
                                                  // operador - unario
   Fraccion operator +(Fraccion f2)  // operador + binario
   {
      int auxden=GetDen()*f2.GetDen();
      int auxnum=(GetNum()*f2.GetDen())+(GetDen()*f2.GetNum());
      return Fraccion(auxnum,auxden);
   }
private:
   int num,den;
   int MCD(int a, int b)
   {
      int aux;
      while (a%b!=0)
      {
         aux=a%b;
         a=b;
         b=aux;
      };
      return b;
   }
   void Reducir()
   {
      int aux=MCD(abs(num),abs(den));
```

```
        num/=aux;
        den/=aux;
        if (den<0)
        {
           num=-num;
           den=-den;
        };
        return;
     }
};

Fraccion Sumar(Fraccion x, Fraccion y)
{
   int auxden=x.GetDen()*y.GetDen();
   int auxnum=(x.GetNum()*y.GetDen())+(x.GetDen()*y.GetNum());
   Fraccion f(auxnum,auxden);
   return f;
}

main()
{
   cout << "Datos: " << endl;
   Fraccion f(7,5);
   Fraccion f2(4,6);
   f.Escribir();
   cout << endl;
   f2.Escribir();
   cout << endl;

   cout << "Uso de una funcion para sumar... " << endl;
   cout << f.EscribirBien() << "+" << f2.EscribirBien() << "="
      << Sumar(f,f2).EscribirBien() << endl;
   cout << "Uso de operadores para sumar... " << endl;
   Fraccion f4=Fraccion(2,3)+Fraccion(4,3);
   f4.Escribir();
   cout << endl;
   cout << "... y para cambiar de signo" << endl;
   Fraccion f3=-f2;
```

```
   f3.Escribir();
   cout << endl;

   cout << "\nPulsa una tecla\n";
   getch();
}

// Ejemplo 6 de C++ para el laboratorio de POO
// * Sobrecarga de operadores
// * Direccionamiento de salida
// * Clases amigas
// * this

#include <iostream>
#include <conio.h>
using namespace std;

class Fraccion
{
   public:
      Fraccion() {num=0; den=1;}            // Constructor por defecto
      Fraccion(int x, int y)
      {
         if (y!=0)
         {num=x; den=y; Reducir();}
         else
         {num=0; den=1;};
      }
      ~Fraccion() {}
      char* EscribirBien()
      {
         char* dato= (char*) malloc(sizeof(char));
         sprintf(dato, "%i/%i", num, den);
         return dato;
      }
      int GetNum() {return num;}
      int GetDen() {return den;}
      Fraccion operator -() {return Fraccion(-num,den);}
```

```
                                        // operador - unario
    Fraccion operator +(Fraccion f2)   // operador + binario
    {
       int auxden=GetDen()*f2.GetDen();
       int auxnum=(GetNum()*f2.GetDen())+(GetDen()*f2.GetNum());
       return Fraccion(auxnum,auxden);
    }
    Fraccion operator -(Fraccion f2)   // operador - binario
    {
       //return *this + (-f2); // Referencia al objeto actual
       return Fraccion(num,den) + (-f2);
    }
    friend ostream& operator << (ostream &os, Fraccion f);
 private:
    int num,den;
    int MCD(int a, int b)
    {
       int aux;
       while (a%b!=0)
       {
          aux=a%b;
          a=b;
          b=aux;
       };
       return b;
    }
    void Reducir()
    {
       int aux=MCD(abs(num),abs(den));
       num/=aux;
       den/=aux;
       if (den<0)
       {
          num=-num;
          den=-den;
       };
       return;
    }
```

```
};

