

Décimo novena Sesión

Metodologías y Técnicas de Programación II

**Programación Orientada a
Objeto (POO)
C++**

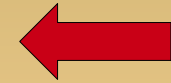
Herencia III

Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

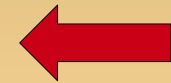
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores
Constantes e "inline"

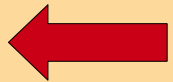
Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

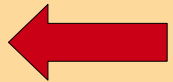
Creación Dinámica de Objetos



Herencia.

Tipos de Visibilidad

Herencia Múltiple



Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.

Plantillas

Contenedores

Iteradores

19.1 Repaso

Herencia y Composición

Formas de reutilización sistemática. Mejor. Más concienzudas.

Composición: Creamos objetos de la clase existente dentro de la nueva clase que estamos creando.

Herencia: Se toma la forma de la clase existente y se añade código sin modificar la clase existente.

La herencia nos permite:

- Crear nuevas clases a partir de clases existentes.

- Conservando las propiedades de la clase original.

- Añadir nuevos métodos y atributos a la clase original.

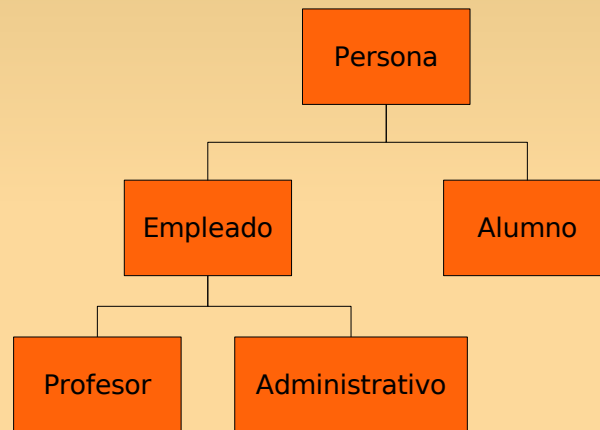
- Redefinir comportamiento de métodos.

19.1 Repaso

Herencia

Cuando una clase deriva de más de una clase base: **Herencia Múltiple.**

Jerarquías de Clases.



```
class <clase_derivada> :  
    [public|private] <base1> [, [public|private] <base2> ] {};
```

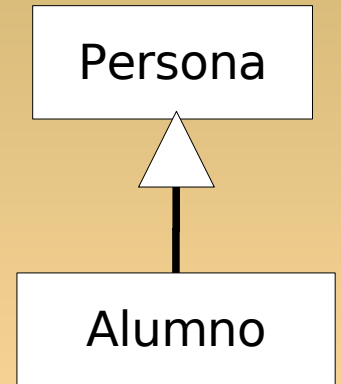
Clase base, clase padre, clase madre, superclase: La clase base es la clase ya creada, de la que se hereda.

Clase derivada, clase hija, clase heredada: es la clase que se crea a partir de la clase base.

19.1 Repaso

Control de Acceso en la Herencia

```
class <clase_derivada> :  
[public|private] <base1> [, [public|private] <base2>] {};
```



| Tipo de Dato Clase Base | Herencia con public | Herencia con private | Otros |
|-------------------------|---------------------------|---------------------------|--------------------|
| Private | No accesible directamente | No accesible directamente | No accesible |
| Protected | Protected | Private | No accesible |
| Public | Public | Private | Accesible (. ó ->) |

19.1 Repaso

Hay Miembros que no se heredan automáticamente

La clase derivada hereda todos los atributos y todos los métodos, excepto:

- **Constructores**
- **Destructor**
- **Operador de asignación**

Constructores y destructor:

Si en la subclase no están definidos, se crea un constructor por defecto y un destructor, aunque sin que hagan nada en concreto.

Operador de asignación:

Si en la subclase no está definido, se crea uno por defecto que se basa en el operador de asignación de la superclase.

