

# Programación Orientada a Objetos

## Tema 3. Relaciones entre clases. Parte II

José Fidel Argudo Argudo    Francisco Palomo Lozano  
Inmaculada Medina Buló    Gerardo Aburrizaga García



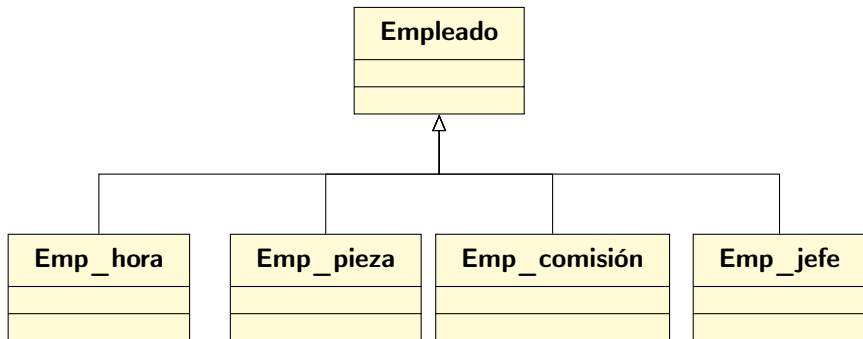
Versión 1.0



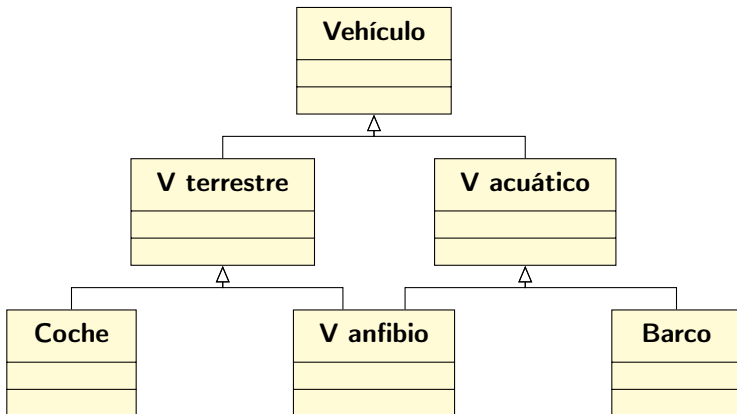
# Índice

- 1 Generalizaciones y especializaciones
- 2 Interfaces e implementaciones

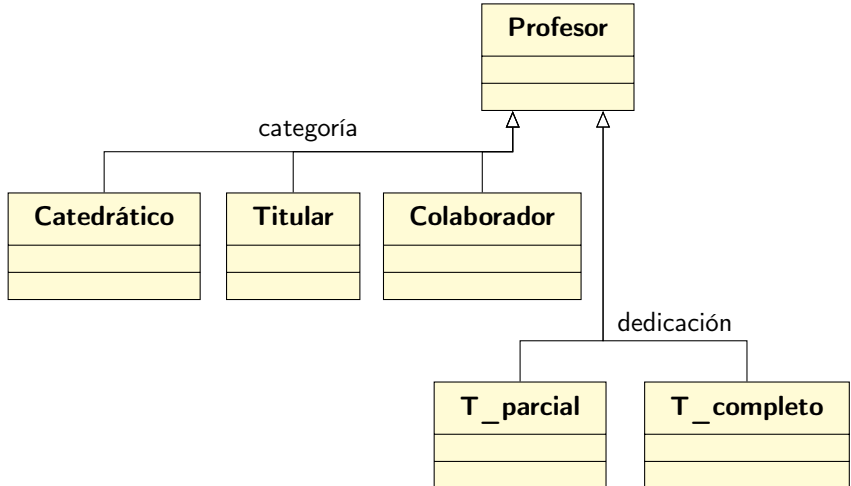
# Generalización



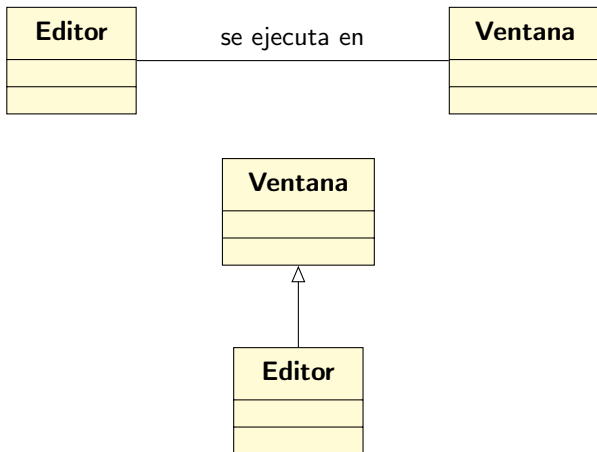
# Generalización múltiple



# Generalización según criterios independientes




# Asociación vs. generalización



# Herencia

## Lista de derivacion

```
class clase-derivada: [accesibilidad]  base-base  
{  
    // declaraciones de miembros  
};
```

Se heredan todos los miembros de la clase base menos:

- Constructores
- Destructor
- Operadores de asignación

# Herencia

## Ejemplo

```
1 class Base {
2   public:
3     int publico;
4   protected:
5     int protegido;
6   private:
7     int privado;
8 };

10 class DerivadaPublica: public Base { /* ... */ };
11 class DerivadaProtegida: protected Base { /* ... */ };
12 class DerivadaPrivada: private Base { /* ... */ };
```



# Herencia: Modos de acceso a los miembros heredados

<u>Accesibilidad</u>	un miembro... de la clase base	pasa a ser... en la derivada
<u>public</u> (por defecto en <u>struct</u> )	público protegido privado	→ público → protegido → inaccesible
<u>protected</u>	público protegido privado	→ protegido → protegido → inaccesible
<u>private</u> (por defecto en <u>class</u> )	público protegido privado	→ privado → privado → inaccesible

# Especialización y delegación de operaciones

```
1  #ifndef LISTA_H_
2  #define LISTA_H_
3  #include <deque>

5  class Lista {
6  public:
7      bool vacia() const;
8      int primero() const;
9      int ultimo() const;
10     void insertarPrincipio(int e);
11     void insertarFinal(int e);
12     void eliminarPrimero();
