

## Herencia (III)

### CONTENIDOS

1. Completar el ejemplo de Herencia:  
Superclase Persona-Subclase Alumno
2. Redefinición de métodos.
3. Jerarquía de clases.
4. Ejecución de los pasos de mensajes con Herencia.
5. Problemas con la vinculación de mensajes.
6. Funciones virtuales.
7. Clases Abstractas.

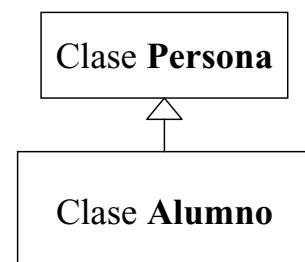
1

## Herencia (III)

### Ejemplo de Herencia

```
class Persona
{
  private:
    char * nif;
    int edad;
    char * nombre, *apellidos;
  public:
    Persona(char * , int = 0, char *, char * );
    Persona & operator=( Persona &);
    ~Persona(); // Destructor
    void medad(int);
    void mnombre(char *);
    char * mnombre() ;
    void mostrar() ;
    char * nombreCompleto() ;
    void felizCumple(); // El día del cumpleaños
    void leer(); // Lectura de los datos de la persona
};
```

```
Class Alumno : public Persona
{
  private:
    int curso;
  public:
    Alumno(char * , int = 0, char *, char * , int );
    Alumno& operator=( Alumno &);
    ~Alumno (); // Destructor
    int mcurso ();
    void mcurso (int );
};
```



2

## Herencia (III)

### Ejemplo de Herencia: implementación de los métodos

```
Class Alumno : public Persona
{
private:
int curso;
public:
Alumno(char * , int = 0, char * , char * , int );
Alumno& operator=( Alumno &);
~Alumno (); // Destructor
int mcurso ();
void mcurso (int );
};
```

```
Alumno & Alumno :: operator = ( Alumno a)
{
Persona :: operator = (a);
curso = a.curso;
return *this;
}
```

```
Alumno :: Alumno (char * n, int e, char * nom, char * ape , int c ) : Persona (n, e, nom, ape)
{
curso = c;
}
```

## Herencia (III)

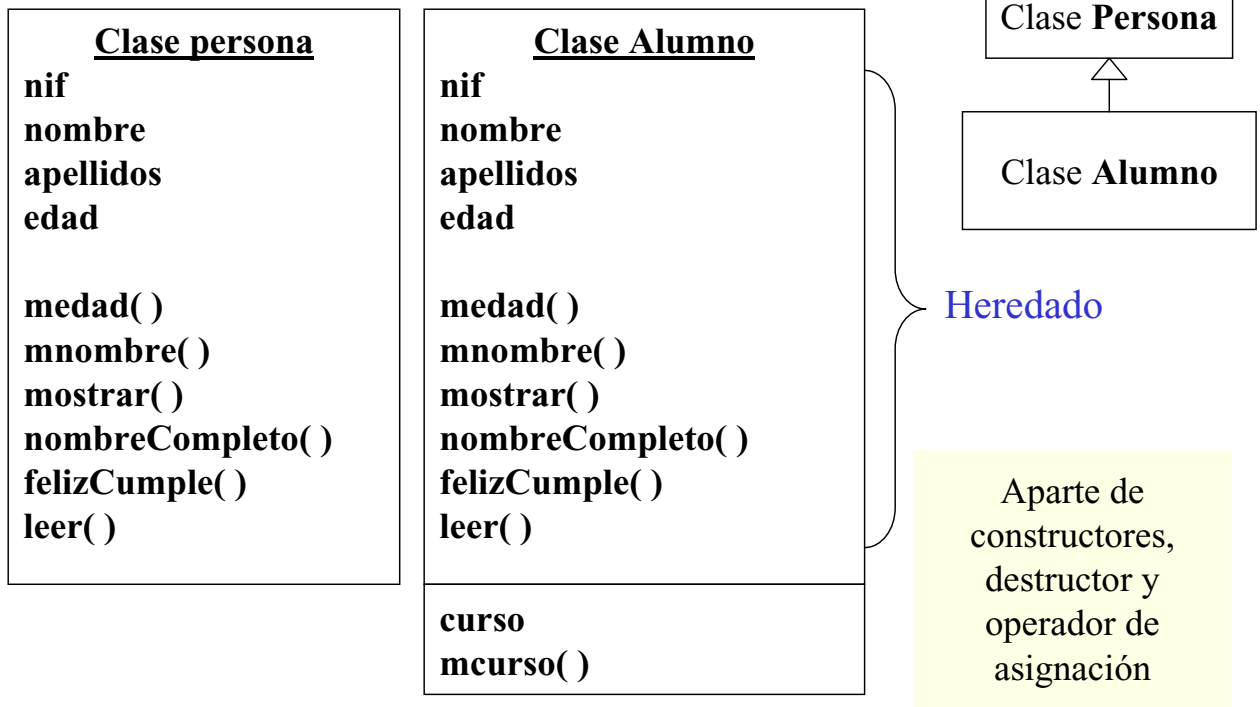
### Ejemplo de Herencia : Implementación de los métodos

```
Class Alumno : public Persona
{
private:
int curso;
public:
Alumno(char * , int = 0, char * , char * , int );
Alumno& operator=( Alumno &);
~Alumno (); // Destructor
int mcurso ();
void mcurso (int );
};
```

```
int Alumno :: mcurso ()
{
return curso;
}
```

```
void Alumno :: mcurso (int c)
{
curso = c ;
}
```