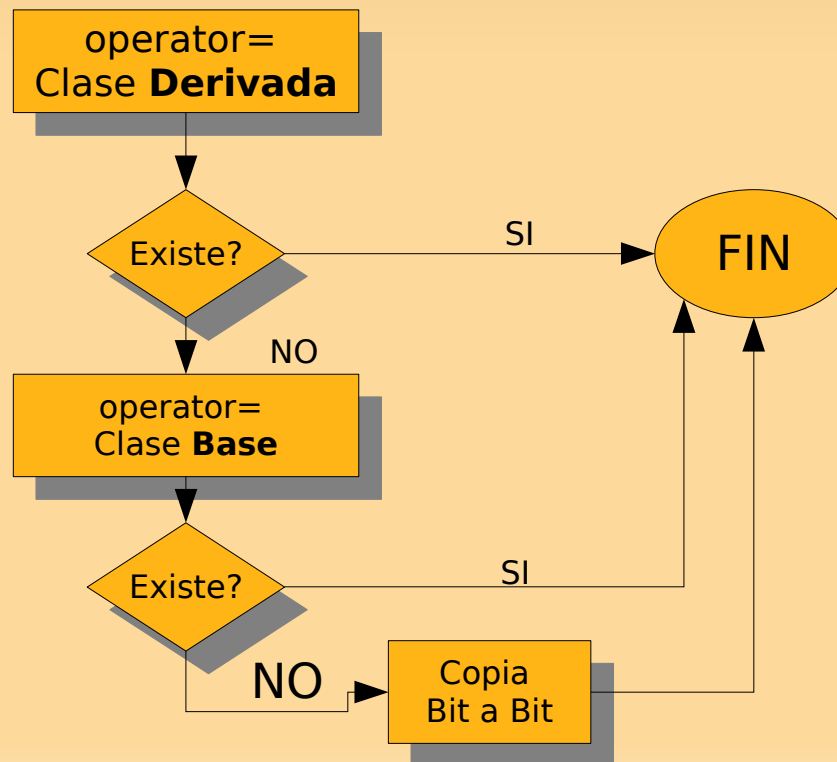
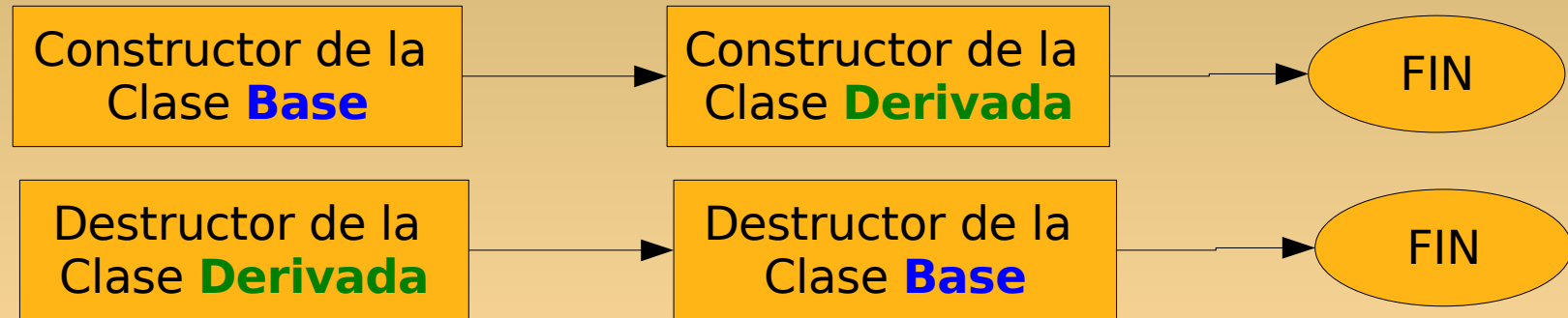
MORALEJA

Se deben crear los constructores y el destructor en la subclase.

Se debe definir el operador de asignación en la subclase.

19.1 Repaso

Orden de llamada



19.1 Repaso

Operador Asignación

```
ClaseBase& ClaseBase ::  
operator = ( ClaseBase c )  
{  
    b1 = c.b1;  
    b2 = c.b2;  
    b3 = c.b3;  
    return *this;  
};  
ClaseDerivada& ClaseDerivada ::  
    operator = ( ClaseDerivada d )  
{  
    s1 = d.s1;  
    s2 = d.s2;  
    return *this;  
};  
ClaseDerivada ob1 ( 6,7,4,8.2, 'f' );  
ClaseDerivada ob2 ( 0,0,0, 0, 'h' );  
ob2 = ob1;
```

