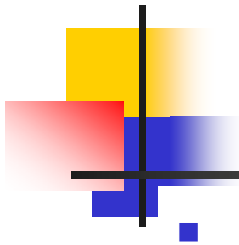
Tema 5: Programación Orientada a Objetos en C++

Programación Orientada a Objetos

Curso 2009/2010

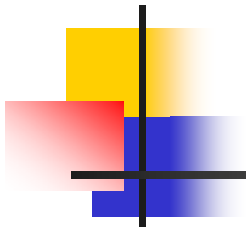
Begoña Moros Valle

DIS
Departamento de
Informática y Sistemas



Contenido

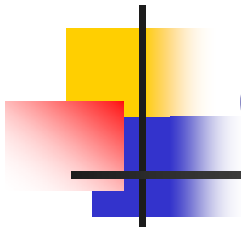
- Introducción
- Clases y Objetos en C++:
  - Módulos: Clases, estructuras y espacios de nombres
  - Semántica referencia (punteros) y operadores
  - Métodos y mensajes
  - Creación y destrucción de objetos
  - Genericidad (`template`)
- Herencia en C++:
  - Tipos de herencia
  - Herencia y niveles de visibilidad
  - Herencia y creación
  - Redefinición de métodos
  - Conversión de tipos y consulta del tipo dinámico
  - Clases abstractas
  - Punteros a función
  - Herencia múltiple
- Corrección y Robustez en C++: asertos y excepciones



Introducción

- Creado por **Bjarne Stroustrup** en los 80.
- Diseñado como una **extensión de C** que incluye **características orientadas a objetos**.

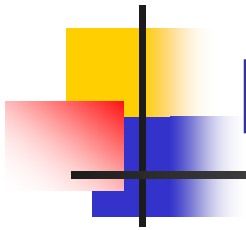
  ➔ Es un **lenguaje híbrido**.
- Ha sido inspiración de lenguajes de programación posteriores como Java y C#.
- A finales de los 90 fue estandarizado: **ISO C++**
- Las librerías del lenguaje son escasas. La librería más notable es **STL** (*Standard Template Library*).
- Actualmente, sigue siendo un lenguaje de programación importante en algunos dominios.