Fraccion Sumar(Fraccion x, Fraccion y)
{
   int auxden=x.GetDen()*y.GetDen();
   int auxnum=(x.GetNum()*y.GetDen())+(x.GetDen()*y.GetNum());
   Fraccion f(auxnum,auxden);
   return f;
}

main()
{
   Fraccion f(7,5);
   Fraccion f2(4,6);
   cout << "Mediante funcion: " << endl;
   cout << f.EscribirBien() << "+" << f2.EscribirBien() << "="
        << Sumar(f,f2).EscribirBien() << endl;
   cout << endl;

   cout << "Mediante operadores: " << endl;
   cout << Fraccion(3,4).EscribirBien() << "+"
        << Fraccion(7,4).EscribirBien() << "="
        << (Fraccion(3,4)+Fraccion(7,4)).EscribirBien() << endl;
   cout << Fraccion(3,4).EscribirBien() << "-"
        << Fraccion(7,4).EscribirBien() << "="
        << (Fraccion(3,4)-Fraccion(7,4)).EscribirBien() << endl;
   cout << endl;

   cout << "Escritura directa: " << endl;
   Fraccion f4=Fraccion(2,3)+Fraccion(4,3);
   cout << f4 << endl;

   cout << "\nPulsa una tecla\n";
   getch();
}

   ostream& operator << (ostream &os, Fraccion f)
      {
```

```
        os << f.num << "//" << f.den;
        return os;
    }

// Ejemplo 7 de C++ para el laboratorio de POO
// * Plantillas

#include <iostream>
#include <conio.h>
using namespace std;

template <class Tipo> Tipo MiMin (Tipo a, Tipo b)
{
  if (a<=b) return a;
  else return b;
}

template <class Tipo> void Cambiar (Tipo& a, Tipo& b)
{
  Tipo aux;
  aux=a;
  a=b;
  b=aux;
  return;
}

template <class Tipo> void Escribir (Tipo m[], int Elem)
{
  Elem=Elem/sizeof(Tipo);
  cout << "Hay " << Elem << " datos." << endl;
  for (int i=0; i<Elem; i++)
     cout << "Dato " << i << ": " << m[i] << endl;
  cout << endl;
  return;
}


main()
```

```
{
   cout << "Uso de la plantilla MiMin." << endl;
   cout << MiMin(3,4) << endl;
   cout << MiMin('g', 'c') << endl;
   cout << MiMin(4.56353,-234.2) << endl;
   cout << endl;

   cout << "Uso de la plantilla Cambiar." << endl;
   char a='j', b='s';
   cout << "Antes del cambio" << endl;
   cout << "a: " << a << ". b: " << b << endl;
   Cambiar(a,b);
   cout << "Despues del cambio" << endl;
   cout << "a: " << a << ". b: " << b << endl;
   cout << endl;

   int datos[8]={3,2,4,3,7,99,-3,2};
   char cosas[5]={'J','e','s','u','s'};
   char frase[]="Hola mundo";
   cout << "Uso de la plantilla Escribir para vectores." << endl;
   Escribir (datos,sizeof(datos));
   Escribir (cosas,sizeof(cosas));
   Escribir (frase,sizeof(frase)); // Ojo con el caracter "extra"

   cout << "\nPulsa una tecla\n";
   getch();
}
```

/ **Ejemplo 8 de C++ para el laboratorio de POO**

```
// * Mas Plantillas

#include <iostream>
#include <conio.h>
using namespace std;

const int Tope=10;

template <class Tipo> class Vector
```

```
{
   private:
      Tipo valor[Tope];
      int cuantos;
   public:
      Vector() {cuantos=0;}
      Vector(int c)
      {
         if ((c>Tope) || (c<0)) cuantos=0;
         else cuantos=c+1;
         for (int i=0;i<cuantos;i++)
            valor[i]=0;
         return;
      }
      void Escribir()
      {
         if (cuantos>0)
         {
            for (int i=0;i<cuantos-2;i++)
               cout << valor[i] << ", ";
            cout << valor[cuantos-1] << endl;;
         };
         return;
      }
      void Cargar(Tipo dato)  // Rellena la estructura con el valor dato
      {
         if (cuantos>0)
         {
            for (int i=0;i<cuantos;i++)
               valor[i]=dato;
         };
         return;
      }
      void Poner(Tipo dato, int posicion) // Pone el dato en una posición
      {
         if ((posicion>0) && (posicion<=cuantos))
            valor[posicion]=dato;
         return;
```

