Ingeniería en Electrónica
Informática II
Clases y objetos

Ing José Luis MARTÍNEZ

28 de octubre de 2021

1. Programación orientada a Objetos

La programación orientada a objetos (POO), tal vez el paradigma de programación más utilizado en el mundo del desarrollo de software y de la ingeniería de software del siglo XXI, trae un nuevo enfoque a los retos que se plantean en la programación estructurada cuando los problemas a resolver son complejos. Al contrario que la programación procedimental que enfatiza en los algoritmos, la POO enfatiza en los datos. En lugar de intentar ajustar un problema al enfoque procedimental de un lenguaje, POO intenta ajustar el lenguaje al problema. La idea es diseñar formatos de datos que se correspondan con las características esenciales de un problema. La idea fundamental de los lenguajes orientados a objetos es combinar en una única unidad o módulo, tanto los datos como las funciones que operan sobre esos datos. Tal unidad se llama objeto.

Las funciones de un objeto se llaman *funciones miembro en C++ o métodos* (éste es el caso de Smalltalk, uno de los primeros lenguajes orientados a objetos), y son el único medio para acceder a sus datos. Los *datos de un objeto, se conocen también como atributos o variables de instancia.* Si se desea leer datos de un objeto, se llama a una función miembro del objeto. Se accede a los datos y se devuelve un valor. No se puede acceder a los datos directamente. Los datos están ocultos, de modo que están protegidos de alteraciones accidentales. Los datos y las funciones se dice que están encapsulados en una única entidad. El encapsulamiento de datos y la ocultación de los datos son términos clave en la descripción de lenguajes orientados a objetos (figura 1).

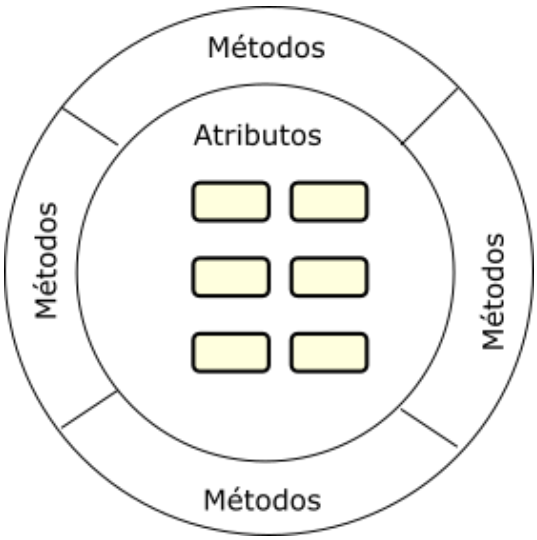

Figura 1: Esquema de la POO, los atributos (datos) están ocultos y solamente pueden ser accedidos a través de los métodos (funciones).

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles son las funciones que interactúan con las funciones miembro del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa. Un programa C++ se compone, normalmente,

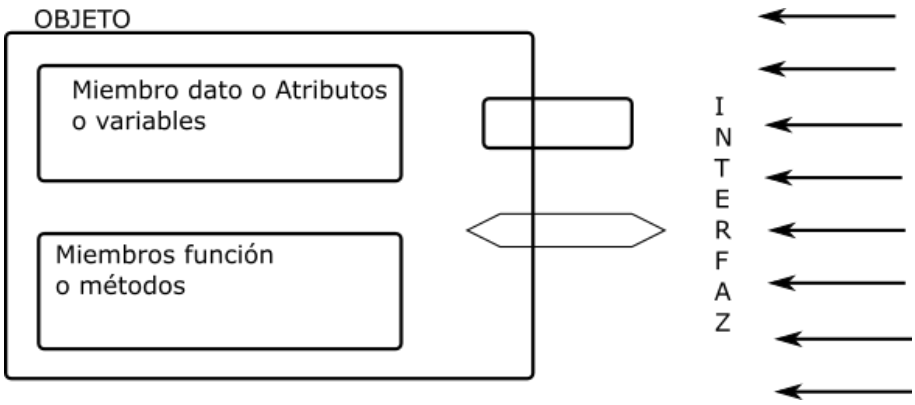

Figura 2: Esquema de la POO.

de un número de objetos que se comunican unos con otros mediante la llamada a otras funciones miembro. La organización de un programa en C++ se muestra en la Figura . La llamada a una función miembro de un objeto se denomina *enviar un mensaje a* otro objeto (Figura 3).

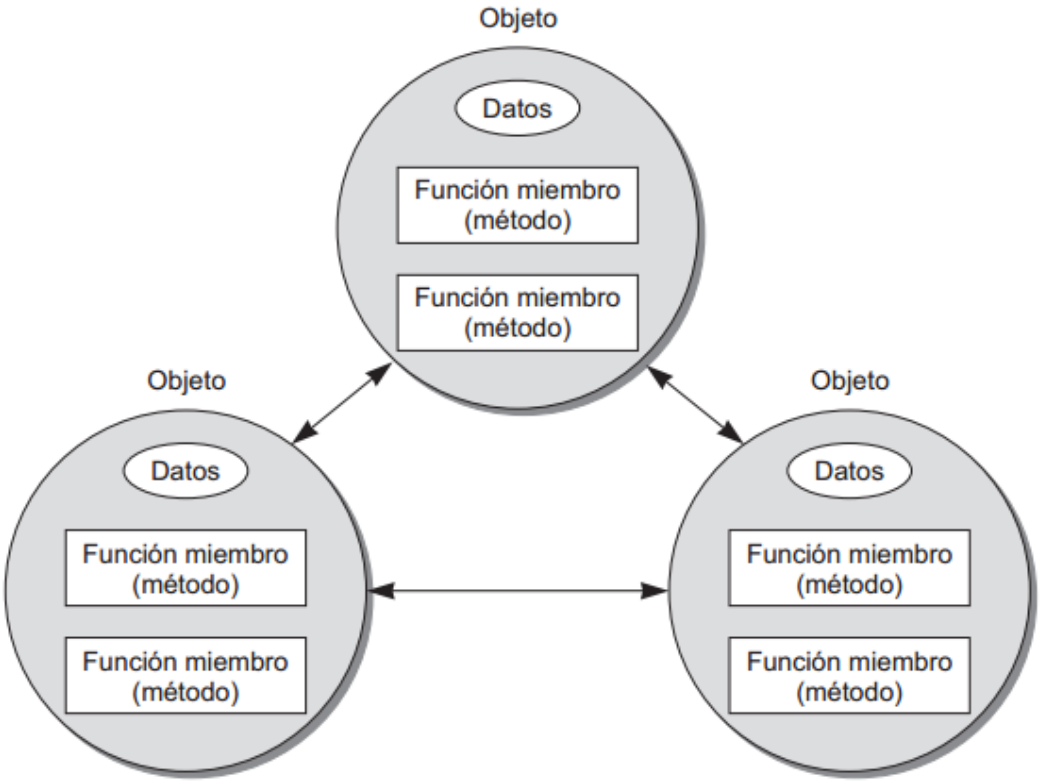

Figura 3: Esquema de la POO, la comunicación entre objetos se da por medio de mesajes.

En el paradigma de programación orientada a objetos, el programa se organiza como un conjunto finito de objetos que contienen datos y operaciones (funciones miembro en C++) que llaman a esos datos y se comunican entre sí mediante mensajes

1.1. Propiedades fundamentales de la orientación a objetos

Existen diversas características ligadas a la orientación a objetos. Todas las propiedades que se suelen considerar, no son exclusivas de este paradigma, ya que pueden existir en otros paradigmas, pero en su conjunto definen claramente los lenguajes orientados a objetos. Estas propiedades son:

- Abstracción (tipos abstractos de datos y clases).

- Encapsulado o encapsulamiento de datos.
- Ocultación de datos.
- Herencia.
- Polimorfismo

C++ soporta todas las características anteriores que definen la orientación a objetos, aunque hay numerosas discusiones en torno a la consideración de C++ como lenguaje orientado a objetos. La razón es que, en contraste con lenguajes tales como Smalltalk, Java o C#, C++ no es un lenguaje orientado a objetos puro. C++ soporta orientación a objetos pero es compatible con C y permite que programas C++ se escriban sin utilizar características orientadas a objetos.

De hecho, C++ es un lenguaje multiparadigma que permite programación estructurada, procedimental, orientada a objetos y genérica.