Clases y Objetos en C++

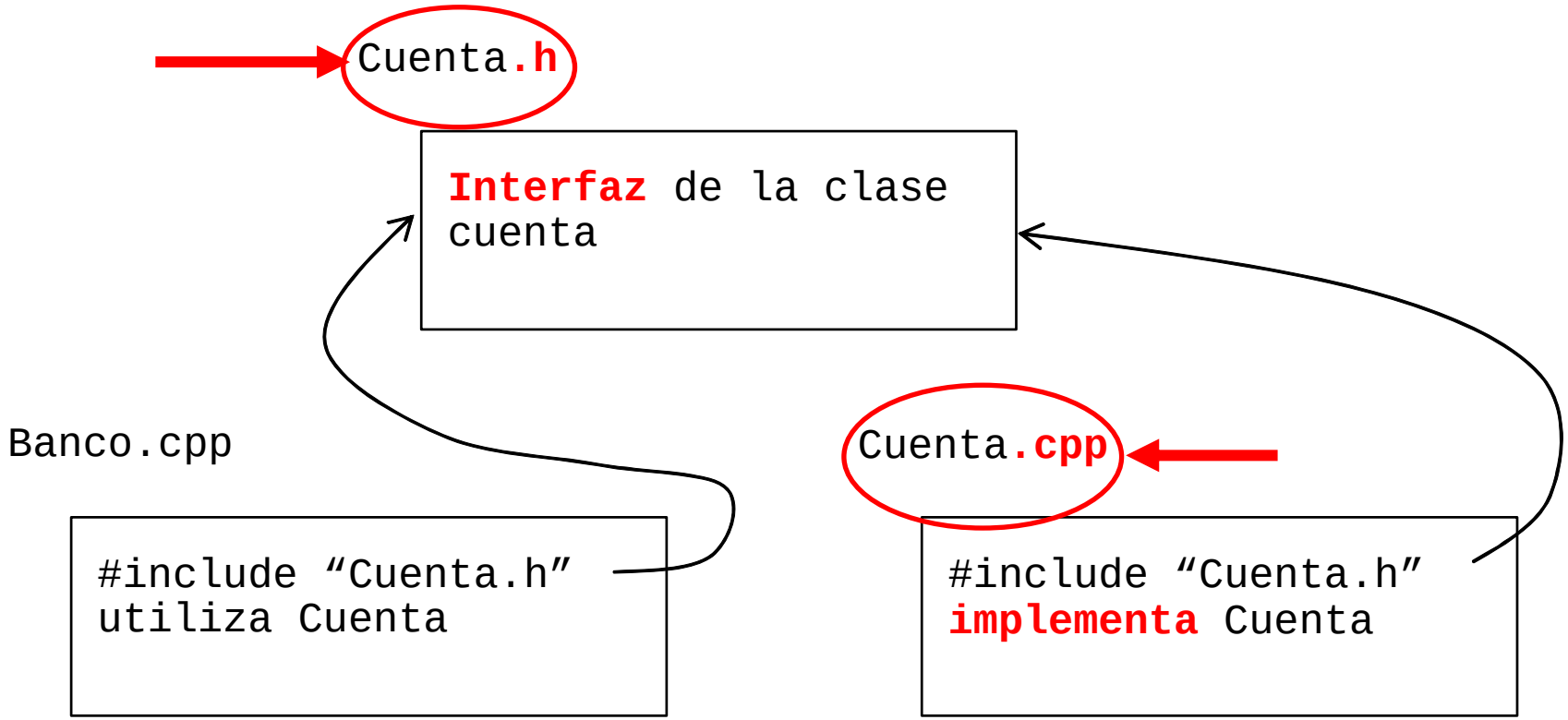
- Módulos en C++:
  - Clases
  - Estructuras (`struct`)
  - Espacios de nombres (`namespace`)
- Semántica referencia
  - Semántica de los operadores “==“ e “=“
- Métodos y mensajes
- Creación y destrucción de objetos
- Genericidad → `template`

Especificación de la clase Cuenta



- A diferencia de Java una clase se implementa en ficheros separados para:
  - Definición de la interfaz de la clase (fichero cabecera) → `Cuenta.h`
  - La implementación → `Cuenta.cpp`
- Hay que incluir el fichero de cabecera en el fichero de implementación y en los ficheros de las clases cliente → **`#include "Cuenta.h"`**
- No tiene por qué existir correspondencia entre la estructura física de un programa (organización de ficheros fuente) y la estructura lógica (organización de las clases).

Especificación de la clase Cuenta



Cuenta.h (1/2)

```
#include "Persona.h“


class Cuenta {
public:
  void reintegro(double cantidad);
  void ingreso(double cantidad);
  double getSaldo() const;
  Persona* getTitular() const;
  double* getUltimasOperaciones(int n) const;
  static int getNumeroCtas();
  …
```

Cuenta.h (2/2)

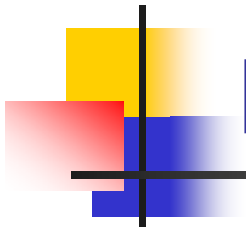
```
private:
  const static int MAX_OPERACIONES = 20;
  const static double SALDO_MINIMO = 100;

  Persona* titular;
  double saldo;
  int codigo;
  static int ultimoCodigo;
  double* ultimasOperaciones;
  bool puedoSacar(double cantidad);
};
```

Cuenta.cpp

```
int Cuenta::ultimoCodigo = 0;
void Cuenta::reintegro(double cantidad){
        if (puedoSacar(cantidad))
                saldo = saldo - cantidad;
        }
}
void Cuenta::ingreso(double cantidad){
        saldo = saldo + cantidad;
}
bool Cuenta::puedoSacar(double cantidad){
        return (saldo >= cantidad);
}
…
```