## ¿Cómo quedan las clases?



## Redefinición de métodos

En la clase derivada se puede redefinir algún método ya definido en la clase base: *redefinición o superposición de métodos*.

Para redefinir un método en la subclase, basta con declarar una función miembro con el mismo nombre.

```
class Persona
{
private:
...
public:
Persona(char * , int , char * , char * );
...
void mostrar() ;
...
};
```

El método **mostrar()** no resulta adecuado para Alumno, ya que no muestra el curso del alumno.

```
Class Alumno : public Persona
{
private:
int curso;
public:
...
void mostrar() ;
};
```

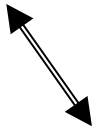
Redefinimos el método **mostrar()** en la clase Alumno

## Redefinición de métodos

Tenemos definido un método `mostrar()` para la clase **Alumno**, y será éste el que se ejecute cuando se le pase el mensaje `mostrar` a un objeto de la clase **Alumno**.

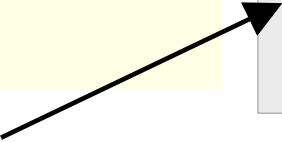
```
void Persona :: mostrar()
{
  cout << nif;
  cout << "Nombre: " << nombre;
}
```

```
void Alumno :: mostrar()
{
  cout << nif;
  cout << "Nombre: " << nombre;
  cout << "Curso: " << curso;
}
```



¿Se puede aprovechar el código de la superclase?  
En éste caso sí

```
void Alumno :: mostrar()
{
  Persona :: mostrar( );
  cout << "Curso: " << curso;
}
```



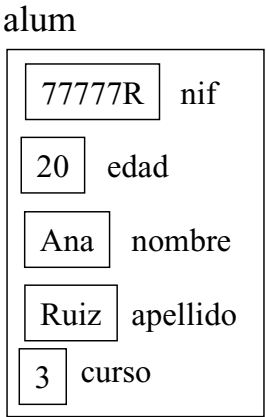
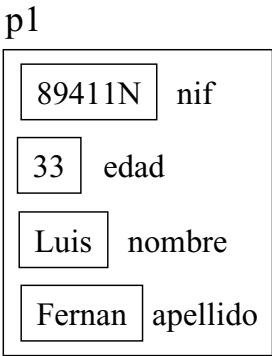
Para ello, podemos ejecutar dicho código en la subclase.

## Prueba de la clase Alumno y Persona

```
Alumno :: Alumno (char * n, int e, char * nom, char * ape , int c ) : Persona (n, e, nom, ape)
{
  curso = c;
}
```

```
void main()
{
  Persona p1("89411N", 33, "Luis", "Fernan");
  Alumno alum ("77777R", 20, "Ana", "Ruiz", 3 );
}
```

Se ejecuta el constructor de la clase Persona y posteriormente se ejecuta el constructor de la clase Alumno,



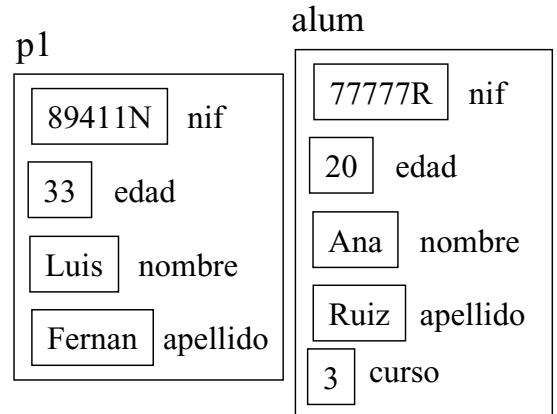
## Prueba de la clase Alumno y Persona

```
void Persona :: mostrar()  
{  
    cout << nif;  
    cout << "Nombre: " << nombre;  
}
```

```
void Alumno :: mostrar()  
{  
    Persona :: mostrar( );  
    cout << "Curso: " << curso;  
}
```

```
void main()  
{  
    Persona p1("89411N", 33, "Luis", "Fernan");  
    Alumno alum ("77777R", 20, "Ana", "Ruiz", 3);  
    p1.mostrar( );  
    alum .mostrar( );  
}
```