1.1.1. Abstracción

La abstracción es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta los restantes aspectos. El término abstracción, que se suele utilizar en programación, se refiere al hecho de diferenciar entre las propiedades externas de una entidad y los detalles de la composición interna de dicha entidad. Es la abstracción la propiedad que permite ignorar los detalles internos de un dispositivo complejo tal como una computadora, un automóvil, una lavadora o un horno de microondas, etc., y usarlo como una única unidad comprensible. Mediante la abstracción se diseñan y fabrican estos sistemas complejos en primer lugar y, posteriormente, los componentes más pequeños de los cuales están compuestos. Cada componente representa un nivel de abstracción en el cual el uso del componente se aísla de los detalles de la composición interna del componente. La abstracción posee diversos grados denominados niveles de abstracción. En consecuencia, la abstracción posee diversos grados de complejidad que se denominan niveles de abstracción que ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. En el modelado orientado a objetos de un sistema esto significa centrarse en qué es y qué hace un objeto y no en cómo debe implementarse. Durante el proceso de abstracción es cuando se decide qué características y comportamiento debe tener el modelo.

Aplicando la abstracción se es capaz de construir, analizar y gestionar sistemas de computadoras complejos y grandes que no se podrían diseñar si se tratara de modelar a un nivel detallado. En cada nivel de abstracción se visualiza el sistema en términos de componentes, denominados herramientas abstractas, cuya composición interna se ignora. Esto nos permite concentrarnos en cómo cada componente interactúa con otros componentes y centrarnos en la parte del sistema que es más relevante para la tarea a realizar en lugar de perderse a nivel de detalles menos significativos. En estructuras o registros, las propiedades individuales de los objetos se pueden almacenar en los miembros. Para los objetos es de interés cómo están organizados sino también qué se puede hacer con ellos. Es decir, las operaciones que forman la composición interna de un objeto son también importantes. El primer concepto en el mundo de la orientación a objetos nació con los tipos abstractos de datos (TAD). Un tipo abstracto de datos describe no sólo los atributos de un objeto, sino también su comportamiento (las operaciones). Esto puede incluir también una descripción de los estados que puede alcanzar un objeto. Un medio de reducir la complejidad es la abstracción. Las características y los procesos se reducen a las propiedades esenciales, son resumidas o combinadas entre sí. De este modo, las características complejas se hacen más manejables.

Ejemplo. Diferentes modos de abstracción de un auto.

- Un auto es la combinación (o composición) de diferentes partes, tales como motor, carrocería, cuatro ruedas, cinco puertas, etc.
- Un auto es un concepto común para diferentes tipos de coches. Pueden clasificarse por el nombre del fabricante (Audi, BMW, Seat, Toyota, Chrisler...), por su categoría (turismo, deportivo, todoterreno...), por el carburante que utilizan (gasolina, gasoil, gas, híbrido...).

La abstracción coche se utilizará siempre que la marca, la categoría o el carburante no sean significativos. Así, un auto se utilizará para transportar personas.

1.1.2. Encapsulación y ocultación de datos

El encapsulado o encapsulación de datos es el proceso de agrupar datos y operaciones relacionadas bajo la misma unidad de programación. En el caso de los objetos que poseen las mismas características y comportamiento se agrupan en clases, que no son más que unidades o módulos de programación que encapsulan datos y operaciones.
La ocultación de datos permite separar el aspecto de un componente, definido por su interfaz con el exterior, de sus detalles internos de implementación. Los términos ocultación de la información (information hiding) y encapsulación de datos (data encapsulation) se suelen utilizar como sinónimos, pero no siempre es así, a veces se utilizan en contextos diferentes. En C++ no es lo mismo, los datos internos están protegidos del exterior y no se pueden acceder a ellos más que desde su propio interior y, por tanto, no están ocultos. El acceso al objeto está restringido sólo a través de una interfaz bien definida.
El diseño de un programa orientado a objetos contiene, al menos, los siguientes pasos:

1. Identificar los objetos del sistema.
2. Agrupar en clases a todos los objetos que tengan características y comportamiento comunes.
3. Identificar los datos y operaciones de cada una de las clases.
4. Identificar las relaciones que pueden existir entre las clases.

En C++, un objeto es un elemento individual con su propia identidad; por ejemplo, un libro, un automóvil... Una clase puede describir las propiedades genéricas de un ejecutivo de una empresa (nombre, título, salario, cargo...) mientras que un objeto representará a un ejecutivo específico (Luis M, director general). En general, una clase define qué datos se utilizan para representar un objeto y las operaciones que se pueden ejecutar sobre esos datos.
Cada clase tiene sus propias características y comportamiento; en general, una clase define los datos que se utilizan y las operaciones que se pueden ejecutar sobre esos datos. Una clase describe un conjunto de objetos. En el sentido estricto de programación, una clase es un tipo de datos. Diferentes variables se pueden crear de este tipo. En programación orientada a objetos, éstas se llaman *instancias*. Las *instancias* son, por consiguiente, la realización de los objetos descritos en una clase. Estas instancias constan de datos o atributos descritos en la clase y se pueden manipular con las operaciones definidas dentro de ellas.

Los términos objeto e instancia se utilizan frecuentemente como sinónimos (especialmente en C++). Si una variable de tipo Automovil se declara, se crea un objeto Automovil (una instancia de la clase Automovil)

Las operaciones definidas en los objetos se llaman *métodos*. Cada operación llamada por un objeto se interpreta como un mensaje al objeto, que utiliza un método específico para procesar la operación. En el diseño de programas orientados a objetos se realiza en primer lugar el diseño de las clases que representan con precisión aquellas cosas que trata el programa. Por ejemplo, un programa de dibujo, puede definir clases que representan rectángulos, líneas, pinceles, colores, etcétera. Las definiciones de clases, incluyen una descripción de operaciones permisibles para cada clase, tales como desplazamiento de un círculo o rotación de una línea.

El diseño de clases fiables y útiles puede ser una tarea difícil. Afortunadamente, los lenguajes POO facilitan la tarea ya que incorporan clases existentes en su propia programación. Los fabricantes de software proporcionan numerosas bibliotecas de clases, incluyendo bibliotecas de clases diseñadas para simplificar la creación de programas para entornos tales como Windows, Linux, Macintosh, Unix o Android. Uno de los beneficios reales de C++ es que permite la reutilización y adaptación de códigos existentes y ya bien probados y depurados.

1.2. Objetos

El objeto es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y juega un rol o papel. Si se programa con enfoque orientado a objetos, se intenta descubrir e implementar los objetos que juegan un rol en el dominio del problema y en consecuencia del programa. La estructura interna y el comportamiento de un objeto, en una primera fase, no tiene prioridad.
Dependiendo del problema, diferentes aspectos de un dominio son relevantes. Un auto puede ser ensamblado por partes tales como un motor, una carrocería, unas puertas o puede ser descrito utilizando propiedades tales como su velocidad, su kilometraje o su fabricante. Estos atributos indican el objeto. De modo similar una persona, también se puede ver como un objeto, del cual se disponen diferentes atributos. Dependiendo de la definición del problema, esos atributos pueden ser el nombre, apellido, dirección, número de teléfono, color del cabello, altura, peso, profesión, etc. Un objeto no necesariamente ha de realizar algo concreto o tangible. Puede ser totalmente abstracto y también puede describir un proceso. Por ejemplo, un partido de fútbol o de rugby puede ser descrito como un objeto. Los atributos de este objeto pueden ser los jugadores, el entrenador, la puntuación y el tiempo transcurrido de partido.
Cuando se trata de resolver un problema con orientación a objetos, dicho problema no se descompone en funciones como en programación estructurada tradicional, caso de C, sino en objetos. El pensar en términos de objetos tiene una gran ventaja: se asocian los objetos del problema a los objetos del mundo real.

¿Qué tipos de cosas son objetos en los programas orientados a objetos? La respuesta está limitada por su imaginación aunque se pueden agrupar en categorías típicas que facilitarán su búsqueda en la definición del problema de un modo más rápido y sencillo.

- Recursos Humanos:
  - Empleados.
  - Estudiantes.
  - Clientes.
  - Vendedores.
  - Socios.
- Colecciones de datos:
  - Arrays (arreglos).
  - Listas.
  - Pilas.
  - Árboles.
  - Árboles binarios.
  - Grafos.
- Tipos de datos definidos por usuarios:
  - Hora.
  - Números complejos.
  - Puntos del plano.
  - Puntos del espacio.
  - Ángulos.
  - Lados.
- Elementos de computadoras:
  - Menús.
  - Ventanas.

