# Capítulo 2
# Clases y objetos

## Objetivos

Con el estudio de este capítulo usted podrá:

- Entender el concepto de encapsulación de datos a través de las clases.
- Definir clases como una estructura que encierra datos y métodos.
- Especificar tipos abstractos de datos a través de una clase.
- Establecer controles de acceso a los miembros de una clase.
- Identificar los métodos de una clase con el comportamiento o funcionalidad de los objetos.

## Contenido



## Conceptos clave

- Componentes.
- Constructores.
- Encapsulación.
- Especificadores de acceso: `public`, `protected`, `private`.
- Ocultación de la información.
- Reutilización.



www.FreeLibros.me

## INTRODUCCIÓN

La modularización de un programa utiliza la noción de *tipo abstracto de dato* (**TAD**) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado **TAD**.

Una **clase** es un tipo de dato que contiene código (métodos) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, tal como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En el capítulo se aprenderá a crear (definir y especificar) y utilizar clases individuales.

## 2.1. CLASES Y OBJETOS

Las tecnologías orientadas a objetos combinan la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos como instancias o ejemplares de ellas, son los elementos clave sobre los que se articula la orientación a objetos.

### 2.1.1. ¿Qué son objetos?

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, tales como posición, tamaño, color, forma, textura, etc., que definen su estado. Los objetos también tienen ciertos comportamientos que los hacen diferentes de otros objetos.

Booch[1], define un *objeto* como "algo que tiene un estado, un comportamiento y una identidad". Supongamos una máquina de una fábrica. El *estado* de la *máquina* puede estar *funcionando/parada* ("on/of"), su potencia, velocidad máxima, velocidad actual, temperatura, etc. Su *comportamiento* puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, condiciones de señal de error o cambiar la velocidad. Su *identidad* se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar en el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En un diseño de un programa orientado a objetos, se crea una abstracción (un modelo simplificado) de la máquina basado en las propiedades y comportamiento que son útiles en el tiempo.

Martin/Odell definen un objeto como "cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos".

Un *mensaje* es una instrucción que se envía a un objeto y que cuando se recibe ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz*, un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una

[1] Booch, Grady. *Análisis y diseño orientado a objetos con aplicaciones*. Madrid : Díaz de Santos/Addison-Wesley, 1995. (libro traducido del inglés por Luis Joyanes, coautor de esta obra).

www.FreeLibros.me

interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico tal como una máquina de fax tiene una interfaz de usuario bien definida; esa interfaz incluye el mecanismo de avance del papel, botones de marcado, receptor y el botón "enviar". El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles.

### 2.1.2. ¿Qué son clases?

En términos prácticos, una **clase** es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch denomina a una clase como "un conjunto de objetos que comparten una estructura y comportamiento comunes".

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios* o *métodos*. Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos, variables* o *variables instancia*. El término *atributo* se utiliza en análisis y diseño orientado a objetos y el término *variable instancia* se suele utilizar en programas orientados a objetos.

## 2.2. DECLARACIÓN DE UNA CLASE

Antes de que un programa pueda crear objetos de cualquier clase, la clase debe ser *declarada* y los métodos definidos (implementados). La declaración de una clase significa dar a la misma un nombre, darle nombre a los elementos que almacenan sus datos y describir los métodos que realizarán las acciones consideradas en los objetos.

**Formato**

```
class NombreClase
{
    lista_de_miembros
};
```

| | |
|---|---|
| *NombreClase* | Nombre definido por el usuario que identifica a la clase (puede incluir letras, números y subrayados como cualquier identificador). |
| *lista_de_miembros* | Datos y funciones miembros de la clase. |

Las *declaraciones* o *especificaciones* no son código de programa ejecutable. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer los métodos que utilizará el programa. En C++ los métodos se denominan funciones miembro, normalmente en la declaración sólo se escribe el prototipo de la función. Las declaraciones de las clases se sitúan en archivos `.h` (`NombreClase.h`) y la implementación de las funciones miembro en el archivo `.cpp` (`NombreClase.cpp`).

www.FreeLibros.me

**EJEMPLO 2.1.** Definición de una clase llamada `Punto` que contiene las coordenadas *x* e *y* de un punto en un plano.

La declaración de la clase se guarda en el archivo *Punto.h*:

```
//archivo Punto.h
class Punto
{
private:
   int x, y;                    // coordenadas x, y
public:
   Punto(int x_, int y_)        // constructor
   {
       x = x_;
       y = y_;
   }
   Punto() { x = y = 0;}        // constructor sin argumentos
   int leerX() const;           // devuelve el valor de x

   int leerY() const;           // devuelve el valor de y
   void fijarX(int valorX)      // establece el valor de x
   void fijarY(int valorY)      // establece el valor de y
};
```

La definición de las funciones miembro se realiza en el archivo *Punto.cpp*:

```
#include "Punto.h"
int Punto::leerX() const
{
    return x;
}
int Punto::leerY() const
{
    return y;
}
void Punto::fijarX(int valorX)
{
    x = valorX;
}
void Punto::fijarY(int valorY)
{
    y = valorY;
}
```

