Tema 3: Herencia en C++

Programación Orientada a Objetos

Curso 2008/2009

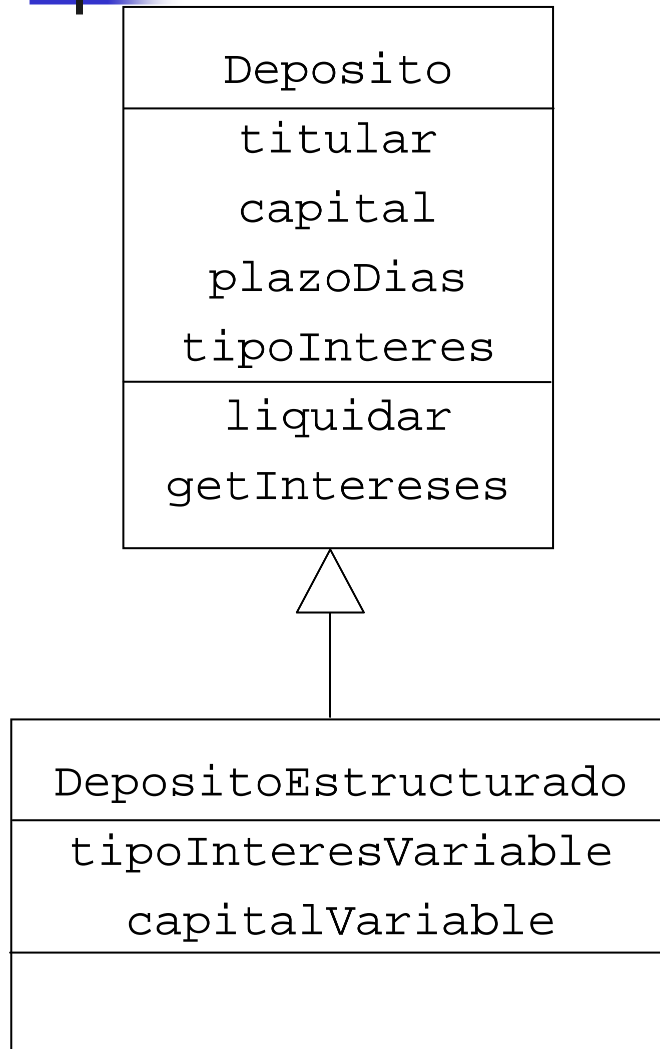
Begoña Moros Valle

DIS
Departamento de
Informática y Sistemas

Contenido

- Tipos de herencia
- Herencia y niveles de visibilidad
- Herencia y creación
- Redefinición de métodos
- Conversión de tipos
- Consulta del tipo dinámico
- Clases abstractas
- Punteros a función
- Iteradores
- Herencia múltiple

Caso de estudio



- Un depósito estructurado **es_un** tipo de depósito
- Un depósito estructurado tiene nuevos atributos
  - Tipo de interés variable
  - Capital variable
- Redefine parte de la funcionalidad heredada de depósito
  - El método que calcula los intereses
  - El método que devuelve el capital