Clases en C++

- Se pueden definir tanto atributos y métodos de clase (**`static`**) como de instancia (= Java).
- Palabra reservada **`const`**
  - Indica que un **atributo** es **inmutable**
    - Equivalente a atributos `final` en Java
    - **`const`** `Persona* titular;` → puntero inmutable
    - `Persona*` **`const`** `titular;` → objeto persona inmutable
  - Indica que la ejecución de una **función no va a cambiar el estado del objeto** receptor de la llamada
- En el fichero de implementación el nombre de los métodos está calificado con la clase.
  - `NombreClase::nombreMetodo`



Niveles de visibilidad en C++

- Especificación de acceso **para un grupo de miembros**:
  - **`public`**: un cliente puede consultarlo y modificarlo
  - **`private`**: sólo accesible dentro de la clase
    - Opción por defecto
    - Se puede acceder a los campos privados de los objetos de la misma clase como en Java

- **Clases amigas**: Se le concede acceso TOTAL a la clase amiga
  - La amistad no es hereditaria ni transitiva

Ejemplo: friend class

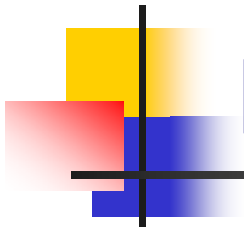
```
class NodoArbol {
        friend class Arbol;
        private:
          int valor;
          NodoArbol decha;
          NodoArbol izda;
                …
};
```

```
class Arbol{
        private:
                NodoArbol *raiz;
                ...
        … raiz->valor = 50; ...
};
```

Concepto de estructura

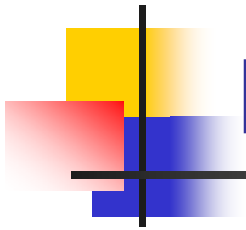
- Unidad modular heredada de C
  - en C++ se amplía con la definición de funciones
- Totalmente equivalente al concepto de clase salvo:
  - Cambia la palabra `class` por **`struct`**
  - Por defecto todos los elementos son públicos salvo que se diga lo contrario.

```
struct Cuenta{
  void ingreso (double cantidad);
  void reintrego (double cantidad);
 private:
  int codigo;
  Persona* titular;
  double saldo;
};
```



Espacio de nombres

- El espacio de nombres (**`namespace`**) es un mecanismo para agrupar un conjunto de elementos (clases, enumerados, funciones, etc.) relacionados
- Es importante el orden de definición de los elementos en el espacio de nombres
- Puede estar definido en ficheros diferentes
- Es un concepto **diferente a los paquetes de Java**:
  - No hay relación entre la estructura lógica y física.
  - No proporcionan privilegio de visibilidad.



Espacio de nombres

- Para utilizar un elemento definido en un espacio de nombres:
  - Se utiliza el **nombre calificado** del elemento:
    - `gestionCuentas::Cuenta`
  - Se declara el **uso del espacio** de nombres:
    - `using namespace gestionCuentas;`

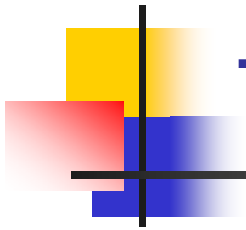
Espacio de nombres

- En `Cuenta.h`

```
namespace gestionCuentas{
  class Cuenta {
        …
  };
}
```

- En `Banco.h`

```
namespace gestionCuentas{
  class Banco {
        …
  };
}
```

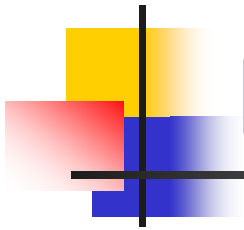


Tipos del lenguaje

- Tipos de datos primitivos:
  - `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `bool`, `etc.`
- Enumerados:
  - `enum {OPERATIVA, INMOVILIZADA, NUM_ROJOS};`
- Objetos embebidos:
  - Subobjetos
- Punteros:
  - `T*` es el tipo “puntero a T”
  - Una variable de tipo `T*` puede contener la dirección de un objeto de tipo `T`

Arrays en C++

- `Cuenta cuentas[10];`
  - `cuentas` es un array de tamaño 10
  - No se asigna un valor inicial para cada posición
- `Cuenta* cuentas = new Cuenta[10];`
  - `cuentas` es un ptro a un array de cuentas
- `Cuenta** cuentas = new Cuenta*[10];`
  - cuentas es ptro a un array de punteros a Cuenta (= Java)
- Las dos primeras declaraciones sólo funcionarían si la clase Cuenta tuviera definido un constructor por defecto
- No existe una función equivalente a `length` de Java
- No se controla el acceso a posiciones fuera de los límites del array.