### 2.2.1. Objetos

Una vez que una clase ha sido definida, un programa puede contener una *instancia* de la clase, denominada un *objeto de la clase*. Un objeto se crea de forma estática, de igual forma que se define una variable. También de forma dinámica, con el operador `new` aplicado a un constructor de la clase. Por ejemplo, un objeto de la clase `Punto` inicializado a las coordenadas `(2,1)`:

www.FreeLibros.me

```
Punto p1(2, 1);                 // objeto creado de forma estática
Punto* p2 = new Punto(2, 1);    // objeto creado dinámicamente
```

**Formato para crear un objeto**

```
NombreClase varObj(argumentos_constructor);
```

**Formato para crear un objeto dinámico**

```
NombreClase* ptrObj;
ptrObj = new NombreClase(argumentos_constructor);
```

Toda clase tiene una o más funciones miembro, denominadas constructores, para inicializar el objeto cuando es creado; tienen el mismo nombre que el de la clase, no tienen tipo de retorno y pueden estar sobrecargados.

El *operador de acceso* a un miembro del objeto, selector punto (`.`), selecciona un miembro individual de un objeto de la clase. Por ejemplo:

```
Punto p2;                       // llama al constructor sin argumentos
p2.fijarX(10);
cout << " Coordenada  x es " << p2.leerX();
```

El otro *operador de acceso* es el selector *flecha* (`->`), selecciona un miembro de un objeto desde un puntero a la clase. Por ejemplo:

```
Punto* p;
p = new Punto(2, -5);           // crea objeto dinámico
cout << " Coordenada  y es " << p -> leerY();
```

## 2.2.2. Visibilidad de los miembros de la clase

Un principio fundamental en programación orientada a objetos es la *ocultación de la información*. Significa que determinados datos del interior de una clase no se puede acceder por funciones externas a la clase. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o métodos privados sólo se puede acceder desde dentro de la clase. Por el contrario, los datos o métodos públicos son accesibles desde el exterior de la clase (véase la Figura 2.1).

Para controlar el acceso a los miembros de la clase se utilizan tres diferentes *especificadores de acceso:* `public`, `private` y `protected`. Cada miembro de la clase está precedido del especificador de acceso que le corresponde.

**Formato**

```
class NombreClase
{
private:
     declaraciones de miembros privados;
protected:
     declaraciones de miembros protegidos;
```

www.FreeLibros.me

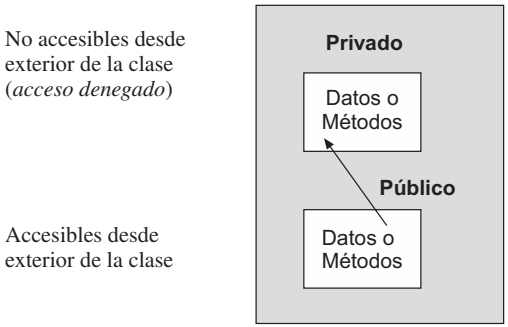


**Figura 2.1.** Secciones pública y privada de una clase.

```
public:
     declaraciones de miembros públicos;
};
```

Por omisión, los miembros que aparecen a continuación de la llave de inicio de la clase, `{`, son privados. A los miembros que siguen a la etiqueta `private` sólo se puede acceder por funciones miembro de la misma clase. A los miembros `protected` sólo se puede acceder por funciones miembro de la misma clase y de las clases derivadas. A los miembros que siguen a la etiqueta `public` se puede acceder desde dentro y desde el exterior de la clase. Las secciones `public`, `protected` y `private` pueden aparecer en cualquier orden.

---

**EJEMPLO 2.2.** Declaración de la clase `Foto` y `Marco` con miembros declarados con distinta visibilidad.

```
class Foto
{
private:
     int nt;
     char opd;
protected:
     string q;
public:
     Foto(string r)     // constructor
     {
          nt = 0;
          opd = 'S';
          q = r;
     }
     double mtd();
};

class Marco
{
private:
     double p;
     string t;
```

www.FreeLibros.me

```
public:
     Marco();       // constructor
     void poner()
     {
         Foto* u = new Foto("Paloma");
         p = u -> mtd();
         t = "**" + u -> q + "**";
     }
};
```

**Tabla 2.1.** Visibilidad, "x" indica que el acceso está per mitido

| Tipo de miembro | Miembro de la misma clase | Miembro de una clase derivada | Miembro de otra clase (externo) |
|---|---|---|---|
| `private` | x | | |
| `protected` | x | x | |
| `public` | x | x | x |

Aunque las secciones *públicas*, *privadas* y *protegidas* pueden aparecer en cualquier orden, los programadores suelen seguir ciertas reglas en el diseño que citamos a continuación, y que usted puede elegir la que considere más eficiente.

1. Poner los miembros privados primero, debido a que contiene los atributos (datos).
2. Se pone los miembros públicos primero debido a que los métodos y los constructores son la interfaz del usuario de la clase.