89411N Nombre: Luis  
77777R Nombre: Ana Curso: 3



## Jerarquía de Clases

A medida que se establecen relaciones de herencia entre clases, se va construyendo la jerarquía de clases del sistema.

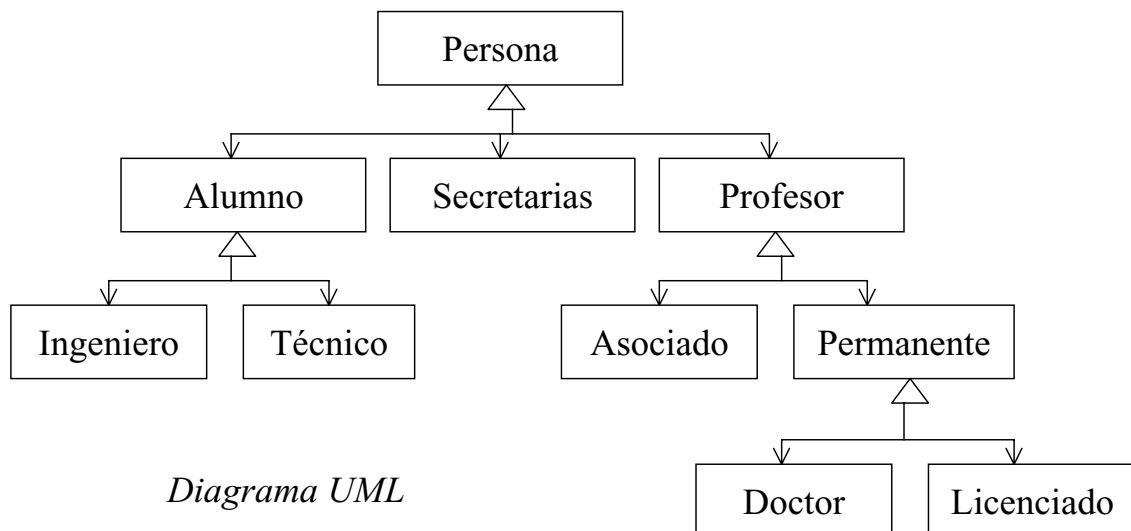
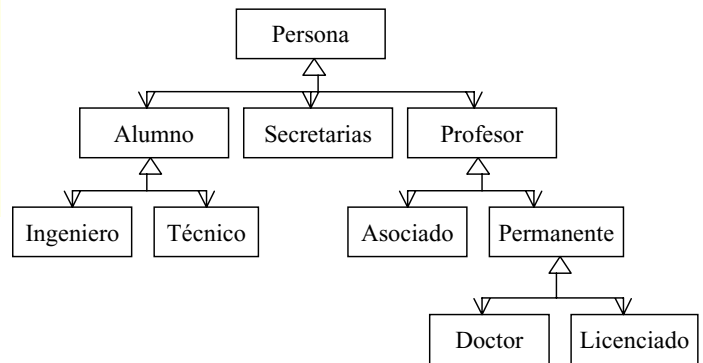


Diagrama UML

### Jerarquía de clases

Cuánto más arriba en la jerarquía, menor nivel de detalle.