¿Qué método se ejecuta?

```
b1 - 6    s1 - 8.2  
b2 - 7    s2 - f  
b3 - 4
```

```
b1 - 0    s1 - 0  
b2 - 0    s2 - h  
b3 - 0
```

```
b1 - 0    s1 - 8.2  
b2 - 0    s2 - f  
b3 - 0
```

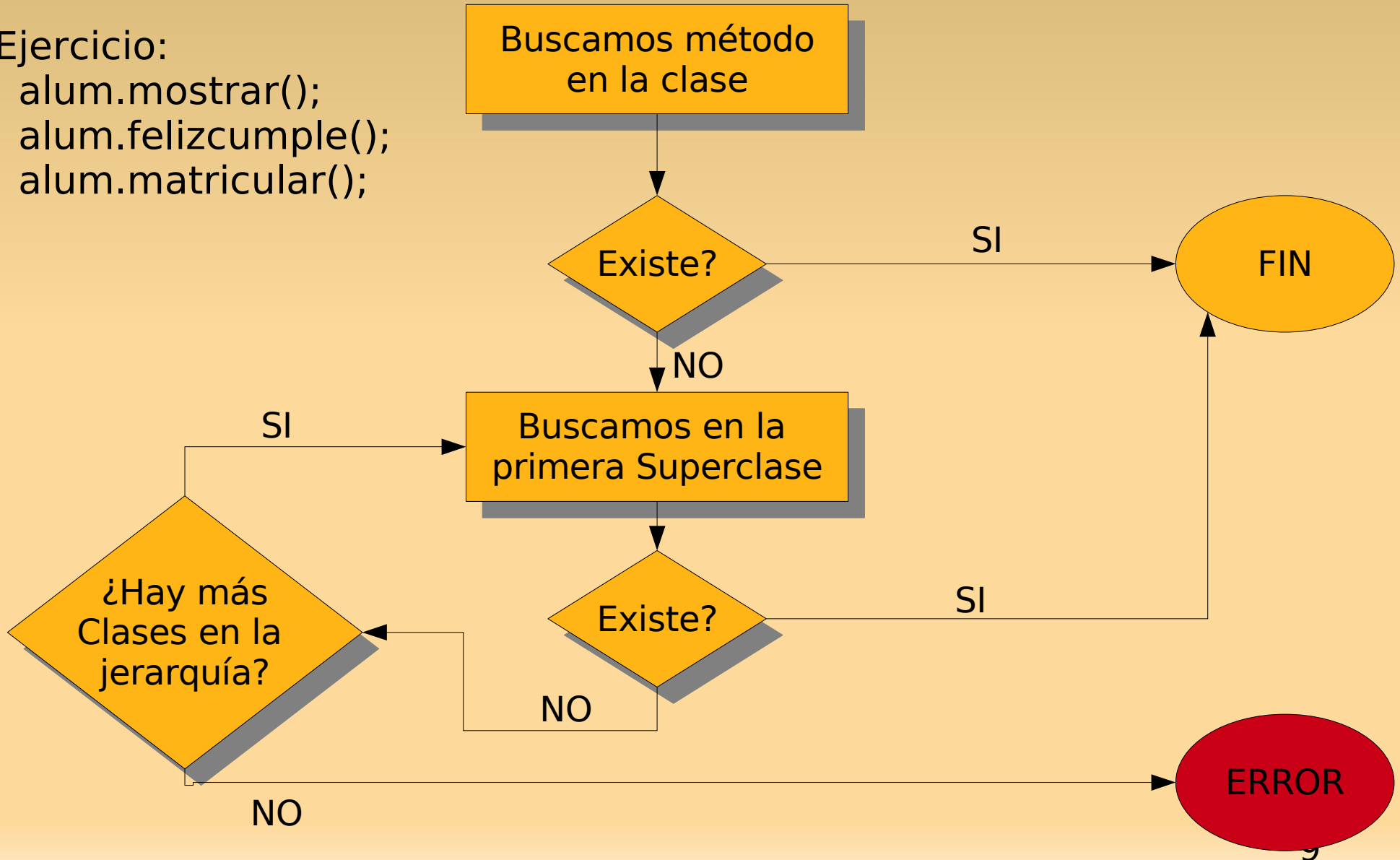
ClaseBase::operator=(d);

19.1 Repaso

Pasos de Mensaje con Herencia

Ejercicio:

```
alum.mostrar();  
alum.felizcumple();  
alum.matricular();
```



19.1 Repaso

Problemas con la Vinculación

```
void Persona :: felizcumple()
{
    edad++;
    cout << "Felicidades!!";
    mostrar();
}

int main()
{
    Alumno alum ("77777R", 20,
                "Ana" , "Ruiz", 3 );
    alum.felizcumple();
}
```

El método felizcumple() no está definido en Alumno.

