# INFORMÁTICA INDUSTRIAL

## Herencia en C++.

M. Abderrahim, A. Castro, J. C. Castillo
Departamento de Ingeniería de Sistemas y Automática

uc3m | Universidad Carlos III de Madrid

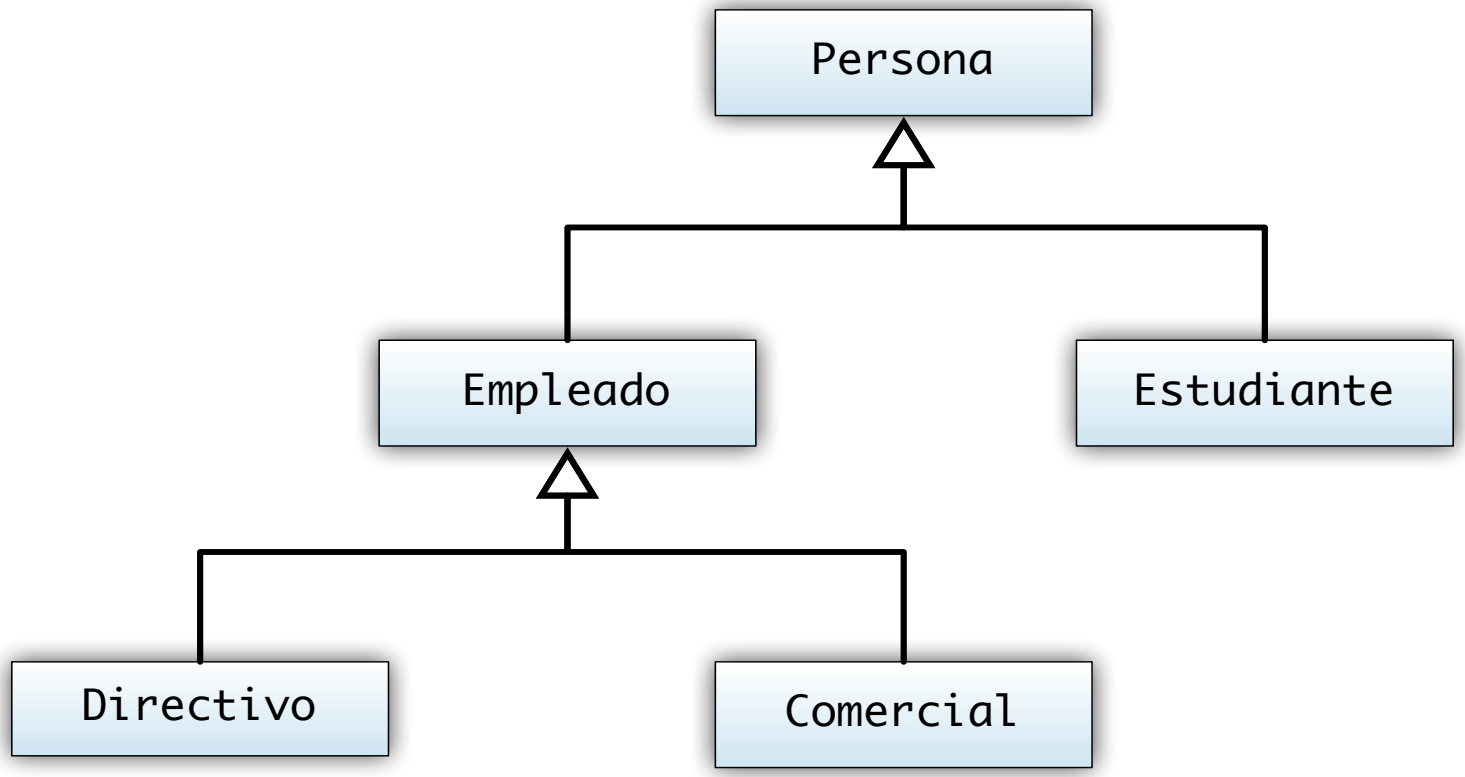
# AGENDA

- Herencia
- Constructores de las Clases Derivadas
- Destructores de las Clases Derivadas
- Herencia Múltiple
- Clases Base Virtuales

# Herencia

- Una de las principales propiedades de las clases es la *herencia*.
- Esta propiedad nos permite crear nuevas clases a partir de clases existentes, conservando las propiedades de la clase original y añadiendo otras nuevas.
- Cada nueva clase obtenida mediante herencia se conoce como *clase derivada o hija*, y las clases a partir de las cuales se deriva, *clases base o padre.*
- Además, cada clase derivada puede usarse como clase base para obtener una nueva clase derivada.