Clase Deposito

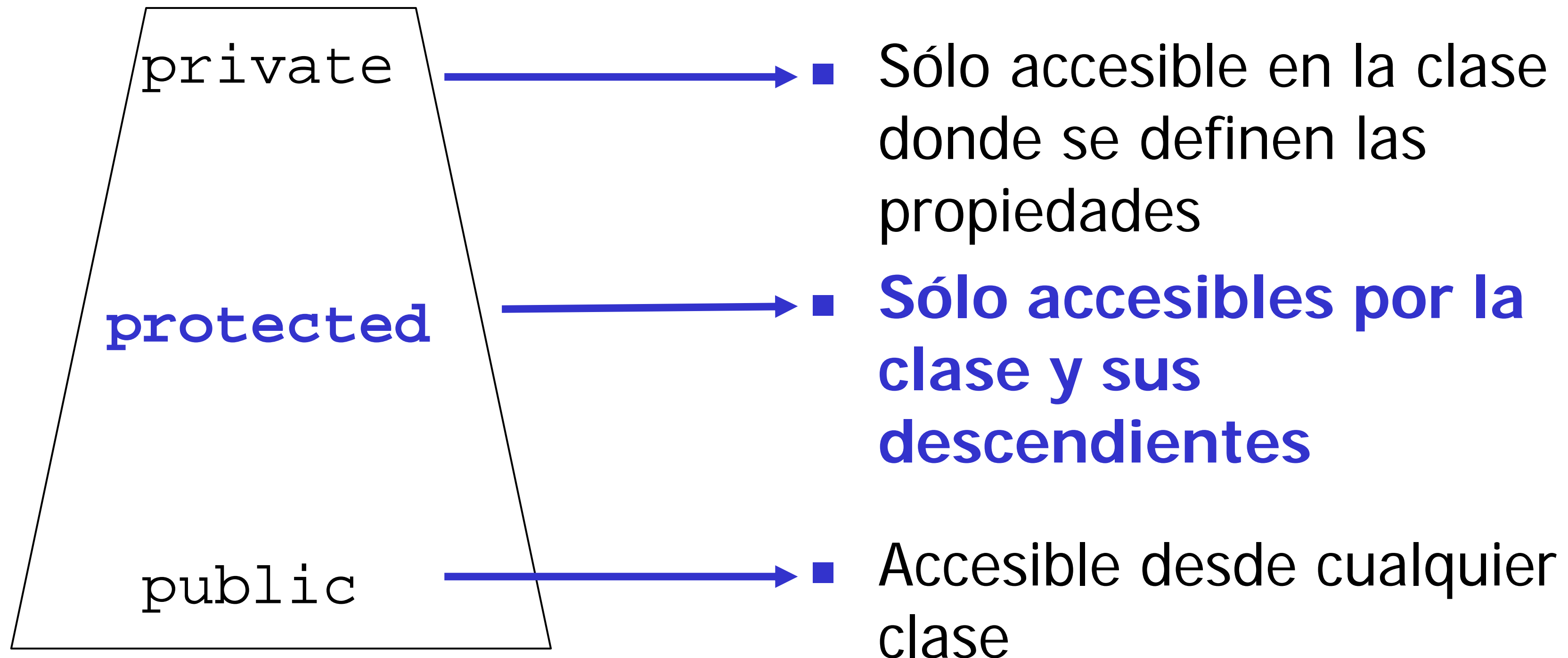
```
class Deposito {
private:
   Persona* titular;
   double capital;
   int plazoDias;
   double tipoInteres;
public:
   Deposito(…);
   double liquidar();
   double getIntereses();
   double getCapital();
   int getPlazoDias();
   double getTipoInteres();
   Persona* getTitular();
};
```

Clase Depósito Estructurado

```
class DepositoEstructurado: public Deposito{
private:
    double tipoInteresVariable;
    double capitalVariable;
public:
DepositoEstructurado(Persona titular, double capital, int
plazoDias,double tipoInteres, double tipoInteresVariable,
double capitalVariable);


    double getInteresesVariable();
    void setTipoInteresVariable(double interesVariable);
    double getTipoInteresVariable();
    double getCapitalVariable();
};
```

Niveles de visibilidad



- Sólo accesible en la clase donde se definen las propiedades
- **Sólo accesibles por la clase y sus descendientes**
- Accesible desde cualquier clase

Herencia pública

```
class B: public A {...}
```

- Por defecto, se mantiene el nivel de visibilidad de las propiedades heredadas (= Java)
- Se puede ampliar la visibilidad de las características heredadas
- Se puede reducir la visibilidad de las características heredadas

  →"agujero de tipos" debido a asignaciones polimórficas

Herencia privada

`class B: private A {...}`

- Todas las características de A se heredan como privadas
- Los tipos no son compatibles.
  - No se permiten hacer asignaciones polimórficas
- Es la opción por defecto
- Se puede mantener el nivel de visibilidad original calificando la rutina en el bloque `public` o `protected`
- Útil para la herencia de implementación
  - Heredar de una clase sólo para reutilizar la implementación

Constructor de Depósito Estructurado

- Los constructores no se heredan (= Java)
- El constructor de la clase hija (*clase derivada*) siempre tiene que invocar al constructor de la clase padre (*clase base*)

```
DepositoEstructurado::DepositoEstructurado(Persona titular,
double capital, int plazoDias, double tipoInteres, double
tipoInteresVariable, double capitalVariable):Deposito(titular,
capital, plazoDias, tipoInteres){

   this.tipoInteresVariable = tipoInteresVariable;

   this.capitalVariable = capitalVariable;

}
```