Enumerados

- Es un tipo que puede almacenar un conjunto de valores
- El enumerado define un conjunto de constantes de tipo entero
- Por defecto los valores se asignan de forma creciente desde `0`.
- El tipo de cada uno de los elementos es el del enumerado.
- Un enumerado es un tipo, por lo que el usuario puede definir sus propias operaciones.

Enumerados. Definición

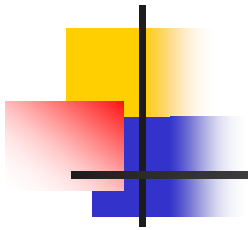
```
namespace banco{
  enum EstadoCuenta{
       OPERATIVA, INMOVILIZADA, NUM_ROJOS
     //OPERATIVA == 0, INMOVILIZADA == 1, NUM_ROJOS == 2
  };

  class Cuenta {
       …
    private:
       …
       EstadoCuenta estado;
   };
}
```

Enumerados. Uso

- En `Cuenta.cpp`

```
#include "Cuenta.h"

using namespace banco;

…

void Cuenta::reintegro(double cantidad){
    if (estado!=INMOVILIZADA && puedoSacar(cantidad)){
        saldo = saldo - cantidad;
    }
}
```

- Para referenciar un valor del enumerado no tiene que ir precedido por el nombre del enumerado



Inicialización de los atributos

- No se puede asignar un valor inicial a los atributos en el momento de la declaración a menos que sea una constante (`const static`)
  - Se considera definición y no declaración
- A diferencia de Java, **no podemos asegurar que los atributos tengan un valor inicial**
- Solución → Definición de constructores

Constructores

- Método especial con el mismo nombre que la clase y sin valor de retorno (= Java)
- Se permite sobrecarga
- Si no existe ningún constructor en la clase el compilador proporciona el constructor por defecto

```
class Cuenta {
public:
Cuenta (Persona *titular);
Cuenta (Persona *titular, double saldoInicial);
…
};
```

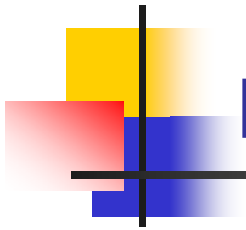
Constructores de la clase Cuenta

```
Cuenta::Cuenta (Persona *persona){
 codigo = ++ultimoCodigo;
 titular = persona;
 saldo = SALDO_MINIMO;
 ultimasOperaciones = new double[Cuenta::MAX_OPERACIONES];
 numOperaciones = 0;
 estado = OPERATIVA;
}
Cuenta::Cuenta (Persona *persona, double saldoInicial){
 codigo = ++ultimoCodigo;
 titular = persona;
 saldo = saldoInicial;
 ultimasOperaciones = new double[Cuenta::MAX_OPERACIONES];
 numOperaciones = 0;
 estado = OPERATIVA;
}
```

Sobrecarga de constructores

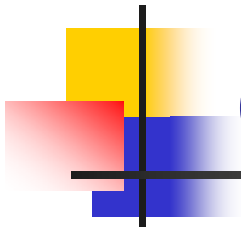
- A diferencia de Java, **`this` no se puede utilizar como una función** para reutilizar el código de los constructores

- **Soluciones**:
  - Utilizar un método privado al que se invoca desde los constructores
  - Utilizar argumentos por defecto para los constructores