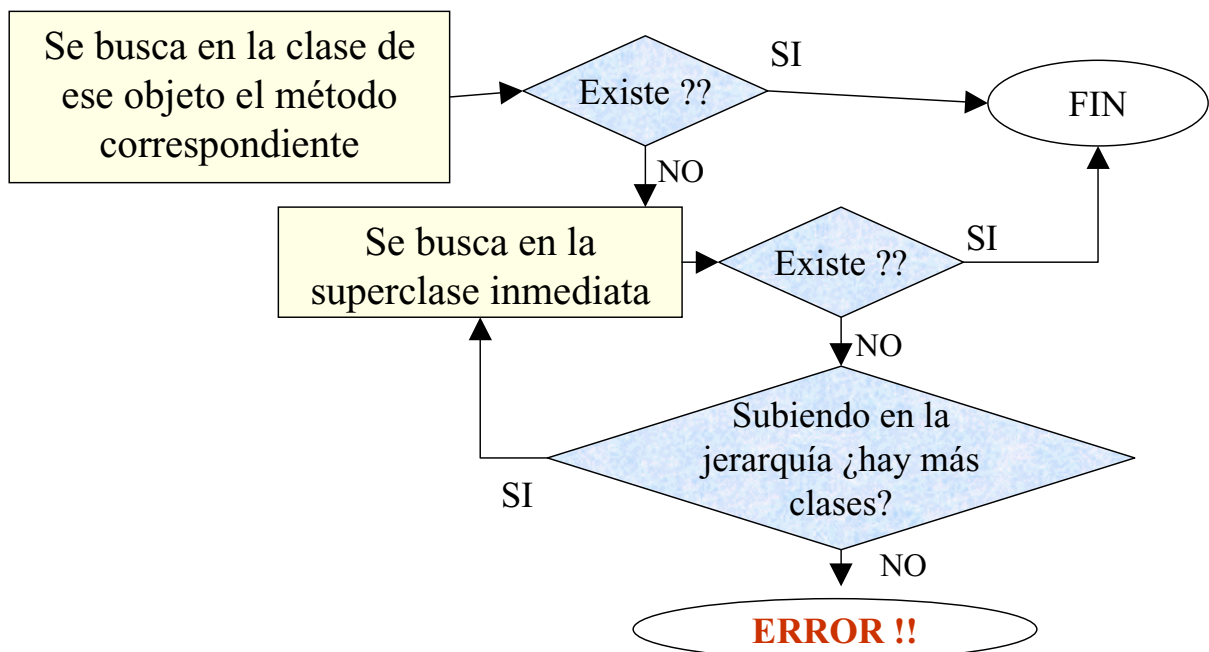
**Clase más general**



- ➔ Cada clase de la jerarquía, debe implementar todas las características que son comunes a todas sus subclases.
- ➔ Cada subclase debe contemplar únicamente las peculiaridades que la distinguen de su superclase.

### Ejecución de los pasos de mensajes con Herencia

Cuando a un objeto se le pasa un mensaje:



## Herencia (III)

### Ejecución de los pasos de mensajes con Herencia

```
void Persona :: mostrar()
{
    cout << nif;
    cout << "Nombre: " << nombre;
}
```

```
void Alumno :: mostrar()
{
    Persona :: mostrar( );
    cout << "Curso: " << curso;
}
```

```
void main()
{
    Alumno alum ("77777R", 20, "Ana", "Ruiz", 3 );

    alum . mostrar( );
}
```

El método `mostrar( )` está definido en la clase `Alumno`, por lo que se ejecuta dicho método.

El mensaje `mostrar( )` se vincula con el método `mostrar( )` de la clase `Alumno`.

## Herencia (III)

### Ejecución de los pasos de mensajes con Herencia

```
class Persona
{
    private:
        char * nif;
        int edad;
        char * nombre, *apellidos;
    public:
        ....
        void mostrar( );
        char * nombreCompleto( );
        void felizCumple( ); // El día del cumpleaños
        void leer();
};
```

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

El método `felizCumple( )` NO está definido en la clase `Alumno`, por lo que se busca en la superclase (`Persona`), se encuentra y se ejecuta dicho método.

El mensaje `felizCumple( )` se vincula con el método `felizCumple( )` de la clase `Persona`.

### Ejecución de los pasos de mensajes con Herencia

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

El método pasarCurso( ) NO está definido en la clase Alumno, NI en la superclase Persona, por lo que se puede decir que el objeto alum no entiende el mensaje.

El mensaje pasarCurso( ) no se puede vincular con ningún método.

**Error de compilación** (vinculación estática).

### Tipos de vinculación

Vinculación estática: se trata del intento de vincular el mensaje con el método correspondiente en tiempo de compilación.

(Si se produce error de vinculación, será en tiempo de compilación)

Vinculación dinámica: la vinculación entre mensaje y método se realiza en tiempo de ejecución.

(Si se produce error de vinculación, será en tiempo de ejecución)

### Problemas con la vinculación

```
void Persona :: felizCumple()
{
    edad ++ ;
    cout << " ¡¡¡ FELICIDADES !!!" ;
    mostrar ( ) ;
}
```

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

```
¡¡¡ FELICIDADES !!!
77777R Nombre: Ana
```

Incrementa el atributo edad y muestra por pantalla información del objeto.