# Ejemplo de herencia



# Herencia

- La forma general de declarar clases derivadas es la siguiente:

```
class <nombre clase derivada> :
    [public|protected|private] <nombre clase base 1> [,[public|private]
<nombre clase base 2>] {};
```

  - Para cada clase base podemos definir tres tipos de acceso, **public, protected** o **private**. Si no se especifica ninguno, por defecto se asume que es **private**.
    - **public**: los miembros heredados de la clase base conservan el tipo de acceso con que fueron declarados en ella.
    - **protected**: los miembros public y protected se heredan como protected
    - **private**: todos los miembros heredados de la clase base pasan a ser miembros privados en la clase derivada.
  - **OJO**: en cualquier caso los miembros privados en la clase base no podrán ser accedidos desde la clase derivada

# Ejemplo de herencia en C++

```
// Clase base Persona:
class Persona {
public:
      Persona(char *n, int e);
      const char *LeerNombre(char *n) const;
      int LeerEdad() const;
      void CambiarNombre(const char *n);
      void CambiarEdad(int e);
protected:
      char nombre[40];
      int edad;
};
```

```
// Clase derivada Empleado:
class Empleado : public Persona {
public:
     Empleado(char *n, int e, float s);
     float LeerSalario() const;
     void CambiarSalario(const float s);
protected:
     float salarioAnual;
};
```

# Miembros *Protected*

- En general es recomendable declarar siempre los datos de nuestras clases como privados.
- En el ejemplo anterior se han declarado los datos miembros de las clases como *protected*.
- Este tipo de dato es privado para todas aquellas clases que no son derivadas; una clase derivada de la clase en la que se ha definido la variable como *protected* puede acceder a él.

# Constructor Clases Derivadas

- Un objeto de la clase derivada contiene todos los miembros de la clase base y todos esos miembros deben ser inicializados.

- Por esta razón el constructor de la clase derivada debe llamar al constructor de la clase base.

# Ejemplo de contructor

```
#include <iostream>
using namespace std;
class ClaseA {
public:
      ClaseA() : datoA(10) { cout << "Constructor de A" << endl; }
      int LeerA() const { return datoA; }
protected:
      int datoA;
};
class ClaseB : public ClaseA {
public:
      ClaseB() : datoB(20) { cout << "Constructor de B" << endl; }
      int LeerB() const { return datoB; }
protected:
      int datoB;
};
```

```
int main() {
      ClaseB objeto;
      cout      << "a = " << objeto.LeerA()
        << ", b = " << objeto.LeerB()
        << endl;
      return 0;
}
```

```
WJRG-MacBookPro:Ejemplos wladimir$ g++ ClaseA.cpp -o ClaseA
WJRG-MacBookPro:Ejemplos wladimir$ ./ClaseA
Constructor de A
Constructor de B
a = 10, b = 20
WJRG-MacBookPro:Ejemplos wladimir$
```

# Inicialización de la clase base

- Cuando queremos inicializar las clases base usando parámetros desde el constructor de una clase derivada.
- Se usará el constructor de la clase base con los parámetros adecuados.
- Las llamadas a los constructores deben escribirse antes de las inicializaciones de los parámetros.
- Sintaxis:

```
<nombre clase derivada>(<lista de parámetros>) :
    <nombre clase base>(<lista de parámetros>)
{  }
```