En realidad, tal vez el uso más importante 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 que permite que los detalles de implementación de los objetos sean ignorados. Por consiguiente, los datos y métodos públicos forman la interfaz externa del objeto, mientras que los elementos *privados* son los aspectos internos del objeto que no necesitan ser accesibles para usar el objeto.

> El principio de *encapsulamiento* significa que las estr ucturas de datos inter nas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

## 2.2.3. Funciones miembro de una clase

La declaración de una clase incluye la declaración o prototipo de las funciones miembros (métodos). Aunque la implementación se puede incluir dentro del cuerpo de la clase (`inline`), normalmente se realiza en otro archivo (con extensión `.cpp`) que constituye la definición de la clase. La Figura 2.2 muestra la declaración de la clase `Producto`.

www.FreeLibros.me

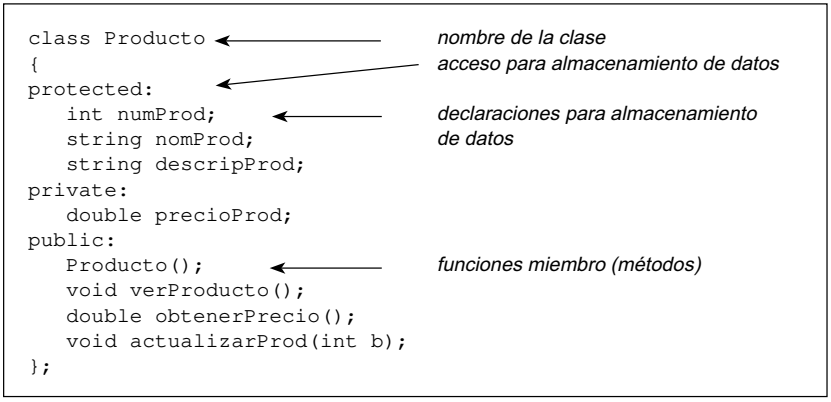


**Figura 2.2.** Definición típica de una clase.

---

**EJEMPLO 2.3.** La clase `Racional` representa un número racional. Por cada dato, numerador y denominador, se proporciona una función miembro que devuelve su valor y otra función para asignar numerador y denominador. Tiene un constructor que inicializa un objeto a 0/1.

En esta ocasión las funciones miembro se implementan directamente en el cuerpo de la clase.

```
   // archivo Racional.h
class Racional
{
private:
    int numerador;
    int denominador;
public:
    Racional()
    {
        numerador = 0;
        denominador = 1;
    }
    int leerN() const { return numerador; }
    int leerD() const { return denominador; }
    void fijar (int n, int d)
    {
        numerador = n;
        denominador = d;
    }
};
```

---

## 2.2.4. Funciones en línea y fuera de línea

Las funciones miembro definidas dentro del cuerpo de la declaración de la clase se denominan definiciones de funciones *en línea* (`inline`). Para el caso de funciones más grandes, es preferible codificar sólo el prototipo de la función dentro del bloque de la clase y codificar la im-

www.FreeLibros.me

plementación de la función en el exterior. Esta forma permite al creador de una clase ocultar la implementación de la función al usuario de la clase proporcionando sólo el código fuente del archivo de cabecera, junto con un archivo de implementación de la clase precompilada.

En el siguiente ejemplo, `FijarEdad()` de la clase `Lince` se declara pero no se define en la declaración de la clase:

```
class Lince
{
public:
   void FijarEdad(int a);
private:
   int edad;
   string habitat;
};
```

La implementación de una función miembro externamente a la declaración de la clase, se hace en una definición de la función *fuera de línea*. Su nombre debe ser precedido por el nombre de la clase y el signo de puntuación :: denominado *operador de resolución de ámbito*. El operador :: permite al compilador conocer que `FijarEdad()` pertenece a la clase `Lince` y es, por consiguiente, diferente de una función global que pueda tener el mismo nombre o de una función que tenga ese nombre que puede existir en otra clase. La siguiente función global, por ejemplo, puede coexistir dentro del mismo ámbito que `Lince::FijarEdad()`:

```
// función global:
void FijarEdad(int valx)
{
   // ...
}
// función en el ámbito de Lince:
void Lince::FijarEdad(int a)
{
    edad = a;
}
```

### 2.2.5. La palabra reservada `inline`

La decisión de elegir funciones en línea y fuera de línea es una cuestión de eficiencia en tiempo de ejecución. Una función en línea se ejecuta normalmente más rápida, ya que el compilador inserta una copia «fresca» de la función en un programa en cada punto en que se llama a la función. La definición de una función miembro en línea no garantiza que el compilador lo haga realmente en línea; es una decisión que el compilador toma, basado en los tipos de las sentencias dentro de la función y cada compilador de C++ toma esta decisión de modo diferente.

Si una función se compila en línea, se ahorra tiempo de la UCP (CPU) al no tener que ejecutar una instrucción "*call*" (llamar) para bifurcar a la función y no tener que ejecutar una instrucción `return` para retornar al programa llamador. Si una función es corta y se llama cientos de veces, se puede apreciar un incremento en eficiencia cuando actúa como función en línea.