Se busca en persona, y se encuentra.

El mensaje mostrar() se vincula con la clase Persona en lugar de con la clase Alumno.

La salida es:

```
Felicidades!!
77777R Nombre: Ana
```

(*) Nos falta el curso.

19.2 Herencia

Superposición y Sobrecarga

Ya hemos visto que puedo redefinir un método de una clase base en mi nueva clase derivada.

Superponemos el nuevo método sobre el existente.

Cuando se superpone una función, **se ocultan todas las funciones** con el mismo nombre en la clase base.

No accedemos a ninguna de las funciones superpuestas de la clase base, aunque tengan distinto número de parámetros, o éstos o el valor de retorno tengan distintos tipos.

Para acceder a los métodos de la clase base estaremos obligados a utilizar el nombre completo.

```
Clase Base::metodo_que_sea(....);
```

19.2 Herencia

Superposición por redefinición en la Herencia

```
class ClaseA
{
    public:
    void Incrementar()
    {
        cout << "Suma 1" << endl;
    }
    void Incrementar(int n)
    {
        cout << "Suma " << n << endl;
    }
};
```

```
Suma 2
Suma 1
Suma 10
```

```
class ClaseB : public ClaseA
{
    public:
    void Incrementar()
    {
        cout << "Suma 2" << endl;
    }
};
int main()
{
    ClaseB objeto;
    objeto.Incrementar();
    // objeto.Incrementar(10); ¿?
    objeto.ClaseA::Incrementar();
    objeto.ClaseA::Incrementar(10);
    cin.get();
    return 0;
}
```

19.2 Herencia

Superposición y Sobrecarga

Lo más frecuente es que la versión de la clase derivada llame a la versión de la clase base.

La versión de la clase base hace su parte de la nueva tarea.

En el caso de Alumno:

“Llamo a mostrar() de Persona y luego muestro las notas.”

```
Alumno::mostrar(void)
{
    Persona::mostrar();
    cout << "Nota :" << nota_ << endl;
}
```

¿Qué ocurre si no utilizamos el operador de **resolución de ámbito (::)**?

¡¡Recursión Infinita!!

19.2 Herencia

Conversión entre Objetos

Un objeto de la clase base también “ES UN” objeto de la clase derivada.

Pero: el tipo de la clase base y el tipo de la clase derivada son distintos.

Bajo herencia **public**:

Los objetos de la clase derivada se pueden tratar como si fueran de la base. La clase derivada tiene miembros que se corresponden con los de la base. Convertir de derivada a base es correcto y tiene sentido. No al revés.

Si tenemos un puntero de la clase base apuntando a un objeto de la clase derivada:

El puntero de la clase base sólo ve la parte de la clase base.

```
Alumno a;  
Persona p;  
.....  
p = a;  
a = p; // No!!
```

19.2 Herencia

Clases Abstractas

Una clase Abstracta es aquella que solo sirve como base de otras clases.

No se puede crear objetos de esa clase (no se debe).

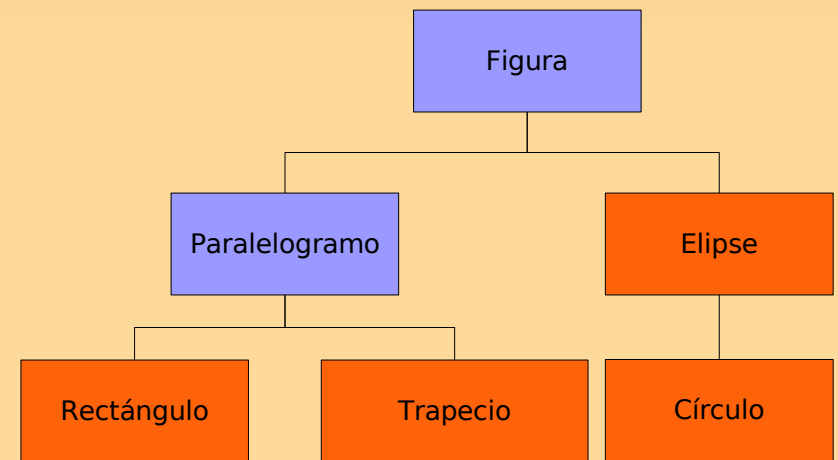
Normalmente se trata de clases que representan conceptos abstractos de los que no tiene sentido crear objetos.