# Ejemplo de inicialización de la clase base

```
#include <iostream>
using namespace std;
class ClaseA {
public:
      ClaseA(int a) : datoA(a) { // constructor con parámetro
        cout << "Constructor de A" << endl;
      }
      int LeerA() const { return datoA; }
protected:
      int datoA;
};
class ClaseB : public ClaseA {
public:
      ClaseB(int a, int b) : ClaseA(a), datoB(b) { // con parámetros
        cout << "Constructor de B" << endl;
      }
      int LeerB() const { return datoB; }
protected:
      int datoB;
};
```

```
int main() {
      ClaseB objeto(5,15);
      cout      << "a = " << objeto.LeerA()
        << ", b = " << objeto.LeerB()
        << endl;
      return 0;
}
```

```
WJRG-MacBookPro:Ejemplos wladimir$ g++ ClaseA2.cpp -o ClaseA2
WJRG-MacBookPro:Ejemplos wladimir$ ./ClaseA2
Constructor de A
Constructor de B
a = 5, b = 15
```

# Destructores de clases derivadas

- Cuando se destruye un objeto de una clase derivada, primero se invoca al destructor de la clase derivada, si existen objetos miembro a continuación se invoca a sus destructores y finalmente al destructor de la clase o clases base. Si la clase base es a su vez una clase derivada, el proceso se repite recursivamente.

- Al igual que pasaba con los constructores, si no hemos definido los destructores de las clases, se **usan los destructores por defecto** que crea el compilador.

# Ejemplo de destructor de clase derivada

```
#include <iostream>
using namespace
std; class ClaseA {
public:
      ClaseA() : datoA(10) { // constructor sin parámetros
        cout << "Constructor de A" << endl;
      }
      ~ClaseA() { cout << "Destructor de A" << endl; }
      int LeerA() const { return datoA; }
protected:
      int datoA;
};
class ClaseB : public ClaseA {
public:
      ClaseB() : datoB(20) { cout << "Constructor de B" << endl; }
      ~ClaseB() { cout << "Destructor de B" << endl; }
      int LeerB() const { return datoB; }
protected:
      int datoB;
};
```

```
int main() {
      ClaseB objeto(5,15);
      cout      << "a = " << objeto.LeerA()
        << ", b = " << objeto.LeerB()
        << endl;
      return 0;
}
```

```
WJRG-MacBookPro:Ejemplos wladimir$ g++ ClaseA3.cpp -o ClaseA3
WJRG-MacBookPro:Ejemplos wladimir$ ./ClaseA3
Constructor de A
Constructor de B
a = 10, b = 20
Destructor de B
Destructor de A
```

# Herencia múltiple

- Una clase puede heredar variables y funciones miembro de **una o mas clases bases** al mismo tiempo
- En el caso que herede miembros de varias clases bases se trata de un caso de herencia múltiple.
- Sintaxis de un constructor de la clase hija:

```
<nombre clase derivada>(<lista de parámetros>) :
   <nombre clase base1>(<lista de parámetros>)
   [, <nombre clase base2>(<lista de parámetros>)]
{}
```

# Ejemplo de herencia múltiple

```
#include <iostream>
using namespace std;
class ClaseA {
public:
        ClaseA() : valorA(10) {}
        int LeerValor() const { return valorA; }
protected:
        int valorA;
};
class ClaseB {
public:
        ClaseB() : valorB(20) {}
        int LeerValor() const { return valorB; }
protected:
        int valorB;
};
class ClaseC : public ClaseA, public ClaseB {
public:
        int LeerValor() const { return ClaseA::LeerValor(); }
};
```

```
int main() {
        ClaseC CC;
        cout      << CC.LeerValor()
          << endl;
        return 0;
}
```

```
WJRG-MacBookPro:Ejemplos wladimir$ g++ ClaseA4.cpp -o ClaseA4
WJRG-MacBookPro:Ejemplos wladimir$ ./ClaseA4
10
```

# Herencia múltiple

- Análogamente a lo que sucedía con la herencia simple, en el caso de herencia múltiple el constructor de la clase derivada deberá llamar a los constructores de las clases base cuando sea necesario.
- En general, cuando se utiliza una lista de clases base, los constructores se deben llamar en orden de izquierda a derecha.
- Los destructores se deben llamar de derecha a izquierda