Una función localizada fuera del bloque de la definición de una clase se puede beneficiar de las ventajas de las funciones en línea si está precedida por la palabra reservada `inline`:

www.FreeLibros.me

```
inline void Lince::FijarEdad(int a)
{
    edad = a;
}
```

Dependiendo de la implementación de su compilador, las funciones que utilizan la palabra reservada `inline` se puede situar en el mismo archivo de cabecera que la definición de la clase. Las funciones que no utilizan `inline` se sitúan en el mismo módulo de programa, pero no el archivo de cabecera. Estas funciones se sitúan en un archivo `.cpp`.

---

### Ejercicio 2.1

*Definir una clase `DiaAnyo` con los atributos `mes` y `día`, los métodos `igual()` y `visualizar()`. El mes se registra como un valor entero (1, Enero, 2, Febrero, etc.). El día del mes se registra en otra variable entera día. Escribir un programa que compruebe si una fecha es su cumpleaños.*

La función *principal*, `main()`, crea un objeto `DiaAnyo` y llama al método `igual()` para determinar si coincide la fecha del objeto con la fecha de su cumpleaños, que se ha leído de dispositivo de entrada.

```
// archivo DiaAnyo.h

class DiaAnyo
{
private:
   int dia, mes;
public:
    DiaAnyo(int d, int m);
    bool igual(const DiaAnyo& d) const;
    void visualizar() const;
};
```

La implementación de las funciones miembro se guarda en el archivo `DiaAnyo.cpp`:

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

DiaAnyo::DiaAnyo(int d, int m)
{
    dia = d;
    mes = m;
}

bool DiaAnyo::igual(const DiaAnyo& d)const
{
    if ((dia == d.dia) && (mes == d.mes))
        return true;
    else
        return false;
}
```

www.FreeLibros.me

```
void DiaAnyo::visualizar() const
{
    cout << "mes = " << mes << " , dia = " << dia << endl;
}
```

Por último, el archivo `DemoFecha.cpp` contiene la función `main()`, crea los objetos y se envian mensajes.

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

int main()
{
    DiaAnyo* hoy;
    DiaAnyo* cumpleanyos;
    int d, m;

    cout << "Introduzca fecha de hoy, dia: ";
    cin >> d;
    cout << "Introduzca el número de mes: ";
    cin >> m;
    hoy = new DiaAnyo(d, m);
    cout << "Introduzca su fecha de nacimiento, dia: ";
    cin >> d;
    cout << "Introduzca el número de mes: ";
    cin >> m;
    cumpleanyos = new DiaAnyo(d, m);
    cout << " La fecha de hoy es ";
    hoy -> visualizar();
    cout << " Su fecha de nacimiento es ";
    cumpleanyos -> visualizar();
    if (hoy -> igual(*cumpleanyos))
        cout << "¡Feliz cumpleaños ! " << endl;
    else
        cout << "¡Feliz dia ! " << endl;
    return 0;
}
```

## 2.2.6. Sobrecarga de funciones miembro

Al igual que sucede con las funciones no miembro de una clase, las funciones miembro de una clase se pueden sobrecargar. Una función miembro se puede sobrecargar pero sólo en su propia clase.

Las mismas reglas utilizadas para sobrecargar funciones ordinarias se aplican a las funciones miembro: dos miembros sobrecargados no puede tener el mismo número y tipo de parámetros. La sobrecarga permite utilizar un mismo nombre para una función y ejecutar la función definida más adecuada a los parámetros pasados durante la ejecución del programa. La ventaja fundamental de trabajar con funciones miembro sobrecargadas es la comodidad que aporta a la programación.

www.FreeLibros.me

Para ilustrar la sobrecarga, veamos la clase `Vector` donde aparecen diferentes funciones miembro con el mismo nombre y diferentes tipos de parámetros.

```
class Vector
{
public:
    int suma(int m[], int n);    //funcion 1
    int suma(const Vector& v);   //funcion 2
    float suma(float m, float n);      //funcion 3
    int suma();                  //funcion 4
};
```

### Normas para la sobrecarga

No pueden existir dos funciones en el mismo ámbito con igual signatura (lista de parámetros).

## 2.3. CONSTRUCTORES Y DESTRUCTORES

Un *constructor* es una función miembro que se ejecuta automáticamente cuando se crea un objeto de una clase. Sirve para inicializar los miembros de la clase.

El constructor tiene el mismo nombre que la clase. Cuando se define no se puede especificar un valor de retorno, nunca devuelve un valor. Sin embargo, puede tomar cualquier número de argumentos.

### Reglas

1. El constructor tiene el mismo nombre que la clase.
2. Puede tener cero, o más argumentos.
3. No tiene tipo de retorno.

---

**EJEMPLO 2.4.** La clase `Rectángulo` tiene un constructor con cuatro parámetros.

```
class Rectangulo
{
private:
    int izdo, superior;
    int dcha, inferior;
public:
    Rectangulo(int iz, int sr, int d, int inf) // constructor
    {
        izdo = iz; superior = sr;
        dcha = d; inferior = inf;
    }
    // definiciones de otros métodos miembro
};
```

www.FreeLibros.me

Al crear un objeto se pasan los valores al constructor, con la misma sintaxis que la de llamada a una función. Por ejemplo:

```
Rectangulo Rect(4, 4, 10, 10);
Rectangulo* ptr = new Rectangulo(25, 25, 75, 75);
```

Se han creado dos instancias de `Rectangulo`, pasando valores concretos al constructor de la clase.