No tiene sentido trabajar con “figuras” o “paralelogramos”.

Modela comportamiento común.

Implementa Métodos comunes.

Establece métodos comunes y obliga a que se implementen en las derivadas.



19.2 Herencia

Desarrollo Incremental

Es una ventaja de la Herencia y de la Composición.

- Podemos añadir código sin causar fallos en el código existente.
- Si hay fallos se rectifican en el nuevo código.
- Las clases – y el código - quedan separadas de forma limpia.
- Incluso sin tener el código fuente (Cabecera.h y código objeto).
- Permite crecer de forma evolutiva (me baso en lo anterior).

La herencia significa:

“Esta nueva clase que estoy creando es un tipo de esta clase antigua”

19.3 Herencia Práctica

Ejemplo

```
class Punto
{
    protected:
        int x;
        int y;
    public:
        Punto (int a=0, b=0);
        void setPunto(int,int);
        int getX() const {return x;};
        int getY() const {return y;};
        void mostrar() const;
};

Punto :: Punto(int a, int b)
{
    setPunto(a,b);
}
```

```
void Punto::setPunto(int a,int b)
{
    x=a;
    y=b;
}

void Punto::mostrar()
{
    cout<<"[" <<p.x<<", "<<p.y<< "]"<< endl;
}


```

Uso de Punto -----

```
Punto p( 72, 115 );
cout << "La coordenada de x es "
<< p.getX() << endl;
cout << "La coordenada de y es "
<< p.getY() << endl;
// No puedo:
// cout << "X es:" << p.x << endl;
p.mostrar();
```

19.3 Herencia práctica

Ejemplo

```
class Circulo : public Punto
{
    protected:
        double radio;
    public:
        Circulo (double r= 0.0, int x=0,
                int y=0);
        void setRadio(double);
        double getRadio() const;
        double area () const;
        void mostrar() const;
};

Circulo::Circulo(double r,int a,int
b) : Punto (a,b)
{
    setRadio(r);
}
```

```
void Circulo :: setRadio(double r)
{
    radio = (r >=0 ? r : 0);
}

double Circulo :: getRadio() const
{
    return radio;
}

double Circulo :: area () const
{
    return 3.14159 * radio * radio;
}

void Circulo :: mostrar () const
{
    Punto::mostrar();
    cout << "Radio =" << radio << endl;
}
```

19.3 Herencia práctica

Ejemplo

```
class Cilindro: public Circulo
{
    protected:
        double altura;
    public:
        Cilindro( double h=0.0,
double r=0.0, int x=0, int y=0);
        void setAltura (double);
        double daAltura () const ;
        double area () const ;
        double volumen () const;
        void mostrar() const;
}
Cilindro::Cilindro(double h, double
r, int x, int y): Circulo(r, x, y)
{
    setAltura( h );
}
```

```
void Cilindro::setAltura(double h)
{ altura=(h >=0 ? h :0);}
double Cilindro::daAltura() const
{
    return altura;
}
double Cilindro::area() const
{
    return 2*Circulo::area() + 2* 3.14159
* radio * altura;
}
double Cilindro::volumen () const
{
    return Circulo::area() * altura;
}
void Cilindro :: mostrar () const
{
    ?????
}
```