    - Objetos gráficos (rectángulos, círculos, rectas, puntos...).
    - Ratón (mouse).
    - Teclado.
    - Impresora.
    - USB.
    - Tarjetas de memoria de cámaras fotográficas.
- Objetos físicos:
    - Autos.
    - Aviones.
    - Trenes.
    - Barcos.
    - Motocicletas.
    - Casas.
- Componentes de videojuegos:
    - Consola.
    - Mandos.
    - Volante.
    - Conectores.
    - Memoria.
    - Acceso a Internet.

La correspondencia entre objetos de programación y objetos del mundo real es el resultado eficiente de combinar datos y funciones que manipulan esos datos. Los objetos resultantes ofrecen una mejor solución al diseño del programa que en el caso de los lenguajes orientados a procedimientos.

Un **objeto** se puede definir desde el punto de vista conceptual como una entidad individual de un sistema y que se caracteriza por un estado y un comportamiento. Desde el punto de vista de implementación, un objeto es una entidad que posee un conjunto de datos y un conjunto de operaciones (funciones o métodos).

El estado de un objeto viene determinado por los valores que toman sus datos, cuyos valores pueden tener las restricciones impuestas en la definición del problema. Los datos se denominan también *atributos* y componen la estructura del objeto y las operaciones — también llamadas *métodos* — representan los servicios que proporciona el objeto.

En la POO se utiliza el *Lenguaje unificado de modelalo UML*. **UML** se ha convertido en el estándar para modelado de aplicaciones de software y es un lenguaje con sintaxis propia , se compone de pseudocódigo, código real, programas, etc. [1] La representación gráfica de un objeto en UML se muestra en la Figura 4.

2. Clases

Una clase es la descripción de un conjunto de **objetos**; consta de **métodos** (o **funciones miembro**) y **datos** o **atributos** que resumen características comunes de un conjunto de objetos. Generalmente,una clase se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por un estado específico y es capaz de realizar una serie de operaciones.

[1] Programación en C, C++, Java y UML - Luis Joyanes Aguilar, Ignacio Zahonero Martínez

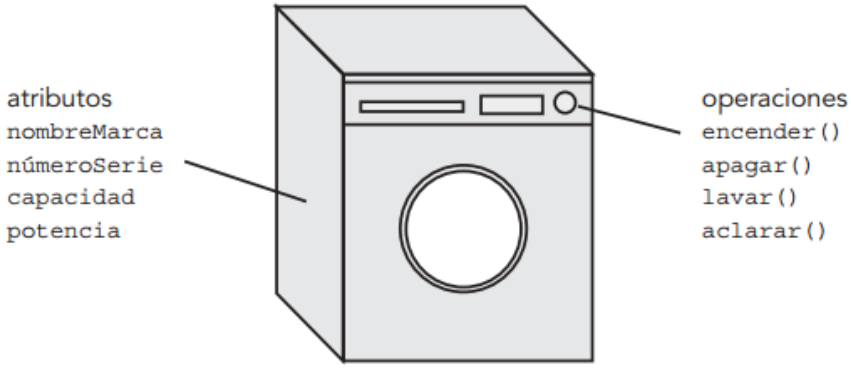

Figura 4: Un lavarropa modelado como un objeto en UML

Una clase es, en esencia, la declaración de un tipo objeto. Las clases son similares a los tipos de datos y equivalen a modelos o plantillas que describen cómo se construyen ciertos tipos de objetos. Definir una clase significa dar a la misma un nombre, así como dar nombre a los elementos que almacenan sus datos y describir las funciones que realizarán las acciones consideradas en los objetos.

Una declaración de una clase consta de una palabra reservada *class* y el nombre de la clase. Se utilizan tres diferentes especificadores de acceso para controlar el acceso a los miembros de la clase. Estos accesos son: *public*, *private* y *protected*.

La sintaxis de una clase es

```
class nombre_clase {
public:
// miembros públicos
protected:
// miembros protegidos
private:
// miembros privados
};
```

La etiqueta *public* define **miembros públicos**, que son aquéllos a los que se puede acceder por cualquier función. A los **miembros privados** que siguen a la etiqueta *private* sólo se puede acceder por funciones miembro de la misma clase o por **funciones y clases amigas**. A los miembros que siguen al especificador *protected* se puede acceder por funciones miembro de la misma clase o de **clases derivadas** de la misma, así como por sus amigas. Los especificadores *public, protected y private* pueden aparecer en cualquier orden. Si se omite el especificador de acceso, el acceso por defecto es privado. En la Tabla 1 cada "x" indica que el acceso está permitido al tipo del miembro de la clase listado en la columna de la izquierda.

Tipo de miembro	**Miembro de la misma clase**	**Amiga**	**Miembro de una clase derivada**	**Función no miembro**
private	X	X		
protected	X	X	X	
publica	X	X	X	X

Cuadro 1: Tabla que muestra la visibilidad de cada miembro de la clase

El uso de los especificadores de acceso es implementar la ocultación de la información. El principio de ocultación de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida para permitir que los detalles de implementación de los objetos sean ignorados. El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

Las *funciones miembro* y los *miembros datos* de la sección pública forman la interfaz externa del objeto, mientras que los elementos de la sección privada son los aspectos internos del objeto que no necesitan ser

accesibles para usar el objeto. La sección protegida necesita para su comprensión el concepto de herencia que se explicará posteriormente.

Las funciones miembro (o métodos) pueden ser de los siguientes tipos:

- **Constructores y destructores:** funciones miembro a las que se llama automáticamente cuando un operador se crea o se destruye.
- **Selectores:** devuelven los valores de los miembros dato.
- **Modificadores o mutadores:** permiten a un programa cliente cambiar los contenidos de los miembros dato.
- **Operadores:** permiten definir operadores estándar C++ para los objetos de las clases.
- **Iteradores:** procesan colecciones de objetos, tales como arrays y listas.

Ejemplo. Declaración de una clase *Punto* que define un punto en el espacio tridimensional mediante sus coordenadas *(x, y, z)*

```
class Punto // nombre de la clase
{
   public: // zona pública con declaración de funciones miembro
       float ObtenerX() { return x; } //selector x
       float ObtenerY() { return y;} //selector y
       float ObtenerZ() { return z;} //selector z
       void PonerX ( float val_x) { x = val_x; } //modificador x
       void PonerY ( float val_y) { y = val_y; } //modificador y
       void PonerZ ( float val_z) { z = val_z;} //modificador z

   private: // zona privada con declaración de atribitos.
       float x, y, z;
};
```

En la zona pública se declaran las funciones que se van a utilizar para acceder a los datos que se encuentran en la zona privada. Estas funciones reciben el nombre de **getters** (por get en inglés, obtener) y **setters**, por set en inglés colocar, instalar.

Las funciones ObtenerX(), ObtenerZ(), ObtenerZ(); son *getters* que permiten leer la información de los datos que se encuentran en la parte privada.

Las funciones PonerX(), PonerZ(), PonerZ(); son *setters* que permiten escribir la información de los datos que se encuentran en la parte privada.

3. Objetos

Una **clase** es la declaración de un tipo **objeto**, y una variable de tipo clase es un objeto. La clase es la declaración que construirá cada objeto, por ejemplo si creásemos una clase *silla*, los objetos sería *silla de caño, silla de madera, silla de escuela*, etc. La clase brinda la plataforma sobre la cual se cargan los métodos y atributos para cada objeto. Cada vez que se construye un objeto a partir de una clase se crea una instancia (ejemplar) de esa clase. Por consiguiente, los objetos no son más que instancias de una clase.

En general, instancia de una clase y objeto son términos intercambiables. Los tipos objetos definidos por el usuario se comportan como tipos incorporados que tienen datos internos y operaciones internas y externas. La sintaxis de declaración de un objeto es:

```
nombre_clase identificador;
```

Así, la definición de un objeto que me dará las coordenadas de un punto en el espacio es: *Punto p;* El operador de acceso a un miembro (.) selecciona un miembro individual de un objeto de la clase.

Ejemplo. Crear un punto p, fijar su coordenada *x* y visualizar dicha coordenada.

```
Punto p;
p.PonerX (100);  //pone la coordenada x a 100
cout << " coordenada x es " << p.ObtenerX();
```

Cada objeto consta de:

- **Estado** (atributos) determinados por sus datos internos. El estado de un objeto es simplemente el conjunto de valores de todas las variables contenidas dentro del objeto en un instante dado. Un atributo consta de dos partes: un nombre de atributo y un valor de atributo.
- **Métodos, funciones miembro, operaciones o comportamiento** (métodos invocados por mensajes). Estas operaciones se dividen en tres grandes grupos: operaciones que manipulan los datos de alguna forma específica (añadir, borrar, cambiar formato...); operaciones que realizan un cálculo o proceso; y operaciones que comprueban (monitorizan) un objeto frente a la ocurrencia de algún suceso de control. Cada método tiene un nombre y un cuerpo que realiza la acción o comportamiento asociado con el nombre del método.