```
      }
}   ;


main()
{
   Vector <int> datos(5);
   Vector <char> cosas(10);
   datos.Cargar(55);
   datos.Escribir();
   cosas.Cargar('x');
   cosas.Poner('$',4);
   cosas.Escribir();


   cout << "\nPulsa una tecla\n";
   getch();
}
```

```
// Ejemplo 9: Herencia
#include <iostream>
#include <string>
using namespace std;

/* Clase de objetos CCuenta.
 *   Atributos:
 *     nombre, cuenta, saldo y tipo de interés
 *   Métodos:
 *     asignar/obtener nombre
 *     asignar/obtener cuenta
 *     estado de la cuenta (saldo)
 *     ingreso
 *     reintegro
 *     asignar/obtener tipo de interés
 */
class CCuenta
{
 // Atributos
 private:
   string nombre;
   string cuenta;
   double saldo;
   double tipoDeInteres;

 // Métodos
 public:
   CCuenta()
   {
     saldo = 0.0;
     tipoDeInteres = 0.0;
   }

   CCuenta(string nom, string cue, double sal, double tipo)
   {
     asignarNombre(nom);
     asignarCuenta(cue);
     saldo = sal;
```

```
  tipoDeInteres = tipo;
}

void asignarNombre(string nom)
{
  if (nom.length() == 0)
  {
    cout << "Error: cadena vacía\n";
    return;
  }
  nombre = nom;
}

string obtenerNombre()
{
  return nombre;
}

void asignarCuenta(string cue)
{
  if (cue.length() == 0)
  {
    cout << "Error: cuenta no válida\n";
    return;
  }
  cuenta = cue;
}

string obtenerCuenta()
{
  return cuenta;
}

double estado()
{
  return saldo;
}
```

```
  void ingreso(double cantidad)
  {
   if (cantidad < 0)
   {
    cout << "Error: cantidad negativa\n";
    return;
   }
   saldo = saldo + cantidad;
  }

  void reintegro(double cantidad)
  {
   if (saldo - cantidad < 0)
   {
    cout << "Error: no dispone de saldo\n";
    return;
   }
   saldo = saldo - cantidad;
  }

  void asignarTipoDeInteres(double tipo)
  {
   if (tipo < 0)
   {
    cout << "Error: tipo no válido\n";
    return;
   }
   tipoDeInteres = tipo;
  }

  double obtenerTipoDeInteres()
  {
   return tipoDeInteres;
  }
};
/* Clase de objetos CCuentaAhorro.
 */
class CCuentaAhorro : public CCuenta
```

```
{
 // Atributos
 private:
  double cuotaMantenimiento;
 // Métodos
 public:
  CCuentaAhorro() {} // constructor sin parámetros

  CCuentaAhorro(string nom, string cue, double sal,
   double tipo, double mant) :
   CCuenta(nom, cue, sal, tipo)// invoca al constructor CCuenta,
  {                          // esto es, al de la clase base
   asignarCuotaManten(mant);   // inicia cuotaMantenimiento
  }

  void asignarCuotaManten(double cantidad)
  {
   if (cantidad < 0)
   {
    cout << "Error: cantidad negativa\n";
    return;
   }
   cuotaMantenimiento = cantidad;
  }

  double obtenerCuotaManten()
  {
   return cuotaMantenimiento;
  }

  void reintegro(double cantidad)
  {
   double saldo = estado();
   double tipoDeInteres = obtenerTipoDeInteres();

   if ( tipoDeInteres >= 3.5)
   {
    if (saldo - cantidad < 1500)
```

```
      {
        cout << "Error: no dispone de esa cantidad\n";
        return;
      }
    }
    // Invocar al método reintegro de la clase base,
    // también llamada superclase
    CCuenta::reintegro(cantidad);
  }
};