Redefinición de métodos

- La clase padre debe indicar que los métodos se pueden redefinir utilizando el modificador **`virtual`**
  - **¿Viola el Principio de Abierto-Cerrado?**
- Un método en la clase hija que tenga la misma signatura que un método virtual significa que lo está redefiniendo
  - **En la definición de la clase hija (fichero cabecera) hay que incluir los métodos que se redefinen**
- Para invocar la ejecución de la versión de uno de los métodos de cualquier otra clase se utiliza la calificación de rutinas
  - `NombreClase::nombreMétodo`
  - `Despotio::getCapital();`

Redefinición de métodos

```
class Deposito {
private:
   Persona* titular;
   double capital;
   int plazoDias;
   double tipoInteres;
public:
   Deposito(…);
   double liquidar();
   virtual double getIntereses();
   virtual double getCapital();
   int getPlazoDias();
   double getTipoInteres();
   Persona* getTitular();
};
```

Redefinición de métodos

- Métodos redefinidos en `DepositoEstructurado`

```
//Override
double DepositoEstructurado::getIntereses() {
  return Deposito::getIntereses() + getInteresesVariable();
}

//Override
double DepositoEstructurado::getCapital() {
  return Deposito::getCapital() + getCapitalVariable();
}
```

- Invocan a las versiones definidas en la clase `Deposito`

Polimorfismo y Ligadura dinámica

- El *polimorfismo de asignación* está permitido para entidades con semántica por valor y referencia.
- Sólo se consideran que dos métodos están sobrecargados (*polimorfismo ad-hoc*) si se definen dentro del mismo ámbito
  - Una función de la clase hija con el mismo nombre que una función heredada con distinta signatura la oculta.
- **Ligadura dinámica**:
  - Sólo es posible para **métodos virtuales**.
  - La entidad polimórfica debe ser de **tipo referencia**.
- **Ligadura estática**:
  - Se aplica la versión del método asociada al tipo estático de la variable.

Asignaciones polimórficas

```
Deposito deposito(…);
DepositoEstructurado de(…);

//Asignación polimórfica entre objetos valor
deposito = de;
//Ligadura estática, Deposito::getCapital
cout<<"Capital total "<<deposito.getCapital()<<endl;

Deposito* ptrDeposito = new Deposito(…);
DepositoEstructurado* ptrDe = new DepositoEstructurado(…);

//Asignación polimórfica de punteros
ptrDeposito = ptrDe;
//Ligadura dinámica, DepositoEstructurado::getCapital
cout<<"Capital total "<<ptrDeposito->getCapital()<<endl;
 ptrDesposito->liquidar(); //Ligadura estática
```

Sobrecarga en C++

```
class Deposito {
…
public:
    virtual double getCapital();
};
class DepositoEstructurado: public Deposito{
…
public:
    double getCapital(bool tipo);
};
```

- `getCapital` está definido en distinto ámbito
- `getCapital` **no está sobrecargado** en la clase `DepositoEstructurado`

Sobrecarga en C++

```
class Deposito {
…
public:
    virtual double getCapital();
};
class DepositoEstructurado: public Deposito{
…
public:
   double getCapital();
   double getCapital(bool tipo);
};
```

- `getCapital` **está sobrecargado**
  - La versión redefinida devuelve el capital total
  - La versión sobrecargada devuelve el capital fijo o variable en función del parámetro

Conversión de tipos

- Operador **`dynamic_cast<Tipo*>(ptro)`**
  - Convierte el `ptro` en el puntero a `Tipo`
  - `ptro` debe ser una entidad polimórfica (su clase debe tener algún método virtual)
  - La conversión se hace entre tipos compatibles
  - Si la conversión falla se le asigna cero (puntero NULL)
- También **`dynamic_cast<Tipo>(ref)`**
  - En caso de que la conversión no sea posible se lanza una excepción (`bad_cast`)

Conversión de tipos

- Establecemos el tipo de interés variable a los depósitos estructurados

```
Deposito** productos;
depositos = new Deposito*[MAX_DEPOSITOS];
…
DepositoEstructurado* depEst;

for (int i =0; i<MAX_DEPOSITOS; i++){
  depEst = dynamic_cast<DepositoEstructurado*>(depositos[i]);
  if (depEst != NULL)
       depEst->setTipoInteresVariable(0.05);
}
```