Los objetos ocupan espacio en memoria y, en consecuencia, existen en el tiempo, y deberán *crearse* o *instanciarse*. Por la misma razón, se debe liberar el espacio en memoria ocupado por los objetos.

Un *mensaje* es la acción que realiza un objeto. El conjunto de mensajes a los cuales puede responder un objeto se denomina *protocolo del objeto*. Un mensaje consta de tres partes:

- Identidad del receptor.
- El método que se ha de ejecutar.
- Información especial necesaria para realizar el método invocado (argumentos o parámetros requeridos).

Dos operaciones comunes típicas en cualquier objeto son:

1. **Constructor:** una operación que crea un objeto y/o inicializa su estado.
2. **Destructor:** una operación que libera el estado de un objeto y/o destruye el propio objeto.

Estas operaciones se ejecutarán implícitamente por el compilador o explícitamente por el programador, mediante invocación a los citados constructores. Los objetos tienen las siguientes características:

- Se agrupan en tipos llamados clases.
- Tienen datos internos que definen su estado actual.
- Soportan ocultación de datos.
- Pueden heredar propiedades de otros objetos.
- Pueden comunicarse con otros objetos pasando mensajes.
- Tienen métodos que definen su comportamiento.

Ejemplo. Escriba un programa en C++ que utilice una clase que defina un punto en el espacio.

```
#include <cstdlib>
#include <iostream>
using namespace std;
class Punto // nombre de la clase
{
    public: // zona pública con declaración de funciones miembro
        float ObtenerX() { return x; } //selector x
        float ObtenerY() { return y;} //selector y
        float ObtenerZ() { return z;} //selector z
        void PonerX ( float valx) { x = valx; } //modificador x
        void PonerY ( float valy) { y = valy; } //modificador y
        void PonerZ ( float valz) { z = valz;} //modificador z
        Punto( float valx): x(valx), y(0.0), z(0.0){} //constructor
        // constructor que inicializa miembros
        //Las declaraciones anteriores son definiciones de métodos, ya que
        //incluyen el cuerpo de cada función. Son funciones en línea, inline
        Punto(); //constructor por defecto
        Punto(float , float, float); //constructor alternativo
        Punto(const Punto &p); // constructor de copia
        ~Punto(){};

        void EscribirPunto(); // selector
        void AsignarPunto( float , float, float); // modificador
        //Las declaraciones anteriores no incluyen el cuerpo de cada función
        //Son funciones fuera de línea. El cuerpo de la función se declara
        //independientemente
    private: // zona privada con declaración de atribitos.
        float x, y, z;
};
Punto::Punto()
{ //cuerpo constructor por defecto
    x = 0;
    y = 0;
    z = 0;
}

Punto::Punto( float valx, float valy, float valz)
{ //cuerpo constructor alternativo
    x = valx;
    y = valy;
    z = valz;
}

Punto::Punto ( const Punto &p)
{ // cuerpo del constructor de copia
    x = p.x;
    y = p.y;
    z = p.z;
}

void Punto::EscribirPunto()
{ // cuerpo
```

```
    cout << " " << ObtenerX() << " " << ObtenerY()
    << " " << ObtenerZ() << endl;
}

void Punto::AsignarPunto( float valx, float valy, float valz)
{ //Cuerpo. El nombre de los parámetros puede omitirse en declaración
    x = valx;
    y = valy;
    z = valz;
}

int main( int argc, char *argv[])
{
    Punto p, p1; // p, y p1 son objetos de la clase Punto
    //p.EscribirPunto();
    //p1.EscribirPunto();
    Punto p2 = Punto(5.0, 6.0, 7.0); //p2 objeto de la clase Punto
    Punto p3(8.0); //p3 objeto de la clase Punto
    //Punto* p4;
    p.AsignarPunto(2.0,3.0,4.0); // llamada a función miembro;
    p.EscribirPunto(); // llamada a función miembro
    //cin.get();
    p1.EscribirPunto(); //Escribe valores constructor por defecto
    p2.EscribirPunto(); //Escribe valores constructor alternativo
    p3.EscribirPunto(); //Escribe valores constructor lista
    //p4-> EscribirPunto();
    cin.get();
    return EXIT_SUCCESS;
}
```

Los métodos *ObtenerX, ObtenerY, ObtenerZ, PonerX, PonerY, PonerZ*, son funciones miembro en línea, y se declaran en una sola línea.

Los métodos *Punto, EscribirPunto*, y *AsignarPunto*, tienen su prototipo en la declaración de la clase, pero la declaración de su cuerpo se realiza fuera de línea. La declaración de su prototipo termina en punto y coma ; .

Los cuerpos de las funciones *Punto, EscribirPunto*, y *AsignarPunto* se realizan fuera de línea. Estas declaraciones van precedidas del operador de resolución de ámbito cuatro puntos (::). Las funciones miembro de una clase se definen de igual modo que cualquier otra función excepto que se necesita incluir el operador de resolución de ámbito :: en la definición de la función (en su cabecera).

Formato del cuerpo de funciones que no son declaradas en línea

```
Tipo_devuelto Nombre_clase ::Nombre_función (Lista_parámetros)
{
sentencias del cuerpo de la función
}
```

El símbolo :: (operador de resolución de ámbito) se utiliza en sentencias de ejecución para acceder a los miembros de la clase. Por ejemplo, la expresión Punto::X se refiere al miembro dato X de la clase Punto.

- Las llamadas a las funciones miembro de los objetos p, p1, p2, p3 fuera de la clase (en el programa principal) tienen el formato *p.funcionmiembro*.
- Las llamadas a las funciones miembro de una clase dentro de la propia clase no van precedidas de punto, como se hace por ejemplo en el método *EscribirPunto*.

- Hay tres tipos de constructores que se explican posteriormente.
- Una clase de C++ es una generalización de una estructura de C. Una clase que sólo tiene miembros públicos y sin funciones es una estructura.
- Cada clase contiene un puntero implícito **this** que apunta a sí misma. El puntero **this** es la dirección de la instancia de una clase. ***this** es el objeto real.

4. Constructores

Un constructor es una función miembro que se ejecuta automáticamente cuando se crea el objeto de una clase. Los constructores tienen siempre el mismo nombre que la propia clase. Cuando se define un constructor no se puede especificar un valor de retorno, ni incluso void. Un constructor puede, sin embargo, tomar cualquier número de parámetros (cero o más).

Constructor por defecto. Un constructor que no tiene parámetros se llama constructor por defecto. Normalmente, inicializa los miembros dato asignándoles valores por defecto. Así, por ejemplo, en el caso de la declaración de la clase Punto, el constructor Punto() , es llamado en el momento de la declaración de cada objeto, y ejecutándose en el mismo momento sus sentencia de inicialización:

```
Punto p1; // p1.x = 0 p1.y = 0 p1.z = 0
```

C++ crea automáticamente un constructor por defecto cuando no existen otros constructores. Sin embargo, tal constructor no inicializa los miembros dato de la clase a un valor previsible, de modo que siempre es conveniente al crear su propio constructor por defecto, darle la opción de inicializar los miembros dato con valores previsibles.