# Ejemplo de herencia múltiple

```
class ClaseA {
   public:
     ClaseA() : valorA(10) {} //constructor sin parámetros
     ClaseA(int va) : valorA(va) {} // constructor con parámetro
     int LeerValor() const { return valorA; }
   protected:
     int valorA;
};
```

```
class ClaseB {
   public:
     ClaseB() : valorB(20) {}
     ClaseB(int vb) : valorB(vb) {}
     int LeerValor() const { return valorB; }
   protected:
     int valorB;
};
```

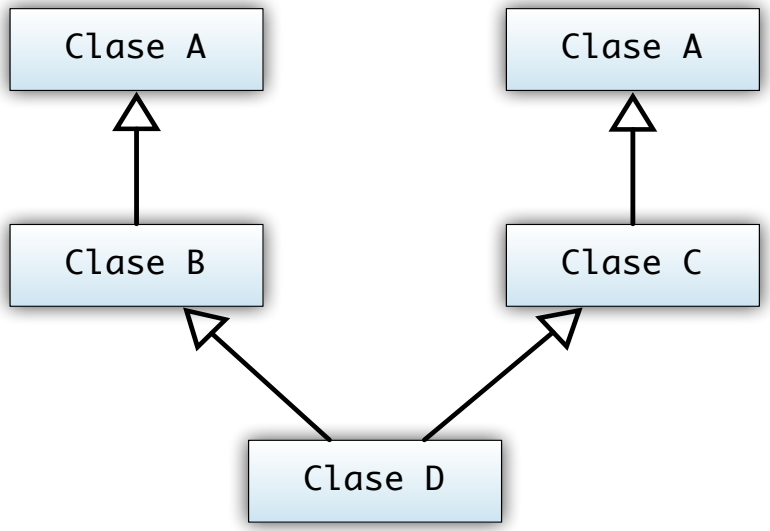
```
class ClaseC : public ClaseA, public ClaseB {
   public:
     ClaseC(int va, int vb) : ClaseA(va), ClaseB(vb) {};
     int LeerValorA() const { return ClaseA::LeerValor(); }
     int LeerValorB() const { return ClaseB::LeerValor(); }
};
```

```
int main() {
     ClaseC CC(12,14);
     cout      << CC.LeerValorA()
       << ",“
       << CC.LeerValorB()
       << endl;
     return 0;
}
```

```
WJRG-MacBookPro:Ejemplos wladimir$ g++ ClaseA5.cpp -o ClaseA5
WJRG-MacBookPro:Ejemplos wladimir$ ./ClaseA5
12,14
```

# Clases Base Virtuales

- Supongamos que tenemos una estructura de clases como ésta:




- La Clase D hereda dos veces los datos y funciones de la Clase A, con la consiguiente ambigüedad a la hora de acceder a datos y funciones heredadas de la Clase A

# Clases Base Virtuales

- Para solucionar esto se usan **clases base virtuales**
- Cuando derivemos una clase partiendo de una o varias clases base, podemos hacer que las clases base sean virtuales. Esto no afectará a la clase derivada.