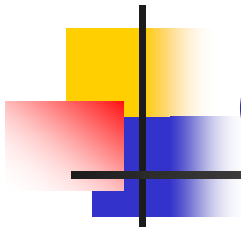
Reutilización de código en constructores

```
Cuenta::Cuenta (Persona *persona){
  inicializa(persona,SALDO_MINIMO);
}
Cuenta::Cuenta (Persona *persona, double saldoInicial){
  inicializa(persona, saldoInicial);
}
void Cuenta::inicializa(Persona *persona, double saldoInicial){
  codigo = ++ultimoCodigo;
  titular = persona;
  saldo = saldoInicial;
  ultimasOperaciones = new double[Cuenta::MAX_OPERACIONES];
  estado = OPERATIVA;
}
```



Constructores con argumentos por defecto

- Un **argumento por defecto** es una valor que se da en la declaración para que el compilador lo inserte automáticamente en el caso de que no se proporcione ninguno en la llamada a la función
- Es una opción para evitar el uso de métodos sobrecargados
- **Reglas para argumentos por defecto**:
  - Sólo los últimos pueden ser por defecto, es decir, no puede poner un argumento por defecto seguido de otro que no lo es.
  - Una vez se empieza a utilizar los argumentos por defecto al realizar una llamada a una función, el resto de argumentos también serán por defecto (esto sigue a la primera regla).
- Los argumentos por defecto **sólo se colocan en la declaración de la función** en el fichero de cabecera
  - El compilador debe conocer el valor por defecto antes de utilizarlo.
- Puede utilizarse para la definición de cualquier función de la clase



Constructor con argumentos por defecto

- Para la clase `Cuenta` definimos **un único constructor** con dos argumentos:
  - `titular`: obligatorio pasarlo como parámetro
  - `saldo`: que tiene como valor por defecto el saldo mínimo.

- No necesitamos dos constructores sobrecargados en la clase `Cuenta`.

Constructor para la clase Cuenta

- Fichero cabecera:

```
class Cuenta {
public:
Cuenta (Persona *titular, double saldoInicial= SALDO_MINIMO);
…
};
```

- Las siguientes creaciones **son equivalentes**:
  - `Cuenta(titular);`
    - Toma como valor del saldo el valor por defecto (100)
  - `Cuenta (titular, 300);`
    - El parámetro establece el valor del saldo a 300

Creación de objetos

- Reserva dinámica de memoria con el operador **new**
  - Si es un puntero a un objeto no se reserva memoria en el momento de la declaración
  - `Cuenta* cta = new Cuenta (unaPersona);`
- **Creación de objetos**:
  - Cuando se declara el objeto se reserva la memoria
  - `Cuenta cta(unaPersona);`
  - `Cuenta cta;`
    - Sería correcto si existe el constructor por defecto o un constructor que tiene definidos todos los parámetros por defecto.
    - Si no existe da un error en tiempo de compilación.

Destrucción de objetos

- No existe un mecanismo automático como en Java para liberar la memoria dinámica
  - Solución → **definición de destructores**
- Los destructores son métodos especiales con el mismo nombre de la clase, precedidos por `~`, y sin argumentos.
- Se invoca automáticamente cada vez que se libera la memoria del objeto (`delete`).
- Ejecuta “trabajos de terminación”
  - El uso habitual es liberar la memoria adquirida en el constructor

Destructor para la clase Cuenta

- Declaración del destructor en el fichero cabecera (`Cuenta.h`):

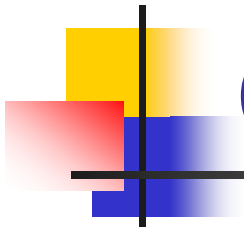
```
class Cuenta {
public:
Cuenta (Persona *titular, double saldoInicial= SALDO_MINIMO);
~Cuenta();
…
};
```

- Implementación (`Cuenta.cpp`):

```
Cuenta::~Cuenta(){
    delete[] ultimasOperaciones;
}
```