Precaución: Tenga cuidado con la escritura de la siguiente sentencia: *Punto p();*
Aunque parece que se realiza una llamada al constructor por defecto lo que se hace es declarar una función de nombre p que no tiene parámetros y devuelve un resultado de tipo Punto

Constructores alternativos. Es posible pasar argumentos a un constructor asignando valores específicos a cada miembro dato de los objetos de la clase. Un constructor con parámetros se denomina constructor alternativo. Estos constructores son llamados en sentencias del tipo

```
Punto p2 = Punto( 5.0, 6.0, 7.0); // p2.x = 5.0 p2.y = 6.0 p3.z = 7.0
```

Constructor de copia. Existe un tipo especializado de constructor denominado constructor de copia, que se crea automáticamente por el compilador. El constructor de copia se llama automáticamente cuando un objeto se pasa por valor: se construye una copia local del objeto que se construye. El constructor de copia se llama también cuando un objeto se declara e inicializa con otro objeto del mismo tipo. Por ejemplo el código siguiente asigna a los atributos del objeto actual los atributos del objeto p.

```
Punto::Punto ( const Punto &p)
{
x = p.x;
y = p.y;
z = p.z;
}
```

Constructor lista inicializadora. Inicialización de miembros. No está permitido inicializar un miembro dato de una clase cuando se define. La inicialización de los miembros dato de una clase se realiza en el constructor especial lista inicializadora de miembros que permite inicializar (en lugar de asignar) a uno o más miembros dato. Una lista inicializadora de miembros se sitúa inmediatamente después de la lista de parámetros en la definición del constructor. Consta de un carácter dos puntos, seguido por uno o más inicializadores de miembro, separados por comas. Un inicializador de miembros consta del nombre de un miembro dato seguido por un valor inicial entre paréntesis. Por ejemplo las declaración de constructor siguiente, inicializa *x* a *valx*, *y* a *0.0* así como *z* a *0.0*.

```
Punto( float valx): x( valx), y(0.0), z(0.0){} // constructor
```

Sobrecarga de funciones miembro. Al igual que se puede sobrecargar una función global, se puede también sobrecargar el constructor de la clase o cualquier otra función miembro de una clase excepto el destructor (posteriormente se describirá el concepto de destructor, pero no se puede sobrecargar). De hecho los constructores sobrecargados son bastante frecuentes; proporcionan medios alternativos para inicializar objetos nuevos de una clase. Sólo un constructor se ejecuta cuando se crea un objeto, con independencia de cuántos constructores hayan sido definidos.

5. Destructor

En una clase se puede definir también una función miembro especial conocida como **destructor**, que se llama automáticamente siempre que se destruye un objeto de la clase. El nombre del destructor es el mismo que el nombre de la clase, precedida con el carácter . Al igual que un constructor, un destructor se debe definir sin ningún tipo de retorno (ni incluso void); al contrario que un constructor no puede aceptar parámetros, y además cada clase tiene sólo un destructor. El uso más frecuente de un destructor es liberar memoria que fue asignada por el constructor . Si un destructor no se declara explícitamente, C++ crea un vacío automáticamente.

El destructor de la clase Punto es

```
~Punto();
```

Ejemplo. Clase pila de reales con funciones miembro para poner un elemento y sacar un elemento de la pila.

Una pila es un tipo de dato ”en el cual el último en entrar es el primero en salir”. Las primitivas (operaciones) básicas tradicionales de gestión de una pila son poner y sacar. La primera añade un elemento a la pila y la segunda extrae un elemento borrándolo. En el ejemplo siguiente, la pila se implementa en un array a de longitud máxima de 100 reales.
La zona privada de la clase pila contiene: la cima que apunta a la siguiente posición libre de la pila, en consecuencia la pila estará vacía cuando cima apunte a cero, y estará llena si la cima apunta a un número mayor o igual que 100; y el array a de 100 reales. La zona pública contiene el constructor, el destructor y las dos funciones miembro de gestión básica. Para poner un elemento en la pila, se coloca en la posición a[cima] y se incrementa cima. Para sacar un elemento de la pila, se decrementa cima en una unidad y se retorna el valor de a[cima].
Las pilas son muy utilizadas en las comunicaciones serie de datos entre microcontroladores.

```
#include <cstdlib>
#include <iostream>
#include <string.h>
using namespace std;
#define max 100
class pila
{
   private:
        int cima;
        float a[max];
   public:
        pila(){cima = 0;};
        float sacar();
        void poner( float x);
        ~pila(){};
};
```

```
float pila::sacar()
{
   if(cima <= 0)
      cout << " error pila vacía ";
   else
      return a[--cima];
}

void pila::poner( float x)
{
   if(cima >= max)
      cout << " error pila llena";
   else
      a[cima++]= x;
}

int main( int argc, char *argv[])
{
   pila p;
   p.poner(6.0);
   p.poner(5.0);
   cout << p.sacar()<< endl;
   cout << p.sacar()<< endl;
   cin.get();
return EXIT_SUCCESS;
}
```

Ejemplo. Realizar una clase Complejo que permita la gestión de números complejos (un número complejo = dos números reales). Las operaciones a implementar son las siguientes:

a Una función $leerComplejo()$ permite leer un objeto de tipo $Complejo$.

b Una función $escribirComplejo()$ realiza la visualización formateada de un Complejo.

c Suma: $a+c=(A+C,(B+D)i)$.

d Resta: $a–c=(A–C,(B–D)i)$.

e Multiplicación: $a*c=(A*C–B*D,(A*D+B*C)i)$.

f Multiplicación: $x*c=(x*C,x*Di)$, donde x es real.

g Conjugado: $a=(A.–Bi)$.

Solucion 1.

```
#include<iostream>
#include<cstdlib>

using namespace std;

class Complejo
{
    public:
        float Or(){return r;} // getter
        float Oi(){return i;}
```

```
        void Pr(float r1){ r = r1;} // setter
        void Pi(float i1){i = i1;}
        Complejo(){ r = 0.0; i = 0.0;} ; // constructor por defecto
        Complejo(float r1, float i1){ r = r1; i = i1;} // constructor
        ~Complejo(){};  // destructor
        void leerComplejo(); // funcion miembro
        void escribirComplejo(); // funcion miembro
        void suma( Complejo a, Complejo b);
        void resta( Complejo a, Complejo b);
        void multiplicacion( Complejo a, Complejo b);
        void multiplicacion( float x, Complejo a);
        void conjugado (Complejo a);

    private:
        float r, i;
};

void Complejo::leerComplejo()
{
    float r, i;  // variable local
    cout << " parte real  :";
    cin >> r;
    cout  << " parte imaginaria :";
    cin >> i;
    Pr(r);
    Pi(i);
}

void Complejo::escribirComplejo()
{
    cout << " real = " << Or() << " imaginaria = " << Oi()<< endl;
}

void Complejo::suma (Complejo a, Complejo b)
{
    //suma del número complejo a  y el complejo b
    Pr(a.Or() + b.Or());
    Pi(a.Oi() + b.Oi());
}

void Complejo::resta (Complejo a, Complejo b)
{
    //suma del número complejo a y el complejo b
    Pr(a.Or() - b.Or());
    Pi(a.Oi() - b.Oi());
}

void Complejo:: multiplicacion(Complejo a, Complejo b)
{
   //  producto de número complejo a  y el complejo b
    Pr(a.Or() * b.Or() - a.Oi() * b.Oi());
    Pi(a.Or()*b.Oi() + a.Oi() * b.Or());
}
```

```
void Complejo::multiplicacion( float x, Complejo a)
{
    Pr(x * a.Or());
    Pi(x * a.Oi());
}

void Complejo::conjugado( Complejo a)
{
     Pr(-a.Oi());
     Pi(a.Or());
}


int main(int argc, char *argv[])
{
    Complejo j1(2, 3);
    Complejo j2, operacion;
    float x;
    cout << "Los valores por defecto para el numero j1 y j2 son: "<<endl;
    j1.escribirComplejo();
    cout << endl;
    j2.escribirComplejo();
    cout << endl;

    cout << "Inicializacion de j2: "<< endl; j2.leerComplejo(); cout << endl;
    j2.escribirComplejo();
    cout<< "\n\n" << endl;

    cout << "La suma de j1 y j2 es: " << endl; operacion.suma(j1, j2); operacion.escribirCompl
    cout << "\n" << endl;

    cout << "La resta de j1 y j2 es: " << endl; operacion.resta(j1, j2); operacion.escribirCom
    cout << "\n" << endl;

    cout << "La multiplicacion entre j1 y j2 es: " << endl; operacion.multiplicacion(j1, j2);
    cout << "\n" << endl;

    cout << "La multiplicacion de j1 por una constante x (inctroduzca x) " << endl;
    cin>> x; operacion.multiplicacion(x, j1); operacion.escribirComplejo();
    cout << "\n" << endl;

    cout << "La multiplicacion de j2 por una constante x (inctroduzca x) " << endl;
    cin>> x; operacion.multiplicacion(x, j2); operacion.escribirComplejo();
    cout << "\n" << endl;

    cout << "El conjugado de j1 "; operacion.conjugado(j1); operacion.escribirComplejo(); cout
    cout << "El conjugado de j2 "; operacion.conjugado(j2); operacion.escribirComplejo(); cout


    cin.get();
    return EXIT_SUCCESS;
}
```