13     void eliminarUltimo();
14     void mostrar() const;
15 private:
16     std::deque<int> l;
17 };

19 #endif // LISTA_H_
```

# Especialización y delegación de operaciones

```
1  #ifndef PILA_H_
2  #define PILA_H_
3  #include "lista.h"
4  class Pila: private Lista {
5  public:
6      bool vacia() const;
7      int cima() const;
8      void apilar(int e);
9      void desapilar();
10     void mostrar() const;
11 };
12 // Delegación de operaciones
13 inline bool Pila::vacia() const { return Lista::vacia(); }
14 inline int Pila::cima() const { return primero(); }
15 inline void Pila::apilar(int e) { insertarPrincipio(e); }
16 inline void Pila::desapilar() { eliminarPrimero(); }
17 inline void Pila::mostrar() const { Lista::mostrar(); }
18 #endif // PILA_H_
```

# Especialización y delegación de operaciones

```
1  #ifndef VECTOR_H
2  #define VECTOR_H
3  class Vector {
4  public:
5      using T = double;
6      explicit Vector(int n);
7      Vector(const Vector& V);
8      Vector& operator =(const Vector& V);
9      Vector(Vector&& V);
10     Vector& operator =(Vector&& V);
11     ~Vector();
12     T& operator [](int i);
13     const T& operator [](int i) const;
14     int longitud() const;
15     void mostrar() const;
16 protected:
17     int n;    // longitud
18     T* v;    // ptr. al primer elto.
19 };
20 #endif    // VECTOR_H
```

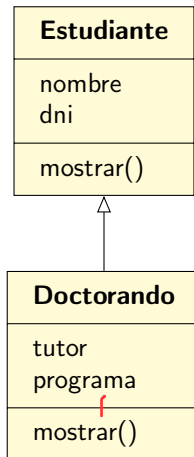
# Especialización y delegación de operaciones

```
1 #ifndef SUPERVECTOR_H
2 #define SUPERVECTOR_H
3 #include "Vector.h"
4 class SuperVector: private Vector {
5 public:
6     explicit SuperVector(int i = 0, int s = 0) :
7         Vector{s - i + 1}, inf{i}, sup{s} {}
8     T& operator [](int i) {
9         assert(i >= inf && i <= sup);
10        return v[i- inf]; // return Vector::operator [] (i-inf);
11    }
12    const T& operator [](int i) const {
13        assert(i >= inf && i <= sup);
14        return v[i- inf];
15    }
```