Semántica referencia vs. Semántica valor

- **Semántica referencia** asociada al tipo "puntero"
  - `Persona *titular;`
  - Valor `0` equivalente a `null` en Java → está predefinida la constante `NULL`
    - `const int NULL = 0;`
- Si no se define un puntero estamos definiendo un objeto (**semántica valor**)
  - Permite definir **objetos embebidos**
  - Beneficios:
    - Representar relaciones de composición → Un objeto forma parte de manera única de otro



Creación de objetos compuestos

- Un objeto compuesto contiene un objeto embebido
- En el momento de la creación de un objeto compuesto se reserva memoria para los objetos embebidos.
- Para inicializar el objeto embebido:
  - O bien existe el constructor por defecto en la clase del objeto embebido
  - O bien se inicializa en línea en el constructor del objeto compuesto

Objeto embebido en Cuenta

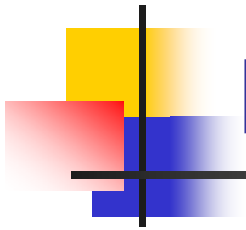
```
class Cuenta {
public:
…
private:
 Persona autorizado;
 Persona* titular;
 double saldo;
 int codigo;
 EstadoCuenta estado;
…
};
```

- Si no existe el constructor por defecto en la clase `Persona` este código no compila

Solución

- Definir un nuevo constructor en `Cuenta` pasando como parámetro los valores iniciales del subobjeto

```
Cuenta::Cuenta (Persona* persona, string nombreAut, string
dniAut, double saldoInicial):autorizado(nombreAut, dniAut){
 codigo = ++ultimoCodigo;
 titular = persona;
 saldo = saldoInicial;
 ultimasOperaciones = new double[Cuenta::MAX_OPERACIONES];
 numOperaciones = 0;
 estado = OPERATIVA;
}
```



Métodos y mensajes

- Los métodos definidos en una clase son los mensajes aplicable sobre los objetos de la clase.
- Un mensaje se aplica siempre sobre un objeto → la instancia actual
  - `this` es un puntero a la instancia actual
- Distinta **sintaxis para los mensajes**:
  - '`->`' (*notación flecha*) si el objeto receptor es un puntero
  - '`.`' (*notación punto*) el receptor es un objeto

Sintaxis de los Mensajes

```
Cuenta  objCta;
Cuenta* ptroCta;
```

- **Notación punto** para objetos

```
objCta.reintegro(1000);
(*ptrCta).reintegro(1000);
```

- **Notación "flecha"** para punteros:

```
ptroCta->reintegro(1000);
(&objCta)->reintegro(1000);
```

Paso de parámetros

- Soporta paso de parámetros por valor y por referencia

```
void f (int val, int& ref){
  val++;
  ref++;
}
```

  - `val` se pasa **por valor** → `val` incrementa una copia del parámetro real (= Java)
  - `ref` se pasa **por referencia** → `ref` incrementa el parámetro real

Paso de objetos como parámetro

```
void Banco::transferencia(Cuenta* emisor, Cuenta* receptor,
double cantidad){
        emisor->reintegro(cantidad);
        receptor->ingreso(cantidad);
        emisor=new Cuenta(emisor->getTitular());
}
```

- **Paso por valor del puntero (= Java)**
- El estado de `emisor` y `receptor` cambia
- El nuevo puntero de `emisor` no afecta al emisor real que se pasa como parámetro (es una copia)

Paso de objetos como parámetro

```
void Banco::transferencia(Cuenta emisor, Cuenta receptor,
double cantidad){

      emisor.reintegro(cantidad);

      receptor.ingreso(cantidad);

}
```

- **Paso por valor de los objetos**
- El estado de `emisor` y `receptor` NO CAMBIA
- `emisor` y `receptor` son una copia de los objetos que se pasan como parámetro.