### 2.3.1. Constructor por defecto

Un constructor que no tiene parámetros, o que se puede llamar sin argumentos, se llama *constructor por defecto*. Un constructor por defecto normalmente inicializa los miembros dato de la clase con valores por defecto.

> ***Regla***
>
> C++ crea automáticamente un constructor por defecto cuando no existen otros constructores. Tal constructor inicializa el objeto a ceros binarios.

**EJEMPLO 2.5.** El constructor de la clase `Complejo` tiene dos argumentos con valores por defecto `0` y `1` respectivamenete, por tanto, puede llamarse sin argumentos.

```
class Complejo
{
private                          :
   double x;
   double y;
public:
   Complejo(double r = 0.0, double i = 1.0)
   {
      x = r;
      y = i;
   }
};
```

Cuando se crea un objeto `Complejo` puede inicializarse a los valores por defecto, o bien a otros valores.

```
Complejo z1;   // z1.x == 0.0, z1.y == 1.0
Complejo* pz = new Complejo(-2, 3);  // pz -> x == -2, pz -> y == 3
```

### 2.3.2. Constructores sobrecargados

Al igual que se puede sobrecargar un método de una clase, también se puede sobrecargar el constructor de una clase. De hecho los constructores sobrecargados son bastante frecuentes, proporcionan diferentes alternativas de inicializar objetos.

www.FreeLibros.me

### Regla

Para prevenir a los usuarios de la clase de crear un objeto sin parámetros, se puede: 1) omitir el constructor por defecto; o bien, 2) hacer el constructor privado.

**EJEMPLO 2.6.** La clase `EquipoSonido` se define con tres constructores; un constructor por defecto, otro con un argumento de tipo cadena y el tercero con tres argumentos.

```
class EquipoSonido
{
private:
    int potencia, voltios;
    string marca;
public:
    EquipoSonido()   // constructor por defecto
    {
        marca = "Sin marca"; potencia = voltios = 0;
    }
    EquipoSonido(string mt)
    {
        marca = mt; potencia = voltios = 0;
    }
    EquipoSonido(string mt, int p, int v)
    {
        marca = mt;
        potencia = p;
        voltios = v;
    }
};
```

La instanciación de un objeto `EquipoSonido` puede hacerse llamando a cualquier constructor. A continuación se crean tres objetos:

```
EquipoSonido rt;   // constructor por defecto
EquipoSonido ft("POLASIT");
EquipoSonido gt("PARTOLA", 35, 220);
```

## 2.3.3. Array de objetos

Los objetos se pueden estructurar como un array. Cuando se crea un array de objetos éstos se inicializan llamando al constructor sin argumentos. Por consiguiente, siempre que se prevea organizar los objetos en un array, la clase debe tener un constructor que pueda llamarse sin parámetros.

### Precaución

Tenga cuidado con la escritura de una clase con sólo un constructor con argumentos. Si se omite un constructor que pueda llamarse sin argumento no será posible crear un array de objetos.

www.FreeLibros.me

**EJEMPLO 2.7.** Se crean arrays de objetos de tipo `Complejo` y `EquipoSonido`.

```
Complejo zz[10];  // crea 10 objetos, cada uno se inicializa a 0,1
EquipoSonido* pq; // declaración de un puntero
int n;
cout << "Número de equipos: "; cin >> n;
pq = new EquipoSonido[n];
```

### 2.3.4. Constructor de copia

Este tipo de constructor se activa cuando al crear un objeto se inicializa con otro objeto de la misma clase. Por ejemplo:

```
Complejo z1(1, -3);      // z1 se inicializa con el constructor
Complejo z2 = z1;        /* z2 se inicializa con z1, actúa el
                            constructor de copia              */
```

También se llama al constructor de copia cuando se pasa un objeto por valor a una función, o bien cuando la función devuelve un objeto. Por ejemplo:

```
extern Complejo resultado(Complejo d);
```

para llamar a esta función se pasa un parámetro de tipo `Complejo`, un objeto. En esta transferencia se llama al constructor de copia. Lo mismo ocurre cuando la misma función devuelve (`return`) el resultado, un objeto de la clase `Complejo`.

El constructor de copia es una función miembro de la clase, su prototipo:

```
NombreClase(const NombreClase& origen);
```

el argumento `origen` es el objeto copiado, `z1` en el primer ejemplo. La definición del constructor de copia para la clase `Complejo`:

```
Complejo(const Complejo& origen)
{
    x = origen.x;
    y = origen.y;
}
```

No es siempre necesario definir el constructor de copia, por defecto se realiza una copia miembro a miembro. Sin embargo, cuando la clase tenga atributos (punteros) que representen memoria dinámica, *buffer dinámico*, sí debe definirse, para realizar una *copia segura*, reservando memoria y copiando en esa memoria o *buffer* los elementos.