Consulta del tipo dinámico

- Contamos el número de depósitos abiertos

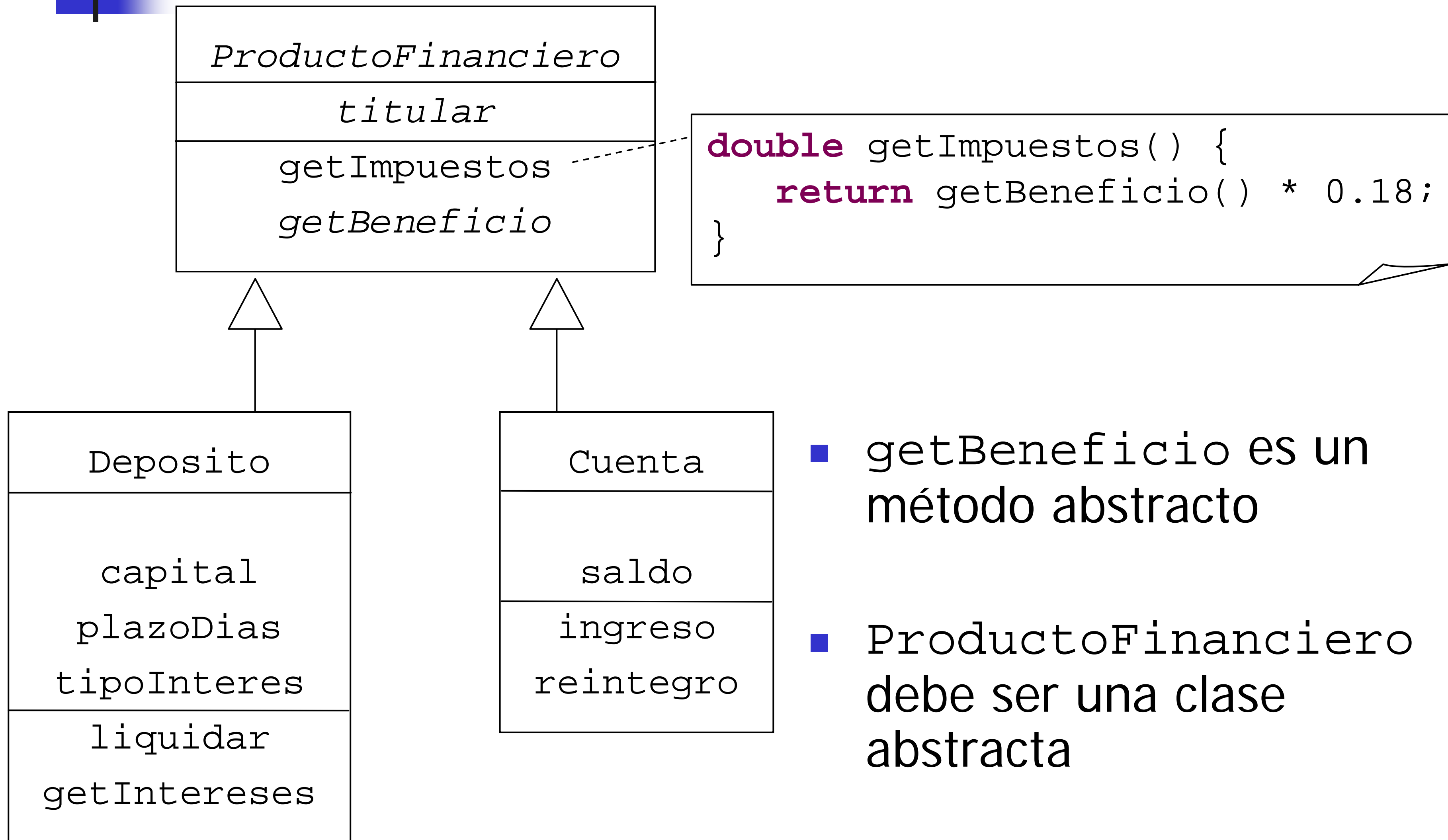
```
int numDepositos = 0;


for (int i =0; i<MAX_PRODUCTOS; i++){
  if (dynamic_cast<Deposito*>(productos[i]))
          ++numDepositos;


}
```

- Equivalente a `instanceof` de Java

Clases abstractas



- `getBeneficio` es un método abstracto
- `ProductoFinanciero` debe ser una clase abstracta