Paso de objetos como parámetro

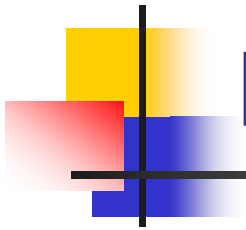
```
void Banco::transferencia(Cuenta* emisor, Cuenta*& receptor,
double cantidad){
  emisor->reintegro(cantidad);
  receptor->ingreso(cantidad);
  receptor = new Cuenta(receptor->getTitular());
}
```

- **Paso por referencia del puntero**
- El cambio del puntero afecta al parámetro real

Paso de objetos como parámetro

```
void Banco::transferencia(Cuenta* emisor, Cuenta& receptor,
double cantidad){
   emisor->reintegro(cantidad);
   receptor.ingreso(cantidad);
   Cuenta otraCuenta;
   receptor = otraCuenta;
}
```

- **Paso por referencia del objeto `receptor`**
  - El parámetro NO ES un puntero, es un objeto!!
- El cambio en el objeto receptor afecta al objeto real que se pasa como parámetro.

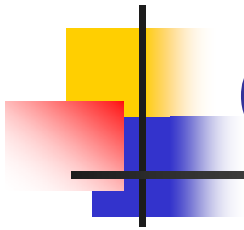


Parámetros `const`

- Pasar un objeto grande por referencia es más eficiente que pasarlo por valor, pero se corre el riesgo de modificar el objeto

- **Solución:** declarar el parámetro **`const`** indica que no se puede modificar el estado del objeto

- Si se declara un parámetro de tipo puntero como `const` significa que no se puede modificar el objeto apuntado por el parámetro.



Operadores

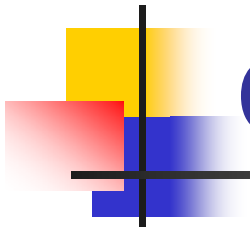
- Es posible definir operadores para las clases implementadas por el usuario
  - Por ejemplo, la suma de matrices
- Algunos operadores (=, ==) tienen un significado predefinido cuando trabaja con objetos que es posible redefinir
- La palabra clave para la definición de un operador es **`operator`**
  - `operator=`
  - `operator==`
- El operador se puede definir en el contexto de una clase (utilizando el puntero `this`) o fuera (necesita dos parámetros)

Semántica asignación (=)

```
Cuenta* cta1;          Cuenta cta3;
Cuenta* cta2;          Cuenta cta4;
```

- **Asignaciones posibles:**
  - `cta1 = cta2;` → copia de punteros (= Java)
  - `cta3 = cta4;` → copia campo a campo de los valores de `cta4` en `cta3`
  - `cta1 = &cta3;` → copia de punteros

- El programador puede **redefinir el operador** de asignación para definir la semántica de copia de objetos más adecuada para la clase.



Operador “=“ para Cuenta

```
Cuenta& Cuenta::operator=(const Cuenta& otraCuenta){
  if (this != &otraCuenta){    //¿son el mismo objeto?
    titular = otraCuenta.titular;
    saldo = otraCuenta.saldo;
    delete [] ultimasOperaciones;
    ultimasOperaciones = new double[Cuenta::MAX_OPERACIONES];
  }
  return (*this);
}
```

Semántica igualdad (==)

```
Cuenta* cta1;          Cuenta cta3;
Cuenta* cta2;          Cuenta cta4;
```

- **Semántica:**
  - `cta1 == cta2;` → igualdad de punteros, identidad (= Java)
  - `cta3 == cta4;` → Por defecto **no está definido**

- El programador puede **definir el operador** de igualdad para definir la semántica de igualdad de objetos más adecuada para la clase.

Operador “==“ para Cuenta

```
bool Cuenta::operator==(const Cuenta& otraCuenta){
  return (titular == otraCuenta.titular &&
            saldo == otraCuenta.saldo);
}

bool Cuenta::operator!=(const Cuenta& otraCta){
   return !(*this == otraCta);
}
```

- Implementar != a partir de == para evitar inconsistencias

Comparación de objetos

```
Persona* p = new Persona(“pepito", “1111111");
Cuenta cta1 (p,200);
Cuenta cta2 (p,100);
if (cta1 == cta2)
    cout<<"Los objetos cuenta son iguales"<<endl;
else
    cout<<"Los objetos cuenta son distintos"<<endl;

Cuenta* ptrCta1 = new Cuenta (p,300);
Cuenta* ptrCta2 = new Cuenta (p,300);
if (*ptrCta1 == *ptrCta2)
    cout<<"Los  objetos apuntados son iguales"<<endl;
else
    cout<<"Los objetos apuntados son distintos"<<endl;
```