### 2.3.5. Asignación de objetos

El operador de asignación, =, se puede utilizar con objetos, de igual forma que con datos de tipo simple. Por ejemplo:

www.FreeLibros.me

```
Racional r1(1, 3);
Racional r2(2, 5);
r2 = r1;          // r2 toma los datos del objeto r1
```

Por defecto, la asignación se realiza miembro a miembro. El numerado de `r2` toma el valor del numerador de `r1` y el denominador de `r2` el valor del denominador de `r1`.

C++ permite cambiar la forma de asignar objetos de una clase. Para ello se implementa una función miembro de la clase especial (se denomina sobrecarga del operador `=`) con este prototipo:

```
nombreClase& operator = (const nombreClase&);
```

A continuación se implementa esta función en la clase `Persona`:

```
class Persona
{
private:
     int edad;
     string nom, apell;
     string dni;
public:
     // sobrecarga del operador de asignación
     Persona& operator = (const Persona& p)
     {
          if (this == &p) return *this; // es el mismo objeto
          edad = p.edad;
          nom = p.nom;
          apell = p.apell;
          dni = p.dni;
          return *this;
     }
```

En esta definición se especifica que no se tome acción si el objeto que se asigna es él mismo: `if (this == &p)`.

En la mayoría de las clases no es necesario definir el operador de asignación ya que, por defecto, se realiza una asignación miembro a miembro. Sin embargo, cuando la clase tenga miembros de tipo puntero (memoria dinámica, *buffer dinámico*) sí debe definirse, para que la asignación sea *segura*, reservando memoria y asignando a esa memoria o *buffer* los elementos.

---

**EJEMPLO 2.8.** La clase `Libro` se define con un array dinámico de páginas (clase `Pagina`). El constructor establece el nombre del autor , el número de páginas y crea el arr ay. Además, se escribe el constructor de copia y la sobrecarga del oper ador de asignación.

```
// archivo Libro.h

class Libro
{
private:
    int numPags, inx;
    string autor;
    Pagina* pag;
    // ...
```

www.FreeLibros.me

```
public:
    Libro(string a, int n);     // constructor
    Libro(const Libro& cl);     // constructor de copia
    Libro& operator = (const Libro& al);  // operador de asignación
    // ... funciones miembro
};

// archivo Libro.cpp

Libro::Libro(string a, int n)
{
    autor = a;
    inx = 0;
    numPags = n;
    pag = new Pagina[numPags];
}
// constructor de copia
Libro::Libro(const Libro& cl)
{
    autor = cl.a;
    inx = cl.inx;
    numPags = cl.numPags;
    pag = new Pagina[numPags];  // copia segura
    for (int i = 0; i < inx; i++)
       pag[i] = cl.pag[i];
}
// operador de asignación
Libro& operator = (const Libro& al)
{
    if (this == &p) return *this;
    autor = al.a;
    inx = al.inx;
    numPags = al.numPags;
    pag = new Pagina[numPags];  // copia segura
    for (int i = 0; i < inx; i++)
        pag[i] = al.pag[i];
    return *this;
}
```

## 2.3.6. Destructor

Un objeto se libera, se destruye, cuando se sale del ámbito de definición. También, un objeto creado dinámicamente, con el operador `new`, se libera al aplicar el operador `delete` al puntero que lo referencia. Por ejemplo:

```
Punto* p = new Punto(1,2);
if (...)
{
    Punto p1(2, 1);
    Complejo z1(8, -9);
}   // los objetos p1 y z1 se destruyen
delete p;
```

www.FreeLibros.me

El destructor tiene el mismo nombre que clase, precedido de una tilde (~). Cuando se define, no se puede especificar un valor de retorno, ni argumentos:

```
~NombreClase()
{
    ;
}
```

El destructor es necesario implementarlo cuando el objeto contenga memoria reserva dinámicamente.

---

**EJEMPLO 2.9.** Se declara la clase `Equipo` con dos atributos de tipo puntero, un constructor con valores por defecto y el destructor.

El constructor define una array de `n` objetos `Jugador` con el operador `new`. El destructor libera la memoria reservada.

```
class Equipo
{
private:
    Jugador* jg;
    int numJug;
    int actual;
public:
    Equipo(int n = 12)
    {
        jg = new Jugador[n];
        numJug = n;  actual = 0;
    }
    ~Equipo()    // destructor
    {
         if (jg != 0) delete [] jg;
    }
}
```

---

## 2.4. 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;
```

www.FreeLibros.me

```
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();
```

www.FreeLibros.me

## 2.5. CLASES COMPUESTAS

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 es exactamente el opuesto 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
```

www.FreeLibros.me

```
{
    PonerId(0);
    PonerNotaMedia(0.0);
}
```

### Regla

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, con la sintaxis siguiente:

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

## 2.6. 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`.

### 2.6.1. Variables `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;
```

www.FreeLibros.me

### Formato

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 dato estático `X` de la clase `Punto`.

---

**Ejercicio 2.2**

*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
}
```

www.FreeLibros.me

```
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)
    {
        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 0;
}
```