El método `felizCumple( )` NO está definido en la clase `Alumno`, por lo que se busca en la superclase (`Persona`), se encuentra y se ejecuta dicho método.

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

El objeto que recibe el mensaje es de la subclase `Alumno`.

No aparece el curso

### Solución a los problemas con la vinculación

Para solucionar el problema anterior, C++ propone el uso de **funciones virtuales** que aseguren que se sigue la pista en todo momento a la identidad del objeto receptor del mensaje.

Para establecer un método como virtual, basta con escribir la palabra `virtual` delante del prototipo del método:

```
class Persona
{
    ...
    virtual void mostrar() ;
    ....
}
```

Los métodos virtuales se encuentran en las superclases.

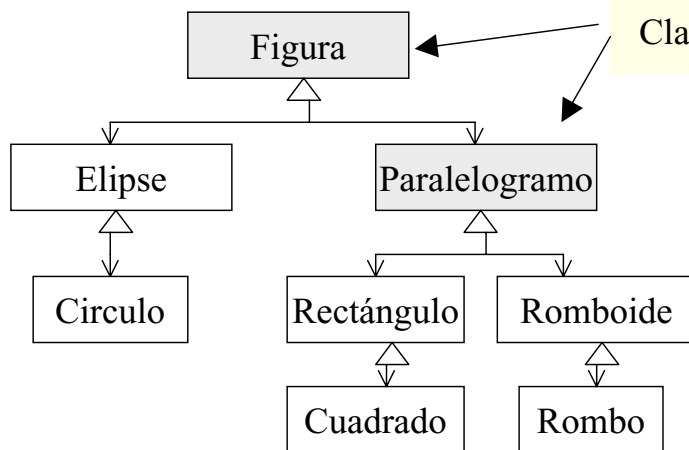
Si la clase `Alumno` no tiene subclases, no es necesario que el método `mostrar()` sea virtual.

`virtual` solo se pone en el prototipo.

### 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.



Clases Abstractas

Solo tiene sentido dibujar y trabajar con objetos concretos:

Elipse, Círculo,  
Rectángulo,  
Cuadrado, Romboide  
Rombo.

### Clases Abstractas

Una clase Abstracta:

- ❁ Modela el comportamiento común a las clases derivadas.
- ❁ Implementa métodos que son comunes a las clases derivadas.
- ❁ Establece métodos comunes a todas las subclasses pero cuya implementación corresponde a las subclasses.

### ¿Cómo se define una clase abstracta?

Basta con declarar un **método virtual puro**. Esto es un método virtual que no se implementa en la superclase.

```
virtual void mostrar() = 0;
```

El prototipo del método termina con =0

# Herencia (III)

## Ejemplo:

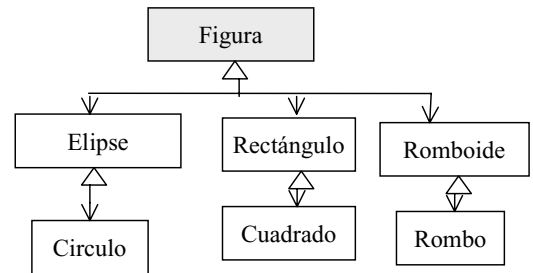
Abstracta

```
class Figura  
{  
  public:  
    void dibujar()  
    { cout << perimetro( ); }  
    virtual float area () = 0;  
    virtual float perimetro()= 0;  
};
```

Métodos virtuales puros

```
class Cuadrado : public Rectangulo  
{  
  public:  
    float area ();  
    float perimetro();  
    { return base*4; }  
};
```

```
class Rectangulo: public Figura  
{  
  protected:  
    Punto p;  
    float base, altura;  
  public:  
    float area ();  
    float perimetro();  
    { return 2*base +2*altura; }  
};
```



# Herencia (III)

## Ejemplo:

Abstracta

```
class Figura  
{  
  public:  
    void dibujar()  
    { cout << perimetro( ); }  
    virtual float area () = 0;  
    virtual float perimetro()= 0;  
};
```

```
class Cuadrado : public Rectangulo  
{  
  public:  
    float area ();  
    float perimetro();  
    { return base*4; }  
};
```

```
class Rectangulo: public Figura  
{  
  protected:  
    Punto p;  
    float base, altura;  
  public:  
    float area ();  
    float perimetro();  
    { return 2*base +2*altura; }  
};
```

```
Cuadrado c(4);  
c. dibujar();
```