Clases abstractas

- No existe una palabra reservada para indicar que una clase es abstracta
- Una clase es abstracta si contiene un **método virtual puro**

```
class ProductoFinanciero{
 private:
  Persona* titular;

 public:
  ProductoFinanciero(Persona* titular);
  virtual double getBeneficio()=0;
  double getImpuestos();
  Persona* getTitular();
};
```

Interfaces

- C++ **no define el concepto de interfaz** de Java.
- No es necesario, ya el lenguaje ofrece herencia múltiple.
  - Si una clase quiere ser compatible con varios tipos, basta con que herede públicamente de otras clases.
- El equivalente a las interfaces de Java sería una **clase totalmente abstracta sólo con métodos virtuales puros**.

Acciones

- Para poder pasar una acción como parámetro de una función podemos utilizar dos estrategias:
  - **Punteros a función:**
    - En C++ es posible pasar una función como parámetro
  - **Clase que represente la acción**:
    - Definir una clase totalmente abstracta que simule la interfaz de Java
    - Definir una subclase por cada acción que se necesite implementar

Acciones mediante punteros a función

- Un **puntero a función** es una variable que guarda la dirección de comienzo de la función
- Puede considerarse como una especie de "alias" de la función que hace que pueda pasarse como parámetro a otras funciones
  - Las reglas del paso de parámetros se aplican también para el paso de funciones como parámetro
- `X (*fptr) (A);`
  - `fptr` es un puntero a función que recibe `A` como argumento y devuelve `X`

Punteros a función

```
namespace banco{
  class Sucursal{
    private:
      ProductoFinanciero** productos;
    public:
      Sucursal();
      ProductoFinanciero* buscar(
                          bool (*condicion) (ProductoFinanciero*));
  };
  //Condiciones de búsqueda
  bool depositoAlto (ProductoFinanciero* producto);
}
```

- El parámetro del método `buscar` es una función que recibe como parámetro un puntero a un `ProductoFinanciero` y devuelve un valor booleano.
  - Por ejemplo, la función `depositoAlto`

Método genérico de búsqueda

```
ProductoFinanciero* Sucursal::buscar(
                    bool (*condicion)(ProductoFinanciero*)){
   bool encontrado = false;
   for (int i =0; i<MAX_PRODUCTOS; i++)
      if (condicion(productos[i])){
            encontrado = true;
            return productos[i];
    }
    if (!encontrado) return NULL;
}
```

Condición de búsqueda

- La función depositoAlto **NO puede ser un método de instancia.** La definimos dentro del espacio de nombres.

```
bool banco::depositoAlto(ProductoFinanciero* producto){
  Deposito* deposito = dynamic_cast<Deposito*>(producto);
  if (deposito!= NULL)
       return (deposito->getCapital()>1000);
  else return false;
}
```

```
Sucursal cam;
…
ProductoFinanciero* producto = cam.buscar(depositoAlto);
```

Clase que representa la acción

- “Interfaz” `Condicion` → Clase totalmente abstracta

```
template <class T> class Condicion{
 public:
   virtual bool test(T elemento) = 0;
};
```

- Habría que definir una subclase por cada criterio de búsqueda

- Por ejemplo, `CondicionCapital`, buscamos, de entre todos los productos financieros del banco aquellos depósitos con un capital superior a un determinado valor umbral.

Método genérico de búsqueda

```
ProductoFinanciero* Banco::buscar
                (Condicion<ProductoFinanciero*>* condicion){
    bool encontrado = false;
    for (int i =0; i<MAX_PRODUCTOS; i++)
    if (condicion->test(productos[i])){
        encontrado = true;
        return productos[i];
    }
    if (!encontrado) return NULL;
}
```

Implementación de una condición

```
class CondicionCapital: public Condicion<ProductoFinanciero*>{
 private:
  double capitalUmbral;
 public:
  CondicionCapital(double capital);
  bool test(ProductoFinanciero* elemento);
};
```

```
bool CondicionCapital::test(ProductoFinanciero* elemento){
Deposito* deposito = dynamic_cast<Deposito*>(elemento);
if (deposito!= NULL)
return (deposito->getCapital()>capitalUmbral);
else return false;
}
```