Solución 2.

```
/*Encabezado complejos.h:  Declaración de clase, constructores,
destructor, y prototipos de función*/
#ifndef COMPLEJOS_H_INCLUDED
#define COMPLEJOS_H_INCLUDED
#include<iostream>
#include<cstdlib>

using namespace std;

class Complejo
{
    public:
        float Or(){return r;} // getter
        float Oi(){return i;}
        void Pr(float r1){ r = r1;} // setter
        void Pi(float i1){i = i1;}
        Complejo(){ r = 0.0; i = 0.0;} ; // constructor por defecto
        Complejo(float r1, float i1){ r = r1; i = i1;} // constructor
        ~Complejo(){};  // destructor
        void leerComplejo(); // funcion miembro
        void escribirComplejo(); // funcion miembro
        void suma( Complejo a, Complejo b);
        void resta( Complejo a, Complejo b);
        void multiplicacion( Complejo a, Complejo b);
        void multiplicacion( float x, Complejo a);
        void conjugado (Complejo a);

    private:
        float r, i;
};


#endif // COMPLEJOS_H_INCLUDED

/*-----------------------------------------------------------------*/

/* Archivo operaciones.cpp incluye la funciones para cargar el numero complejo
e imprimirlo*/
#include<iostream>
#include "complejos.h"
using namespace std;

void Complejo::leerComplejo()
{
    float r, i;  // variable local
    cout << " parte real  :";
    cin >> r;
    cout  << " parte imaginaria :";
    cin >> i;
    Pr(r);
```

```
    Pi(i);
}

void Complejo::escribirComplejo()
{
    cout << " real = " << Or() << " imaginaria = " << Oi()<< endl;
}

/*-----------------------------------------------------------------*/

/* Archivo operaciones.cpp incluye las funciones para
realizar las operaciones matemáticas requeridas*/
#include<iostream>
#include "complejos.h"

void Complejo::suma (Complejo a, Complejo b)
{
    //suma del número complejo a  y el complejo b
    Pr(a.Or() + b.Or());
    Pi(a.Oi() + b.Oi());
}

void Complejo::resta (Complejo a, Complejo b)
{
    //suma del número complejo a y el complejo b
    Pr(a.Or() - b.Or());
    Pi(a.Oi() - b.Oi());
}

void Complejo:: multiplicacion(Complejo a, Complejo b)
{
   //  producto de número complejo a  y el complejo b
    Pr(a.Or() * b.Or() - a.Oi() * b.Oi());
    Pi(a.Or()*b.Oi() + a.Oi() * b.Or());
}

void Complejo::multiplicacion( float x, Complejo a)
{
    Pr(x * a.Or());
    Pi(x * a.Oi());
}

void Complejo::conjugado( Complejo a)
{
     Pr(-a.Oi());
     Pi(a.Or());
}

/*-----------------------------------------------------------------*/
/*Archivo principal, utiliza el objeto Complejo
e invoca las funciones de este*/
#include<iostream>
#include<cstdlib>
```

```
#include "complejos.h"

int main(int argc, char *argv[])
{
    Complejo j1(2, 3);
    Complejo j2, operacion;
    float x;
    cout << "Los valores por defecto para el numero j1 y j2 son: "<<endl;;
    j1.escribirComplejo();
    cout << endl;
    j2.escribirComplejo();
    cout << endl;

    cout << "Inicializacion de j2: "<< endl; j2.leerComplejo(); cout << endl;
    j2.escribirComplejo();
    cout<< "\n\n" << endl;

    cout << "La suma de j1 y j2 es: " << endl; operacion.suma(j1, j2); operacion.escribirCompl
    cout << "\n" << endl;

    cout << "La resta de j1 y j2 es: " << endl; operacion.resta(j1, j2); operacion.escribirCom
    cout << "\n" << endl;

    cout << "La multiplicacion entre j1 y j2 es: " << endl; operacion.multiplicacion(j1, j2);
    cout << "\n" << endl;

    cout << "La multiplicacion de j1 por una constante x (inctroduzca x) " << endl;
    cin>> x; operacion.multiplicacion(x, j1); operacion.escribirComplejo();
    cout << "\n" << endl;

    cout << "La multiplicacion de j2 por una constante x (inctroduzca x) " << endl;
    cin>> x; operacion.multiplicacion(x, j2); operacion.escribirComplejo();
    cout << "\n" << endl;

    cout << "El conjugado de j1 "; operacion.conjugado(j1); operacion.escribirComplejo(); cout
    cout << "El conjugado de j2 "; operacion.conjugado(j2); operacion.escribirComplejo(); cout


    cin.get();
    return EXIT_SUCCESS;
}
```

6. AUTOREFERENCIA DEL OBJETO: this

this es un puntero al objeto que envía un mensaje, o simplemente, un puntero al objeto que llama a una función miembro de la clase (ésta no debe ser static). Este puntero no se define, internamente se define:

```
const NombreClase* this;
```

por consiguiente, no puede modificarse. Las variables y funciones de las clase están referenciados, implícitamente, por *this*. Por ejemplo, la siguiente clase:

```
class Triangulo
{
   private:
      double base, altura;
   public:
      double area() const
      {
         return base*altura /2.0;
      }
};
```

En la función *area()* se hace referencia a las variables instancia base y altura. ¿A la base, altura de qué objeto? El método es común para todos los objetos Triangulo. Aparentemente no distingue entre un objeto y otro, sin embargo, cada variable instancia implícitamente está cualificada por *this*, es como si se hubiera escrito:

```
public double area()
{
 return this -> base * this -> altura/2.0;
}
```

Fundamentalmente *this* tiene dos usos:

- Seleccionar explícitamente un miembro de una clase con el fin de dar mas claridad o de evitar colisión de identificadores. Por ejemplo, en la clase Triangulo:

  ```
  void datosTriangulo(double base, double altura)
  {
   this -> base = base;
   this -> altura = altura;
  }
  ```

  Se ha evitado, con *this*, la colisión entre argumentos y variables instancia

- Que una función miembro devuelva el mismo objeto que le llamó. De esa manera se pueden hacer llamadas en cascada a funciones de la misma clase. De nuevo en la clase Triangulo:

  ```
  const Triangulo& datosTriangulo(double base, double altura)
  {
   this -> base = base;
   this -> altura = altura;
   return *this;
  }
  const Triangulo& visualizar() const
  {
   cout << " Base = " << base << endl;
   cout << " Altura = " << altura << endl;
   return *this;
  }
  ```

Ahora se pueden realizar esta concatenación de llamadas:

```
Triangulo t;
t.datosTriangulo(15.0, 12.0).visualizar();
```

7. Clase compuesta

Una *clase compuesta* es aquella que contiene miembros dato que son asimismo **objetos** de clases. Antes de crear el cuerpo de un constructor de una clase compuesta, se deben construir los miembros dato individuales en su orden de declaración. La clase Estudiante contiene miembros dato de tipo Expediente y Dirección:

```
class Expediente
{
   public:
      Expediente(); // constructor por defecto
      Expediente(int idt);
      // ...
};
class Direccion
{
   public:
      Direccion(); // constructor por defecto
      Direccion(string d);
      // ...
};

class Estudiante
{
   public:
   Estudiante()
   {
      PonerId(0);
      PonerNotaMedia(0.0);
   }

   void PonerId(long);
   void PonerNotaMedia(float);
   private:
      long id;
      Expediente exp;
      Direccion dir;
      float NotMedia;
};
```

Aunque *Estudiante* contiene *Expediente* y *Dirección*, el constructor de *Estudiante* no tiene acceso a los miembros privados o protegidos de *Expediente* o *Dirección*. Cuando un objeto *Estudiante* sale fuera de alcance, se llama a su destructor. El cuerpo de **Estudiante()** se ejecuta antes que los destructores de *Expediente* y *Dirección*. En otras palabras, el orden de las llamadas a destructores a clases compuestas debe hacerse exactamente en la forma opuesta al orden de llamadas de constructores. La llamada al constructor con argumentos de los miembros de una clase compuesta se hace desde el constructor de la clase compuesta. Por ejemplo, este constructor de *Estudiante* inicializa su expediente y dirección:

```
Estudiante::Estudiante(int expediente, string direccion)
:exp(expediente), dir(direccion)  // lista de inicialización
{
 PonerId(0);
 PonerNotaMedia(0.0);
}
```

El orden de creación de un objeto compuesto es:

- En primer lugar los objetos miembros en orden de aparición
- A continuación el cuerpo del constructor de la clase compuesta

La llamada al constructor de los objetos miembros se realiza en la lista de inicialización del constructor de la clase compuesta, de la forma:

```
Compuesta(arg1, arg2, arg3,...):miembro1(arg1, ...), miembro2(arg2, ..)
{
   // cuerpo del consructor de clase compuesta
}
```

8. MIEMBROS STATIC DE UNA CLASE

Cada instancia de una clase, cada objeto, tiene su propia copia de las variables de la clase. Cuando interese que haya miembros que no estén ligados a los objetos sino a la clase y, por tanto, comunes a todos los objetos, éstos se declaran static.