# Especialización y delegación de operaciones

```
16  int limiteInferior() const { return inf; }
17  int limiteSuperior() const { return sup; }
18  int longitud() const { return Vector::longitud(); }
19  void mostrar() const { Vector::mostrar(); }
20 private:
21  int inf,    // límite inferior
22      sup;    // límite superior
23 };
24 #endif // SUPERVECTOR_H
```

# Especialización de Estudiante en Doctorando



# Especialización de Estudiante en Doctorando (estudiante.h)

```
1  #ifndef ESTUDIANTE_H_
2  #define ESTUDIANTE_H_

4  #include <iostream>
5  #include <string>
6  using namespace std;

8  class Estudiante {
9  public:
10     Estudiante(string nombre, int dni);
11     void mostrar() const;
12 protected:
13     string nombre; // nombre completo
14     int dni;       // DNI
15     // ...
16 };


18 #endif
```



# Especialización de Estudiante en Doctorando (doctorando.h)

```
1  #ifndef DOCTORANDO_H_
2  #define DOCTORANDO_H_

4  #include "estudiante.h"
5  #include <string>
6  using namespace std;

8  class Doctorando: public Estudiante {
9  public:
10     Doctorando(string nombre, int dni, string tutor, int programa);
11     void mostrar() const;
12 protected:
13     string tu  // tutor en el programa de doctorado
14     int programa; // código del programa
15     // ...
16 };

18 #endif
```

# Especialización Estudiante–Doctorando (doctorando.cpp)

```
1  #include "doctorando.h"
2  #include <iostream>
3  #include <string>
4  using namespace std;

6  Doctorando::
7  Doctorando(string nombre, int dni, string tutor, int programa):
8  Estudiante(nombre, dni), tutor(tutor), programa(programa) {}

10 void Doctorando::mostrar() const
11 {
12     // Muestra los datos que posee como estudiante
13     Estudiante::mostrar();
14     // Y los específicos de su condición de doctorando
15     cout << "Programa de doctorado: " << programa << "\n"
16           << "Tutor en el programa: " << tutor << endl;
17 }
```

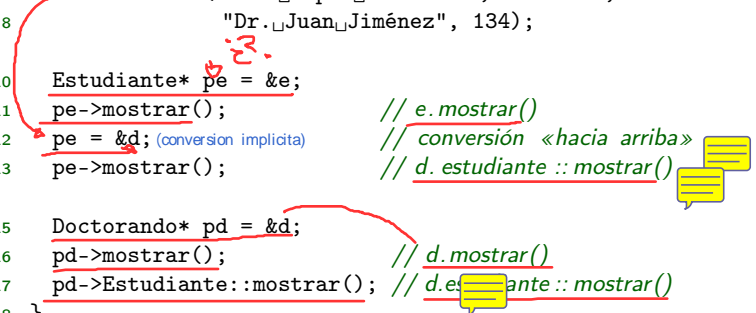
# Especialización de Estudiante en Doctorando (prueba.cpp)

```
1  #include "estudiante.h"
2  #include "doctorando.h"

4  int main()
5  {
6      Estudiante e("María_Pérez_Sánchez", 31682034);
7      Doctorando d("José_López_González", 32456790,
8                  "Dr._Juan_Jiménez", 134);

10     Estudiante* pe = &e;
11     pe->mostrar(); // e.mostrar()
12     pe = &d; (conversion implícita) // conversión «hacia arriba»
13     pe->mostrar(); // d.estudiante :: mostrar()

15     Doctorando* pd = &d;
16     pd->mostrar(); // d.mostrar()
17     pd->Estudiante::mostrar(); // d.estudiante :: mostrar()
18 }
```