Clase que representa la acción

- Para invocar al método de búsqueda hay que crear un objeto del tipo de condición que se vaya a utilizar

```
Sucursal sucursal;
…
ProductoFinanciero* producto;
CondicionCapital* cc = new CondicionCapital(1000);
producto = sucursal.buscar(cc);
…
```

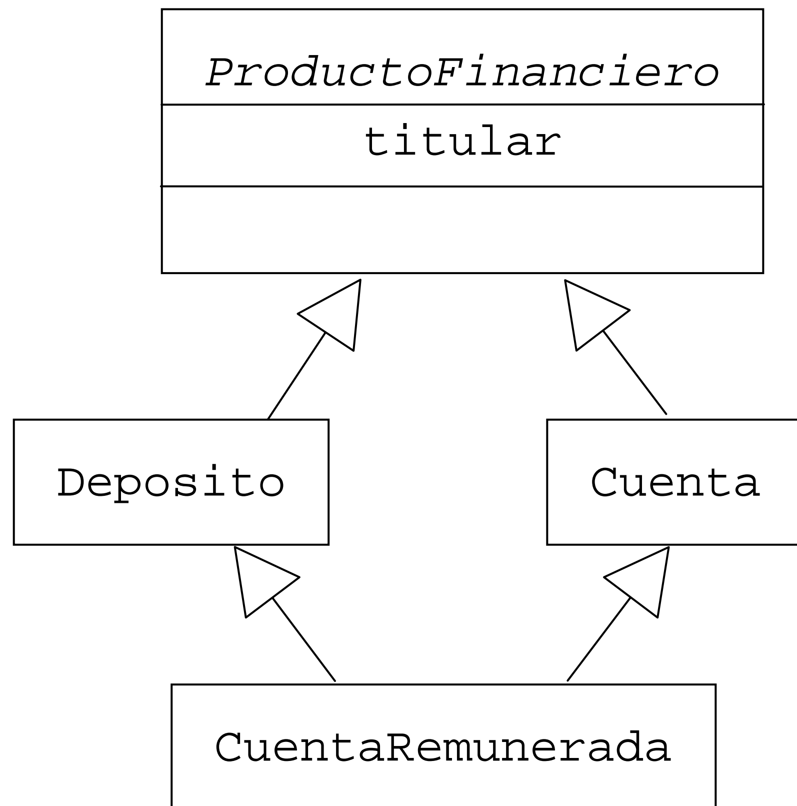
Herencia múltiple

- En C++ es posible que una clase tenga más de una clase padre

- Problemas:
  - **Colisión de nombres:** la clase hija hereda dos métodos efectivos con el mismo nombre y diferentes implementaciones
    - Si se redefine el método en la clase hija se "funden" las dos versiones en una nueva
    - Si se necesitan las dos funciones se deben calificar las rutinas para resolver la ambigüedad.
  - **Herencia repetida:** una clase se hereda dos veces

Herencia repetida

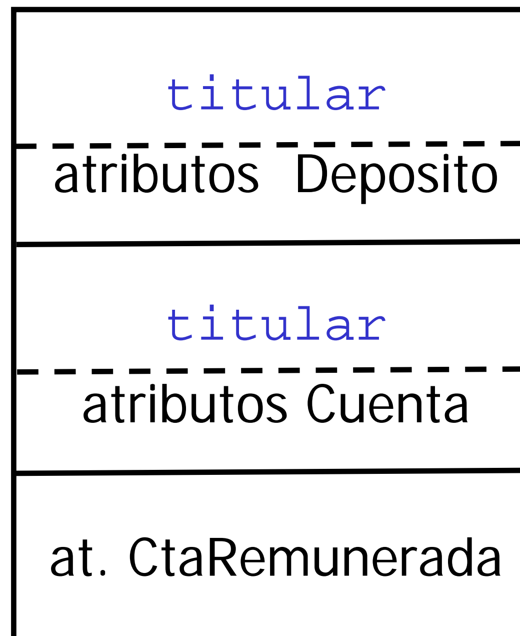
```
class CuentaRemunerada: public Cuenta, public Deposito{
        …
};
```



- `CuentaRemunerada` hereda dos veces de `ProductoFinanciero`
- ¿Existe dos campos `titular` en `CuentaRemunerada`?
- Conflicto de nombres con el método `getBeneficio`