Las variables de clase static son compartidas por todos los objetos de la clase. Se declaran de igual manera que otra variable, añadiendo, como prefijo, la palabra reservada static. Por ejemplo:

```
class Conjunto
{
 static int k;
 static Lista lista;
 // ...
}
```

Los miembros static de una clase deben ser inicializados explícitamente fuera del cuerpo de la clase. Así, los miembros k y lista:

```
int Conjunto::k = 0;
Lista Conjunto::lista = NULL;
```

Dentro de las clases se accede a los miembros static de la manera habitual, simplemente con su nombre. Desde fuera de la clase se accede con el nombre de la clase, el selector y el nombre de la variable, por ejemplo:

```
cout << " valor de k = " << Conjunto.k;
```

> *El símbolo :: (operador de resolución de ámbitos) se utiliza en sentencias de ejecución que accede a los miembros estáticos de la clase. Por ejemplo la expresión Punto::X se refiere al miembro estático X de la clase Punto*

Ejemplo. Dada una clase se quiere conocer en todo momento los objetos activos en la aplicación.
Se declara la clase Ejemplo con dos constructores y el constructor de copia. Todos incrementan la variable *static cuenta*, en 1. De esa manera cada nuevo objeto queda contabilizado. También se declara el destructor para decrementar *cuenta* en 1. main() crea objetos y visualiza la variable que contabiliza el número de sus objetos.

```
// archivo Ejemplo.h
class Ejemplo
{
   private:
      int datos;
```

```
   public:
      static int cuenta;
      Ejemplo();
      Ejemplo(int g);
      Ejemplo(const Ejemplo&);
      ~Ejemplo();
};

// definición de la clase, archivo Ejemplo.cpp
#include "Ejemplo.h"

int Ejemplo::cuenta = 0;

Ejemplo::Ejemplo()
{
   datos = 0;
   cuenta++; // nuevo objeto
}

Ejemplo::Ejemplo(int g)
{
   datos = g;
   cuenta++; // nuevo objeto
}

Ejemplo::Ejemplo(const Ejemplo& org)
{
   datos = org.datos;
   cuenta++; // nuevo objeto
}


Ejemplo::~Ejemplo()
{
   cuenta--;
}

// programa de prueba, archivo Demostatic.cpp
#include <iostream>
using namespace std;
#include "Ejemplo.h"
int main()
{
   Ejemplo d1, d2;
   cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
   if (true) // d3 solo existe en el alcance del if
   {
      Ejemplo d3(88);
      cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
   }
   cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
   Ejemplo* pe;
   pe = new Ejemplo();
```

```
   cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
   delete pe;
   cout << "Objetos Ejemplo: " << Ejemplo::cuenta << endl;
   return EXIT_SUCCESS;
}
```

8.1. Funciones miembro static

Los métodos o funciones miembro de las clases se llaman a través de los objetos. En ocasiones, interesa definir funciones que estén controlados por la clase, incluso que no haga falta crear un objeto para llamarlos, son las funciones miembro static. La llamada a estas funciones de clase se realiza a través de la clase: **NombreClase::metodo()**, respetando las reglas de visibilidad. También se pueden llamar con un objeto de la clase, no es recomendable debido a que son métodos dependientes de la clase y no de los objetos.
Los métodos definidos como static no tienen asignado la referencia this, por eso sólo pueden acceder a miembros static de la clase. Es un error que una función miembro static acceda a miembros de la clase no static.

Ejemplo. La clase **SumaSerie** define tres variables static, y un método **static** que calcula la suma cada vez que se llama.

```
class SumaSerie
{
   private:
      static long n;
      static long m;
   public:
   static long suma()
   {
      m += n;
      n = m - n;
      return m;
   }
};
long SumaSerie::n = 0;
long SumaSerie::m = 1;
```

9. FUNCIONES AMIGAS (FRIEND)

Con el mecanismo amigo (friend) se permite que funciones no miembros de una clase puedan acceder a sus miembros privados o protegidos. Se puede hacer friend de una clase una función global, o bien otra clase. En el siguiente ejemplo la función global *distancia()* se declara friend de la clase **Punto**.

```
double distancia(const Punto& P2)
{
    double d;
    d = sqrt((double)(P2.x * P2.x + P2.y * P2.y));
}
class Punto
{
    friend double distancia(const Punto& P2);
    // ...
}
```

Si *distancia()* no fuera amiga de Punto no podrá acceder directamente a los miembros privado *x* e *y*. Es muy habitual sobrecargar el operador ≪ (la sobrecarga de operadores se verá más adelante) para mostrar por pantalla los objetos de una clase con *cout*. Esto quiere decir que al igual que se escribe un número entero, por ejemplo:

```
int k = 9;
cout << " valor de k = " << k;
```

se pueda escribir un objeto Punto:

```
Punto p(1, 5);
cout << " Punto " << p;
```

Para conseguir esto hay que definir la función global operator≪ y hacerla amiga de la clase, en el ejemplo de la clase Punto:

```
ostream& operator << (ostream& pantalla, const Punto& mp)
{
   pantalla << " x = " << mp.x << ", y = " << mp.y << endl;
   return pantalla;
}
class Punto
{
   friend
   ostream& operator << (ostream& pantalla, const Punto& mp);
   // ...
};
```

Una clase completa se puede hacer amiga de otra clase. De esta forma todas las funciones miembro de la clase amiga pueden acceder a los miembros protegidos de la otra clase. Por ejemplo, la clase MandoDistancia se hace amiga de la clase Television:

```
class MandoDistancia { ... };
class Television
{
 friend class MandoDistancia;
 // ...
}
```

> La declaración de amistad empieza con la palabra reservada friend, solo puede aparecer dentro de la declaración de una clase. Se puede situar en cualquier parte de la clase, es práctica recomendada agrupar todas las declaraciones friend inmediatamente a continuación de la cabecera de la clase

10. Errores de Programación

Las facilidades en el uso de la estructura clase aumenta las posibilidades de errores.

1. *Mal uso de palabras reservadas.* Es un error declarar una clase sin utilizar fielmente una de las palabras reservadas class, struct o union en la declaración.

2. La palabra reservada class no se necesita para crear objetos de clases. Por ejemplo, se puede escribir

   ```
   class C {
   ...
   };
   ```

```
C c1, c2 // definiciones de objetos
en lugar de
class C c1, c2 // no se necesita class
```

3. Si un miembro de una clase es privado (private), se puede acceder solo por funciones amigas de la clase. Por ejemplo, este código no es correcto.

```
class C {
int x;
...
public :
C( ) {x = -9999;}
};
void f( )
{
C c;
cout << c.x // ERROR
}
```

   ya que x es privado en la clase C y la función f no es ni un método ni una amiga de C.

4. Uso de constructores y destructores.

   a) No se puede especificar un tipo de retorno en un constructor o en un destgructor, ni en su declaración ni en su definición. Si C es una clase, entonces no será legal:

   ```
   void C::C(int n) // ERROR
   {
   ...
   }
   ```

   b) De igual modo no se puede especificar ningún argumento en la definición o declaración de un constructor por omisión de la clase:

   ```
   C::C(void) // ERROR
   {
   ...
   }
   ```

   c) Tampoco se pueden especificar argumentos, incluso void, en la declaración o definición del destructor de una clase.

   d) No se pueden tener dos constructores de la misma clase con los mismos tipos de argumentos.

   e) Un constructor de una clase C no puede tener un argumento de un tipo C, pero sí puede tener un argumento de tipo referencia, C&. Este constructor es el constructor de copia.

5. Inicializadores de miembros

```
class C {
// ERROR
int x = 5;
};
class C {
int x;
...
public:
C( ) {x = 5;} // CORRECTO
};
```

6. No se puede acceder a un miembro dato protegido (protected) fuera de su jerarquía de clases, excepto a través de una función amiga. Por ejemplo,

```
class C { // clase base
   protected:
      int x;
};
class D:public C { // clase derivada
   ...
};
int main( )
{
   C c1;
   c1.x = 10 // ERROR, x es protegido
   ...
}
```

   contiene un error, ya que x solo es accesible por métodos de C y métodos y amigas de las clases derivadas de C , como D.