# Especialización Estudiante–Doctorando (conversiones.cpp)

```
1  #include "estudiante.h"
2  #include "doctorando.h"

4  int main()
5  {
6      Estudiante e("María_Pérez_Sánchez", 31682034), *pe;
7      Doctorando d("José_López_González", 32456790,
8                  "Dr._Juan_Jiménez", 134), *pd;

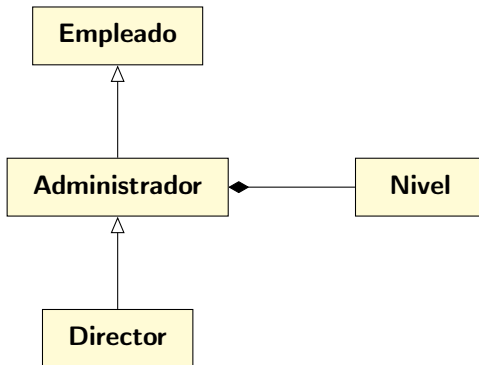
10     pe = &d; // bien
11     pd = pe; // ERROR
12     pd = static_cast<Doctorando*>(pe); // bien
13     e = d; // bien
14     d = e; // ERROR
15     d = Doctorando(e); // ERROR
16     d = static_cast<Doctorando>(e); // ERROR
17     d = reinterpret_cast<Doctorando>(e); // ERROR
18 }
```

Reinterpreta lo que hay almacenado en e como si fuera un doctorando

conversion  
hacia abajo  
prohibida

# Jerarquía de clases

- La herencia simple permite definir una jerarquía de clases relacionadas, en la que cada una se deriva de una sola clase base. La cadena de derivación no puede ser circular.



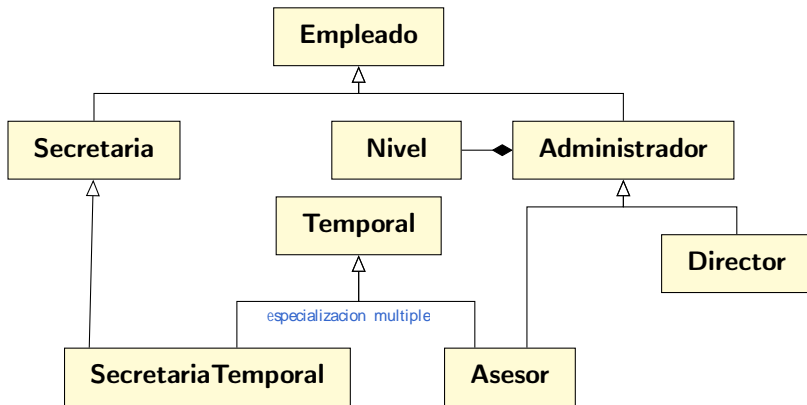
# Jerarquía de clases

```
1 class Empleado {  
2     // ...  
3 };  
  
5 class Administrador: public Empleado {  
6     protected  
7     Nivel nivel;  
8     // ...  
9 };  
  
11 class Director: public Administrador {  
12     // ...  
13 };
```

The diagram illustrates a class hierarchy. A red arrow points from the 'Empleado' class to the 'public Empleado' access specifier in the 'Administrador' class definition. Another red arrow points from the 'protected' access specifier in the 'Administrador' class to the 'public Administrador' access specifier in the 'Director' class definition. Additionally, the 'protected' keyword and the 'Nivel nivel;' line in the 'Administrador' class are underlined in red.

Queremos tener acceso desde director a Nivel de Administrador

# Herencia múltiple



# Herencia múltiple

```
1 class Temporal {  
2     // ...  
3 };  
  
5 class Secretaria: public Empleado {  
6     // ...  
7 };  
  
9 class SecretariaTemporal: public Temporal, public Secretaria {  
10    // ...  
11 };  
  
13 class Asesor: public Temporal, public Administrador {  
14    // ...  
15 };
```

The diagram illustrates multiple inheritance with red annotations. A red circle highlights 'public Empleado' in the 'Secretaria' class definition. A red line connects this circle to the 'public Secretaria' part of the 'SecretariaTemporal' class definition. Another red line connects the 'public Empleado' circle to the 'public Temporal' part of the 'Asesor' class definition. The 'SecretariaTemporal' and 'Asesor' class definitions are also underlined in red.



# Herencia múltiple: Orden de inicialización

- Un objeto de una clase derivada mediante herencia múltiple se inicializa ejecutando los constructores en el siguiente orden:
  - 1 Constructores de las clases bases en el orden en que han sido declaradas en la lista de derivación.
  - 2 Constructores de los atributos en el orden en que aparecen declarados dentro de la clase (independientemente del orden de la lista de inicialización del constructor de la clase derivada).
  - 3 Por último se ejecuta el constructor de la clase derivada.
- Los destructores se ejecutan en orden inverso.