## 2.6.2. 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 2.10.** 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()
```

www.FreeLibros.me

```
    {
        m += n;
        n = m - n;
        return m;
    }
};
long SumaSerie::n = 0;
long SumaSerie::m = 1;
```

## 2.7. 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 << 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
{
```

www.FreeLibros.me

```
    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;
   // ...
}
```

### *Regla*

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

## 2.8. TIPOS ABSTRACTOS DE DATOS EN C++

La estructura más adecuada, en C++, para implementación un **TAD** es la clase. Dentro de la clase va a residir la representación de los datos junto a las operaciones (funciones miembro de la clase). La interfaz del tipo abstracto queda perfectamente determinado con la etiqueta `public`, que se aplicará a las funciones de la clase que representen operaciones.

Por ejemplo, si se ha especificado el **TAD** `PuntoTres` para representar la abstracción punto en el espacio tridimensional, la clase que implenta el tipo:

```
class PuntoTres
{
private:
   double x, y, z;              // representación de los datos
public:                         // operaciones
   double distancia(PuntoTres p);
   double modulo();
   double anguloZeta();
    ...
};
```

### 2.8.1. Implementación del TAD Conjunto

En el Apartado 1.6.2 se ha especificado el **TAD** `Conjunto`, la implementación se realiza con la clase `Conjunto`. Se supone que los elementos del conjunto son cadenas (`string`), aunque se podría generalizar creando una clase plantilla (`template`), pero se prefiere simplificar el desarrollo y más adelante utilizar la genericidad.

www.FreeLibros.me

La declaración de la clase está en el archivo `Conjunto.h`, la implementación de las funciones, la definición, en `Conjunto.cpp`. Los elementos del conjunto se guardan en un array que crece dinámicamente.

### Archivo `Conjunto.h`

```
const int M = 20;
class Conjunto
{
private;             // representación
   string* cto;
   int cardinal;
   int capacidad;
public:              // operaciones
   Conjunto(); const;
   Conjunto (const Conjunto& org);
   bool esVacio() const;
   void annadir(string elemento);
   void retirar(string elemento);
   boolean pertenece(string elemento) const;
   int cardinal() const;
   Conjunto union(const Conjunto& c2);
};
```

### Archivo `Conjunto.cpp`

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

Conjunto::Conjunto()
{
    cto = new string[M];
    cardinal = 0;
    capacidad = M;
}

Conjunto::Conjunto(const Conjunto& org)
{
    cardinal = org.cardinal;
    capacidad = org.capacidad;
    cto = new string[capacidad];  // copia segura
    for (int i = 0; i < cardinal; i++)
       cto[i] = opg.cto[i];
}

bool Conjunto::esVacio() const
{
    return (cardinal == 0);
}

void Conjunto::annadir(string elemento)
{
    if (!pertenece(elemento))
```

www.FreeLibros.me

```
    {
        // amplia el conjunto si no hay posiciones libres
       if (cardinal == capacidad)
       {
           string* nuevoCto;
           nuevoCto = new string[capacidad + M];
           for (int k = 0; k < cardinal; k++)
              nuevoCto[k] = cto[k];
           delete cto;
           cto = nuevoCto;
           capacidad += M;
      }
      cto[cardinal++] = elemento;
    }
}

void Conjunto::retirar(string elemento) const
{
    if (pertenece(elemento))
    {
        int k = 0;
        while (!(cto[k] == elemento)) k++;

        // mueve a la izqda desde elemento k+1 hasta última posición
        for (; k < cardinal ; k++)
            cto[k] = cto[k+1];
        cardinal--;
    }
}

bool Conjunto::pertenece(string elemento) const
{
    int k = 0;
    bool encontrado = false;

    while (k < cardinal && !encontrado)
    {
        encontrado = cto[k] == elemento;
        k++;
    }
    return encontrado;
}

int Conjunto::cardinal() const
{
    return this -> cardinal;
}

Conjunto Conjunto::union(const Conjunto& c2)
{
    int k;

    Conjunto u;   // conjunto unión
       // primero copia el primer operando de la unión
    for (k = 0; k < cardinal; k++)
       u.cto[k] = cto[k];
```

www.FreeLibros.me

```
      u.cardinal = cardinal;
         // añade los elementos de c2 no incluidos
      for (k = 0; k < c2.cardinal; k++)
         u.annadir(c2.cto[k]);
      return u;
   }
```

## RESUMEN

En la mayoría de los lenguajes de programación orientados a objetos, y en particular en C++, los tipos abstractos de datos se implementan mediante **clases**.

Una **clase** es un tipo de dato definido por el programador que sirve para representar objetos del mundo real. Un objeto de una clase tiene dos componentes: un conjunto de atributos o variables instancia y un conjunto de comportamientos (métodos, funciones miembro). Los atributos también se llaman variables instancia o miembros dato y los comportamientos se llaman métodos miembro.

```
class Circulo
{
private:
    double centroX, centroY;
    double radio;
public:
   double superficie();
};
```

Un objeto es una instancia de una clase y una variable cuyo tipo sea la clase es un objeto de la clase.