/* Función main:
 *   Punto de entrada y de salida al programa.
 *   Crea una cuenta, cuenta01, y realiza
 *   operaciones sobre ella. */

int main()
{
 CCuenta cuenta01;
 CCuentaAhorro cuenta02("Un nombre", "Una cuenta", 6000, 3.5, 2);

 cuenta01.asignarNombre("Un nombre");
 cuenta01.asignarCuenta("Una cuenta");
 cuenta01.asignarTipoDeInteres(2.5);
 cuenta01.ingreso(12000);
 cuenta01.reintegro(3000);

 cout << cuenta01.obtenerNombre() << endl;
 cout << cuenta01.obtenerCuenta() << endl;
 cout << cuenta01.estado() << endl;
 cout << cuenta01.obtenerTipoDeInteres() << endl;
 cout << endl;

// Cobrar cuota de mantenimiento
 cuenta02.reintegro(cuenta02.obtenerCuotaManten());
 // Ingreso
 cuenta02.ingreso(6000);
 // Reintegro
```

```
  cuenta02.reintegro(10000);

  cout << cuenta02.obtenerNombre() << endl;
  cout << cuenta02.obtenerCuenta() << endl;
  cout << cuenta02.estado() << endl;
  cout << cuenta02.obtenerTipoDeInteres() << endl;
}
```

// Ejemplo 10
// friend.cpp - Métodos de una clase amigos de otra

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class C1; // declaración adelantada de C1

class C2
{
  private:
    int nc2;
  public:
    void AsignarDato(int n) { nc2 = n; }
    int ObtenerDato(const C1&);
};

class C1
{
  friend int C2::ObtenerDato(const C1&);
  private:
    int nc1;
  public:
    void AsignarDato(int n) { nc1 = n; }
};

int C2::ObtenerDato(const C1& obj)
{
  return obj.nc1 + nc2;
}
```

```
int main()
{
  C1 objeto1; // objeto de la clase C1
  C2 objeto2; // objeto de la clase C2
  int dato;

  cout << "Nº entero: "; cin >> dato;
  objeto1.AsignarDato(dato);
  cout << "Nº entero: "; cin >> dato;
  objeto2.AsignarDato(dato);
  cout << "\nResultado: ";
  cout << objeto2.ObtenerDato(objeto1) << endl;
}
```

// **Ejemplo 11 virtual**

```
// virtual.cpp - Métodos virtuales y no virtuales
#include <iostream>
using namespace std;

class CB
{
  public:
    virtual void mVirtual1(); // método virtual
    void mNoVirtual();       // método no virtual
};
void CB::mVirtual1()
{
  cout << "método virtual 1 en CB\n";
}
void CB::mNoVirtual()
{
  cout << "método no virtual en CB\n";
}

class CD1 : public CB
{
  public:
    void mVirtual1();        // método virtual
```

```
   virtual void mVirtual2(); // método virtual
   void mNoVirtual();       // método no virtual
};
void CD1::mVirtual1()
{
 cout << "método virtual 1 en CD1\n";
}
void CD1::mVirtual2()
{
 cout << "método virtual 2 en CD1\n";
}
void CD1::mNoVirtual()
{  cout << "método no virtual en CD1\n";
}

class CD2 : public CD1
{
 public:
   void mVirtual1();  // método virtual
   void mVirtual2();  // método virtual
   void mNoVirtual(); // método no virtual
};
void CD2::mVirtual1()
{
 cout << "método virtual 1 en CD2\n";
}
void CD2::mVirtual2()
{
 cout << "método virtual 2 en CD2\n";
}
void CD2::mNoVirtual()
{  cout << "método no virtual en CD2\n";
}

int main()
{
 CB *p1CB = new CD1;   // puntero a CB que apunta a un objeto CD1
 CD1 *p1CD1 = new CD2; // puntero a CD1 que apunta a un objeto CD2
```

```
  CB *p2CB = new CD2;   // puntero a CB que apunta a un objeto CD2
    // Llamadas a los métodos
  p1CB->mVirtual1();      // llama a CD1::mVirtual1
  p1CB->mNoVirtual();     // llama a CB::mNoVirtual
  p1CB->CB::mVirtual1();  // llama a CB::mVirtual1
  p1CD1->mVirtual2();     // llama a CD2::mVirtual2
  p1CD1->mNoVirtual();    // llama a CD1::mNoVirtual
  p1CD1->CD1::mVirtual2(); // llama a CD1::mVirtual2
  p2CB->mVirtual1();      // llama a CD2::mVirtual1

  delete p1CB;
  delete p1CD1;
  delete p2CB;
}
```