# Herencia múltiple: Orden de inicialización

## Ejemplo: Orden de constructores para definir un objeto Asesor

- 1 Temporal()  
Empleado(), por ser clase base de Administrador.  
Administrador(), que llamará a Nivel().
- 2 Constructores de los atributos de Asesor.
- 3 Asesor()

# Herencia múltiple: Ambigüedad al heredar miembros con nombres iguales

```
1  #include <iostream>

3  class B1 {
4  public:
5      void f() { std::cout << "B1::f()" << std::endl; }
6      int b;
7      // ...
8  };

10 class B2 {
11 public:
12     void f() { std::cout << "B2::f()" << std::endl; }
13     int b;
14     // ...
15 };

17 class D: public B1, public B2 {
18     // ...
19 };
```

# Herencia múltiple: Ambigüedad al heredar miembros con nombres iguales

```
21 int main()
22 {
23     D d;
24     d.f();           // ERROR, ¿qué f(), el de B1 o el de B2?
25     d.b = 0;        // ERROR, ¿qué «b», el de B1 o el de B2?
26     d.B1::f();      // bien
27     d.B2::f();      // bien
28     d.B1::b = 0;    // bien
29     d.B2::b = 0;    // bien
30 }
```

# Herencia múltiple: Ambigüedad al heredar miembros sobrecargados

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

4  class B1 {
5  public:
6      void f(int i) { cout << "B1::f(int)" << endl; }
7      // ...
8  };

10 class B2 {
11 public:
12     void f(double d) { cout << "B2::f(double)" << endl; }
13     // ...
14 };

16 class D: public B1, public B2 {
17     // ...
18 };
```

# Herencia múltiple: Ambigüedad al heredar miembros sobrecargados

```
20 int main()
21 {
22     D d;
23     d.f(0);           // ERROR, ¿qué f(), el de B1 o el de B2?
24     d.f(0.0);        // ERROR, ¿qué f(), el de B1 o el de B2?
25     d.B1::f(0);       // bien, B1::f()
26     d.B2::f(0.0);     // bien, B2::f()
27 }
```

# Herencia múltiple: Resolución de sobrecarga

```
1  #include <iostream>

3  class B1 {
4  public:
5      void f(char i) { std::cout << "B1::f(char)" << std::endl; }
6      // ...
7  };

9  class B2 {
10 public:
11     void f(int d) { std::cout << "B2::f(int)" << std::endl; }
12     // ...
13 };

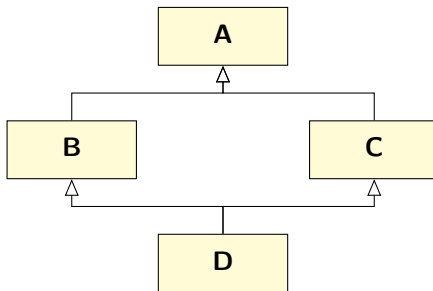
15 class D: public B1, public B2 {
16 public:
17     using B1::f; using B2::f;
18     void f(double c) { std::cout << "D::f(double)" << std::endl; }
19     // ...
20 };
```

# Herencia múltiple: Resolución de sobrecarga

```
22 int main()
23 {
24     D d;
25     d.f('A'); // B1::f(char)
26     d.f(0);   // B2::f(int)
27     d.f(0.0); // D::f(double)
28 }
```



# Herencia múltiple: Ambigüedad por herencia duplicada



```
1 struct A { int a; };  
2 struct B: A { int b; };  
3 struct C: A { int c; };  
4 struct D: B, C { int d; };
```

```
6 D d;  
7 d.a = 0;    // ERROR, ambigüedad  
8 d.B::a = 0; // bien, resolución de la ambigüedad  
9 d.C::a = 0; // bien, resolución de la ambigüedad
```

# Herencia virtual: Supresión de herencia duplicada

```
1 struct A { int a; };  
2 struct B: virtual A { int b; };  
3 struct C: virtual A { int c; };  
4 struct D: B, C { int d; };  
  
6 D d;  
7 d.a = 0;      // bien, d sólo tiene un atributo a
```

# Realización