7. No se pueden sobrecargar ninguno de estos operadores:

```
. .* :: ?: sizeof
```

8. No se puede utilizar this como parámetro, ya que es una palabra reservada. Tampoco se puede utilizar en una sentencia de asignación como

```
this = ...; // this es constante; ERROR
```

   porque this es una constante

9. Un constructor de una clase C no puede esperar un argumento de tipo C:

```
class C {
...
};
C::C(C c) // ERROR
{
...
}
C::C(C& c) // CORRECTO
{
...
}
```

10. Un miembro dato static no se puede definir dentro de una clase, aunque sí puede ser declarado dentro de la declaración de la clase.

```
class C {
   static int x = 7; // ERROR
   ...
};
class D {
   static int x;
};
// definición con inicialización
int D::x = 7;
```

11. Los constructores o destructores no pueden declarar static.

12. Olvido de puntos y coma en definición de clases. Las llaves {} son frecuentes en código C++, y, normalmente, no se sitúa un punto y coma después de la llave de cierre. Sin embargo, la definición de *class* siempre termina en };. Un error típico es olvidar ese punto y coma.

```
class Producto
{
public:
   // ...
private:
   //...
} // olvido del punto y coma
```

13. Olvido de inicialización de todos los campos de un constructor. Todo constructor necesita asegurar que todos los campos (miembros) de datos se fijan a los valores apropiados.

```
class Empleado {
   public:
      Empleado( );
      Empleado(string n); // se olvida el parámetro salario
      //...
   private:
      string nombre;
      float salario;
};
```

14. Referencia a un atributo privado de una clase. Los identificadores declarados como atributos o funciones privados de una clase no pueden ser referenciados desde el exterior de la clase. Cualquier intento de referencia producirá mensaje de error similar a
"undefined symbol"

15. Fallo por no incluir un archivo de cabecera requerido. Se generan numerosos mensajes de error por este fallo de programación ya que un archivo de cabecera contiene la definición de un número de los identificadores utilizados en su programa y el mensaje más frecuente de estos errores será: "undefined symbol"

16. Fallo al definir una función como miembro de una clase. Este error se puede producir de varias formas:

    a) Fallo al prefijar la definición de la función por el nombre de su clase y el operador de resolución de ámbito (::).

    b) Fallo al escribir el nombre de la función correctamente.

    c) Omisión completa de la definición de la función de la clase.
    En cualquiera de los casos, el resultado es el mismo: un mensaje de error del compilador que indica que la función llamada no existe en la clase indicada. "is not a member"

17. Olvido de puntos y coma en prototipos y cabeceras de funciones. La omisión de un punto y coma al final del prototipo de una función puede producir el mensaje de error
"Statement missing"
o bien
"Declaration terminated incorrectly".

11. Problemas

1. Modificar el programa de numero complejos para incluya las operaciones de división, potencia de un numero complejo a con un numero complejo b (a^b), potencia de un numero complejo a con un numero real x (a^x) y radicación.

2. Escribir una clase Conjunto que gestione un conjunto de enteros (int) con ayuda de una tabla de tamaño fino (un conjunto contiene una lista ordenada de elementos y se caracteriza por el hecho de que cada elemento es único: no se debe encontrar dos veces el mismo valor en la tabla). Las operaciones a implementar son las siguientes:

   a) La función vacía() vacía el conjunto.

   b) La función agregar() añade un entero al conjunto.

   c) La función eliminar() retira un entero del conjunto.

   d) La función copiar() recopila un conjunto en otro.

   e) La función es_miembro() reenvía un valor booleano (valor lógico que indica si el conjunto contiene un entero dado).

   f) La función es_igual() reenvía un valor booleano que indica si un conjunto es igual a otro.

   g) La función imprimir() realiza la visualización con un formato que usted establezca del conjunto.

3. Se desea realizar una clase Vector3d que permita manipular vectores de tres componentes (coordenadas x,y, z) de acuerdo a las siguientes normas:

   a) Sólo posee una función constructor y es en línea.

   b) Tiene una función miembro *igual* que permite saber si dos vectores tienen sus componentes o coordenadas iguales.

4. Incluir en la clase Vector3d del ejercicio anterior una función miembro denominada *normamax* que permita obtener la norma mayor de dos vectores (Nota: La norma de un vector $v = (x, y, z)$ es $\sqrt{x^2 + y^2 + z^2}$).

5. Añadir a la clase Vector3d las funciones miembros *suma* (suma de dos vectores), *productoEscalar* (producto escalar de dos vectores: $v_1 = (x_1, y_1, z_1)$; $v_2 = (x_2, y_2, z_2)$; $v_1 \cdot v_2 = x_1 * x_2 + y_1 * y_2 + z_1 * z_2$)

6. Entre las operaciones con vectores es muy importante contar con el producto vectorial $v_1 \times v_2$. Incluir en la clase Vector3d una función miembro *productoVectorial* que realice el cálculo.
   Nota: El producto vectorial queda expresado por la matriz

$$v_1 \times v_2 = \begin{pmatrix} \hat{i} & \hat{j} & \hat{k} \\ v_{1x} & v_{1y} & v_{1z} \\ v_{2x} & v_{2y} & v_{2z} \end{pmatrix}$$

$$(v_{1y}\, v_{2z} - v_{1z}\, v_{2y})\hat{i} - (v_{1x}\, v_{2z} - v_{1z}\, v_{2x})\hat{j} + (v_{1x}\, v_{2y} - v_{1y}\, v_{2x})\hat{k}$$

   Otra forma de calcular el producto vectorial es mediante la matriz antisimétrica del segundo vector

$$v_1 \times v_2 = \begin{bmatrix} v_{1x} & v_{1y} & v_{1z} \end{bmatrix} \begin{pmatrix} 0 & -v_{2z} & v_{2y} \\ v_{2z} & 0 & -v_{2x} \\ -v_{2y} & v_{2x} & 0 \end{pmatrix} = \begin{pmatrix} v_{1y}\, v_{2z} - v_{1z}\, v_{2y} \\ -v_{1x}\, v_{2z} + v_{1z}\, v_{2x} \\ v_{1x}\, v_{2y} - v_{1y}\, v_{2x} \end{pmatrix}$$

7. ¿Cuál es el error de la siguiente declaración de clase?

```
union float_bytes {
private :
char mantisa[3], char exponente ;
public:
```

```
float num;
char exp(void);
char mank(void);
};
```

8. ¿Qué está mal en la siguiente defiición de la clase?

```
#include <string.h>
struct buffer {
char datos[255];
int cursor ;
void iniciar(char *s);
inline int Long(void);
char *contenido(void);
};
void buffer::iniciar(char *s)
{
strcpy(datos,s) ; cursor = 0;
}
char *buffer::contenido(void)
{
if( Long( )) return datos; else
return 0;
}
inline int buffer:: Long(void)
{return cursor;}
```

9. Examine la siguiente declaración de clase y vea si existen errores.

```
class punto {
int x, y;
void punto(int x1, int y1)
{x = x1; y = y1;)
};
```

10. Dado el siguiente programa C++, escribir un programa C equivalente.

```
#include <stdio.h>
class operador {
public:
   float memoria;
   operador(void);
   ~operador(void);
   float sumar(float f);
};

operador::operador(void)
{
   printf("Activar maquina operador\n");
   memoria = 0.0
}

operador::~operador(void)
```

```
{
   printf("Desactivar maquina operador \n");
}
float operador::sumar(float f)
{
   memoria += f;
   return memoria;
}

int main( )
{
   operador o
   o.sumar(10.0);
   o.sumar(30.0);
   o.sumar(3.0);
   printf("la solucion es % f\n" e.memoria);
}
```

11. ¿Cuál es la diferencia de signifiado entre la estructura?

```
struct a {
int i, j, k;
};
```

y la clase

```
class a {
 int i, j, k
};
```

Explique la razón por la que la declaración de la clase no es útil. ¿Cómo se puede utilizar la palabra reservada public para cambiar la declaración de la clase en una declaración equivalente a struct?

12. Dadas las siguientes declaraciones de clase y array. ¿Por qué no se puede construir el array?

```
class punto {
public:
   int x, y;
   punto( int a, int b) {x = a ; y = b;}
};
punto poligono[5];
```