// **Ejemplo 12 Herencia Multiple**

```
#include <iostream>
using namespace std;

class CTermino  // clase base
{
  private:
    double coeficiente;
  public:
    // Constructor
    CTermino(double k = 1) : coeficiente(k) {}
    double coef() { return coeficiente; }
    virtual void mostrar() = 0;
};

class CTerminoEnX : public virtual CTermino
{
  private:
    int exponenteDeX;
  public:
    // Constructor
    CTerminoEnX(double k = 1, int e = 0) :
    CTermino(k), exponenteDeX(e) {}
```

```
   int expX() { return exponenteDeX; }
   void mostrar() { cout << coef() << "x^" << exponenteDeX; }
};

class CTerminoEnY : public virtual CTermino
{
 private:
   int exponenteDeY;
 public:
   // Constructor
   CTerminoEnY(double k = 1, int e = 0) :
   CTermino(k), exponenteDeY(e) {}
   int expY() { return exponenteDeY; }
};

class CTerminoEnXY : public CTerminoEnX, public CTerminoEnY
{
 public:
   // Constructor
   CTerminoEnXY(double k = 1, int ex = 0, int ey = 0) :
   CTermino(k), CTerminoEnX(k, ex), CTerminoEnY(k, ey) {}
   void mostrar() { cout << coef() << "x^" << expX()
                    << "y^" << expY(); }
};

ostream& operator<<(ostream& os, CTermino* t)
{
 t->mostrar();
 return os;
}

void f(CTerminoEnXY* p)
{
 // Clase base normal
 CTerminoEnX *pTx = p; // D a B
 CTerminoEnXY *pTxy = static_cast<CTerminoEnXY *>(pTx); // B a D
 pTxy = dynamic_cast<CTerminoEnXY *>(pTx); // B a D
```

```
  // Clases hermanas
  CTerminoEnY *pTy = dynamic_cast<CTerminoEnY *>(pTx); // h a H
  //pTy = static_cast<CTerminoEnY *>(pTx); // H a H. Error
  cout << pTy << endl;

  // Clase base virtual
  CTermino *pT = p;
  //pTxy = static_cast<CTerminoEnXY *>(pT); // Bv a D. Error.
  pTxy = dynamic_cast<CTerminoEnXY *>(pT); // Bv a D.
  cout << pTxy << endl;
  pTx = dynamic_cast<CTerminoEnXY *>(pT); // Bv a D.
  cout << pTx << endl;
}

int main()
{
  CTerminoEnX *pCTerminoEnX = new CTerminoEnX(3, 2);
  cout << pCTerminoEnX << endl;
  CTerminoEnXY *pCTerminoEnXY = new CTerminoEnXY(2, 5, 3);
  cout << pCTerminoEnXY << endl;
  f(pCTerminoEnXY);
  delete pCTerminoEnX;
  delete pCTerminoEnXY;
  cout << endl;
}
```