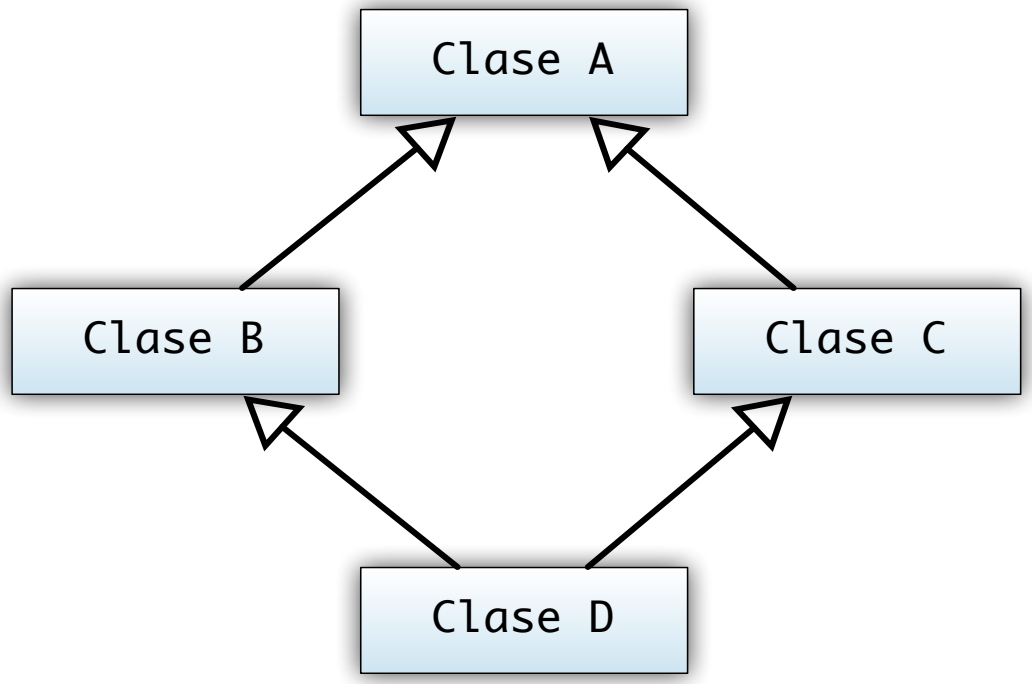
```
Class ClaseB : virtual public ClaseA {  };
```

- Desde el punto de vista de la ClaseB, no hay ninguna diferencia entre ésta declaración y la que hemos usado hasta ahora. La diferencia estará cuando declaramos la ClaseD.

```
class ClaseB : virtual  public  ClaseA {  };
class ClaseC : virtual  public  ClaseA {  };
class ClaseD : public ClaseB, public ClaseC {  };
```

# Clases Base Virtuales

- Ahora, la ClaseD sólo heredará una vez la ClaseA. La estructura quedará así:



# Clases Base Virtuales

- Cuando creemos una estructura de este tipo, deberemos tener cuidado con los constructores; el constructor de la ClaseA **deberá ser invocado** desde el de la ClaseD, ya que ni la ClaseB ni la ClaseC lo harán automáticamente.

- Veamos esto con el ejemplo de la clase "Persona". Derivaremos las clases "Empleado" y "Estudiante", y crearemos una nueva clase "Becario" derivada de estas dos últimas. Además haremos que la herencia de la clase "Persona" sea virtual, de modo que no se dupliquen sus funciones y datos.

# Ejemplo de clases base virtuales

```
#include <iostream>
#include <cstring>
using namespace std;

class Persona {
   public:
     Persona(const char *n) { strcpy(nombre, n); }
     const char *LeeNombre() const { return nombre; }
   protected:
     char nombre[30];
};

class Empleado : virtual public Persona {
   public:
     Empleado(const char *n, int s) : Persona(n), salario(s) {}
     int LeeSalario() const { return salario; }
     void ModificaSalario(int s) { salario = s; }
   protected:
     int salario;
};
```

# Ejemplo de clases base virtuales

```
class Estudiante : virtual public Persona {
   public:
     Estudiante(const char *n, float no) : Persona(n), nota(no) {}
     float LeeNota() const { return nota; }
     void ModificaNota(float no) { nota = no; }
   protected:
     float nota;
};
```

```
class Becario : public Empleado, public Estudiante {
   public:
     Becario(const char *n, int s, float no) :
        Empleado(n, s), Estudiante(n, no), Persona(n) {}
};
int main() {
   Becario Fulanito("Fulano", 1000, 7);
   cout << Fulanito.LeeNombre() << ","
        << Fulanito.LeeSalario() << ","
        << Fulanito.LeeNota() << endl;
   return 0;
}
```

```
WJRG-MacBookPro:Ejemplos wladimir$ g++ Persona.cpp -o Persona
WJRG-MacBookPro:Ejemplos wladimir$ ./Persona
Fulano,1000,7
WJRG-MacBookPro:Ejemplos wladimir$
```

uc3m | Universidad Carlos III de Madrid