```
Circulo unCirculo;          // objeto Circulo
Circulo tipocirculo [10];   // array de 10 objetos
```

La declaración de una clase, en cuanto a visibilidad de sus miembros, tiene tres secciones: *pública, privada* y *protegida*. La sección pública contiene declaraciones de los atributos y el comportamiento del objeto que son accesibles a los usuarios del objeto. Los constructores inicializan los objetos y se recomienda su declaración en la sección pública. La sección privada contiene los métodos miembro y los miembros dato que son ocultos o inaccesibles a los usuarios del objeto. Estos métodos miembro y atributos dato son accesibles sólo dentro del objeto. Los miembros de una clase con visibilidad `protected` son accesibles desde el interior de la clase y para las clases derivadas.

Un **constructor** es una función miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobrecargado.

```
class Racional
{
public:
    Racional (int nm, int dnm);
    Racional(const Racional& org);
}
```

El **constructor** es un método especial que se invoca cuando se crea un objeto. Se utiliza, normalmente, para inicializar los atributos de un objeto. Por lo general, al menos se define un construc-

www.FreeLibros.me

tor sin argumentos, llamado constructor por defecto. En caso de no definirse constructor, implícitamente queda definido un constructor sin argumentos que inicializa el objeto a ceros binarios.

El proceso de crear un objeto se llama *instanciación* (creación de instancia). En C++ se crea un objeto como se define una variable, o bien dinámicamente con el operador `new` y un constructor de la clase.

```
Racional R(1, 2);
Racional* pr = new Racional(4, 5);
```

En C++ la liberación de objetos ocurre cuando termina el bloque dentro del cual se ha definido, o cuando se aplica el operador `delete` a un puntero al objeto. La liberación la realiza una función miembro especial, denominada destructor y de prototipo: `~NombreClase()`. Realmente sólo es necesario definir el destructor cuando el objeto contenga memoria reservada dinámicamente con el operador `new`.

Los miembros de un clase definidos como `static` no están ligados a los objetos de la clase sino que son comunes a todos los objetos, que son de la clase. Se cualifican con el nombre de la clase, por ejemplo:

```
Matematica::fibonacci(5);
```

## EJERCICIOS

**2.1.** ¿Qué está mal en la siguiente definición de la clase?

```
class Buffer
{
private:
    char* datos;
    int cursor ;
    Buffer(int n)
    {
        datos = new char[n];
    };
public: static int Long( return cursor;)
}
```

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

- Sólo posee una método constructor.
- Tiene una función miembro `igual()` que permite saber si dos vectores tienen sus componentes o coordenadas iguales.

**2.3.** Incluir en la clase `Vector3d` del Ejercicio 2.2 la función `normaMax` que permita obtener la norma de un vector (**Nota:** La norma de un vector $v = (x, y, z)$ es $\sqrt{(x^2 + y^2 + z^2)}$.

**2.4.** Realizar la clase `Complejo` que permita la gestión de números complejos (un número complejo consta de dos números reales de tipo `double` : una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

www.FreeLibros.me

- `establecer()` permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.
- `imprimir()` realiza la visualización formateada de un `Complejo`.
- `agregar()` (sobrecargado) para añadir, respectivamente, un `Complejo` a otro y añadir dos componentes `double` a un `Complejo`.

**2.5.** Añadir a la clase `Complejo` del Ejercicio 2.4 las siguientes operaciones :

- Suma: `a + c = (A + C, (B + D)i)`.
- Resta: `a - c = (A - C, (B - D)i)`.
- Multiplicación: `a * c=(A*C-B*D, (A*D+B*C)i)`.
- Multiplicación: `x * c = (x*C, x*Di)`, donde *x* es real.
- Conjugado: `~a = (A ,-Bi)`.
  Siendo `a = A+Bi; c = C+Di`.

**2.6.** Implementar la clase `Hora`. Cada objeto de esta clase representa una hora específica del día, almacenando las horas, minutos y segundos como enteros. Se ha de incluir un constructor, métodos de acceso, una método `adelantar(int h, int m, int s)` para adelantar la hora actual de un objeto existente, una método `reiniciar(int h, int m, int s)` que reinicializa la hora actual de un objeto existente y una método `imprimir()`.

## PROBLEMAS

**2.1.** Implementar la clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, funciones de acceso, la función `reiniciar (int d, int m, int a)` para reiniciar la fecha de un objeto existente, la función `adelantar(int d, int m, int a)` para avanzar una fecha existente (día `d`, mes `m`, y año `a`) y la función `imprimir()`. Escribir una función auxiliar de utilidad, `normalizar()`, que asegure que los miembros dato están en el rango correcto `1 ≤ año, 1 ≤ mes ≤ 12, día ≤ días (Mes)`, siendo `días(Mes)` otra función que devuelve el número de días de cada mes.

**2.2.** Ampliar el programa 2.1 de modo que pueda aceptar años bisiestos. **Nota:** un año es bisiesto si es divisible por 400, o si es divisible por 4 pero no por 100. Por ejemplo el año 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.

www.FreeLibros.me