Herencia repetida

- Por defecto en C++ **se duplican todos los atributos heredados**



Estructura de un objeto `CuentaRemunerada`

Herencia repetida

- El método `getTitular` se hereda dos veces → colisión de nombres
- La llamada `getTitular` (sin calificar) sobre una cuenta remunerada es ambigua.
- Hay que resolver la ambigüedad mediante la calificación de rutinas y atributos

```
CuentaRemunerada* cr = new CuentaRemunerada(…);
cout<<"Titular "<<cr->Cuenta::getTitular()->getNombre();
```

Asignaciones polimórficas

- Hay dos objetos `ProductoFinanciero` en un objeto `CuentaRemunerada`
- La asignación entre ambas clases es ambigua

```
ProductoFinanciero* pf;
CuentaRemunerada* cr = new …;
pf = cr; //Error en tiempo de compilación
```

- Solución: establecer el "camino"

```
ProductoFinanciero* pf;
CuentaRemunerada* cr = new CuentaRemunerada(…);

pf = (Cuenta*)cr;
```

Asignaciones polimórficas ambiguas

- La aplicación del método `getBeneficio` sobre un objeto CuentaRemunerada es ambigua
  - Si no se hace la llamada el compilador no avisa del conflicto de nombres

```
ProductoFinanciero* pf;
CuentaRemunerada* cr = new CuentaRemunerada(…);

pf = (Cuenta*)cr;

cout<<"Cuenta remunerada "<<pf->getTitular()->getNombre()<<endl;
cout<<"beneficio "<<pf->getBeneficio()<<endl;
cout<<"Beneficio de cr "<<cr->getBeneficio()<<endl; //Error
```

Herencia virtual

- Si queremos que la clase `CuentaRemunerada` herede una única copia de `ProductoFinanciero`, las clases intermedias tienen que declarar su herencia como `virtual`.
- Se resuelve la ambigüedad de las asignaciones polimórficas
- Sólo debe existir una versión de los métodos heredados
  - El compilador detecta que se están heredando dos versiones del método `getBeneficio`

Herencia virtual

```
class ProductoFinanciero{
…
};

class Deposito: virtual public ProductoFinanciero {
…
};

class Cuenta:  virtual public ProductoFinanciero {
…
};

class CuentaRemunerada: public Cuenta, public Deposito{
…
};
```

Constructores y herencia virtual

- El constructor de la clase `CuentaRemunerada` tiene que llamar al constructor de `ProductoFinanciero` aunque no sea una clase de la que hereda directamente.

```
 CuentaRemunerada::CuentaRemunerada(Persona* p, double s, int
plazoDias,double tipoInteres)
:ProductoFinanciero(p),
 Cuenta(p,s),
 Deposito(p, s, plazoDias, tipoInteres){
   …
}
```

Herencia repetida virtual

- El método `getBeneficio()` es definido por `Cuenta` y `Deposito`: colisión de nombres.
  - Error en tiempo de compilación no existe una única versión

```
class CuentaRemunerada: public Cuenta, public Deposito{
public:
  CuentaRemunerada(…);
  double getBeneficio();
};
```

- Se evita al redefinir el método eligiendo una de las versiones:

```
double CuentaRemunerada::getBeneficio(){
     return Deposito::getBeneficio();
}
```

Asignaciones polimórficas

```
ProductoFinanciero* pf;
CuentaRemunerada* cr = new CuentaRemunerada(…);

pf = cr;

cout<<"Cuenta remunerada "<<pf->getTitular()->getNombre()<<endl;
cout<<"beneficio "<<pf->getBeneficio()<<endl;
```

- No existe ambigüedad en la asignación
- Se ejecuta el método `getBeneficio` disponible en `CuentaRemunerada`

Función dominante

- Si un método de la clase `ProductoFinanciero` se redefine sólo en una de las clases hijas, no existe ambigüedad
- Se dice que la versión redefinida domina sobre la versión original
- En el caso de una asignación polimórfica se ejecutará la versión dominante.

Herencia de C++ vs. Java

- La herencia en C++ es **diferente a Java** en varios aspectos:
  - **Herencia múltiple**: una clase puede heredar de varias clases.
  - **Herencia privada**: heredar de una clase sólo el código, pero no el tipo.
  - **Redefinición de métodos**: por defecto, los métodos de una clase no pueden ser redefinidos.
  - **No existe el tipo `Object`** raíz de la jerarquía de clases.