19.3 Herencia práctica

Ejemplo

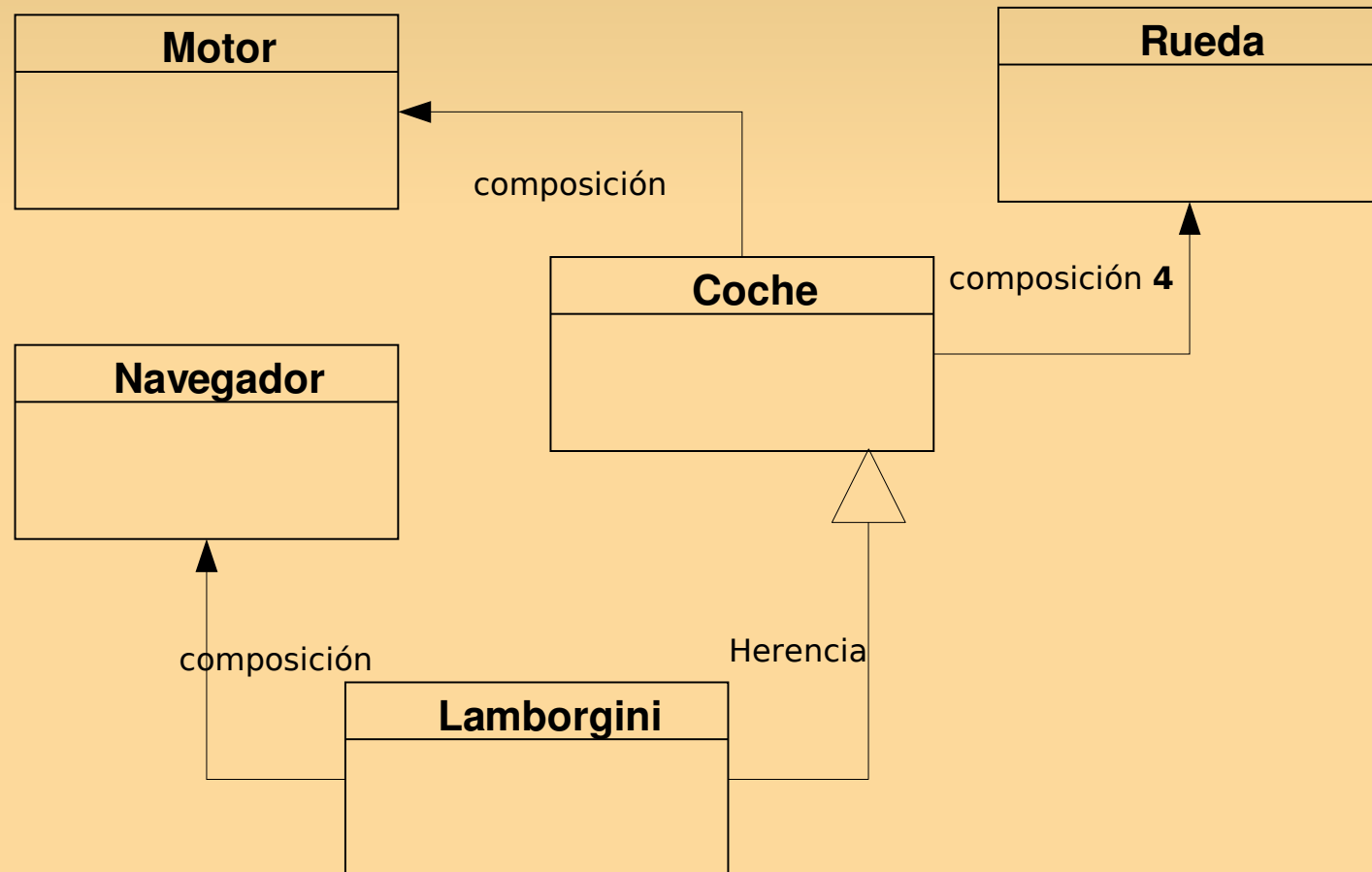
```
Cilindro cil (5.7, 2.5, 12, 23);
// Usamos metodos heredados...
cout<<"x es "<<cil.getX()<< endl;
cout<<"y es "<<cil.getY()<< endl;
cout<<"r es "<< cil.getRadio()<<endl;
cout<<"h es "<< cil.daAltura()<<endl;
// Usamos métodos no constantes.
cil.setAltura(10);
cil.setRadio(4.25);
cil.setPunto(2,2);
cout << "Los datos de cil son: \n";
cil.mostrar();

Point & pRef =cil;
//pRef "piensa" que es un punto
pRef.mostrar();
Circulo & cRef =cil;
//cRef "piensa" que es un circulo
cRef.mostrar();
```

19.3 Herencia práctica

Ejercicio

1.- Implementar esta jerarquía de clases sabiendo que todas las clases muestran por pantalla cuando se llama a su constructor por defecto y a su destructor. ¿Cuál es la salida del programa con ese main()?. Implementar mostrar() en todas las clases.



```
int main()
{
    Lamborgini l;

    l.mostrar();
}
```