Imprimir objetos en la salida estándar: operador externo “<<“

```
ostream& banco::operator<<(ostream& salida,
                           const Cuenta& cuenta){
  salida <<"Cuenta [codigo = "<<cuenta.getCodigo()
         <<", titular = "<<(cuenta.getTitular())->getNombre()
         <<", estado = "<<cuenta.getEstado()
         <<", saldo = "<<cuenta.getSaldo()
         <<"]"<<endl;
  return salida;
}

ostream& banco::operator<<(ostream& salida, const Cuenta* cta){
   salida<<(*cta);
   return salida;
}
```

Operador externo: “<<“

- Por defecto los enumerados imprimen el valor entero

```
ostream& banco::operator <<(ostream& salida,
                            const EstadoCuenta& estado){
 switch(estado){
  case OPERATIVA: salida<<"OPERATIVA"; return salida;
  case INMOVILIZADA: salida<<"INMOVILIZADA"; return salida;
  case NUM_ROJOS: salida<<"NUMEROS ROJOS"; return salida;
 }
 return salida;
}
```

Declaración de operadores

```
namespace banco{
…
class Cuenta {
public:
…
bool operator==(const Cuenta& otraCta);
bool operator!=(const Cuenta& otraCta);
Cuenta& operator=(const Cuenta& otraCta);
…
};
ostream& operator<<(ostream& salida, const EstadoCuenta& estado);
ostream& operator<<(ostream& salida, const Cuenta& cuenta);
ostream& operator<<(ostream& salida, const Cuenta* cuenta);
}
```

Operadores internos

Operadores externos

Genericidad - `template`

- Definición de una clase especificando el tipo/s mediante un parámetro
- Definición de un contenedor genérico:

```
template <class T> class Contenedor{
private:
  T contenido;

public:
  T getContenido();
  void setContenido (T elem);
};
```

Genericidad - `template`

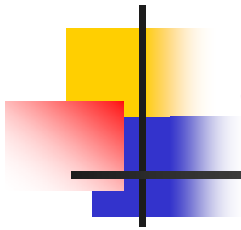
- Implementación del contenedor genérico:
  - EN EL FICHERO CABECERA!!!

```
template<class T> T Contenedor<T>::getContenido(){
  return contenido;
}

template<class T> void Contenedor<T>::setContenido(T elem){
   contenido = elem;
}
```

Instanciación del tipo genérico

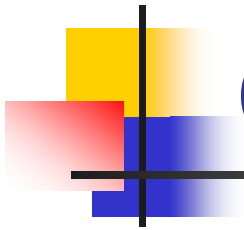
- Se indica el tipo de la clase genérica en su declaración.
- Puede ser aplicada a tipos primitivos

```
Persona* titular = new Persona(“pepito", "34914680");
Cuenta* cuenta = new Cuenta(titular);

Contenedor<Cuenta*> contenedor;
contenedor.setContenido(cuenta);
Cuenta* cta = contenedor.getContenido();
Contenedor<int> contenedorInt;
contenedorInt.setContenido(7);
```



Genericidad restringida

- No se puede restringir la genericidad
- No hace falta porque el `template` puede utilizar cualquier método sobre las entidades de tipo T
- El error lo dará en tiempo de compilación
  - Si la clase utilizada en la instanciación no dispone de los métodos utilizados en la definición de la clase genérica
  - Problemas con las llamadas a métodos si se instancia con una clase o un puntero a una clase

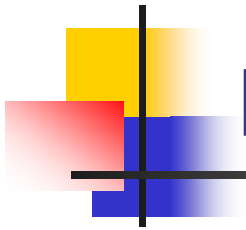


Genericidad - Críticas

- **C++ no implementa un auténtico sistema de genericidad**.

- Cuando se usa una clase genérica, se realiza un reemplazo del texto del parámetro en la declaración.

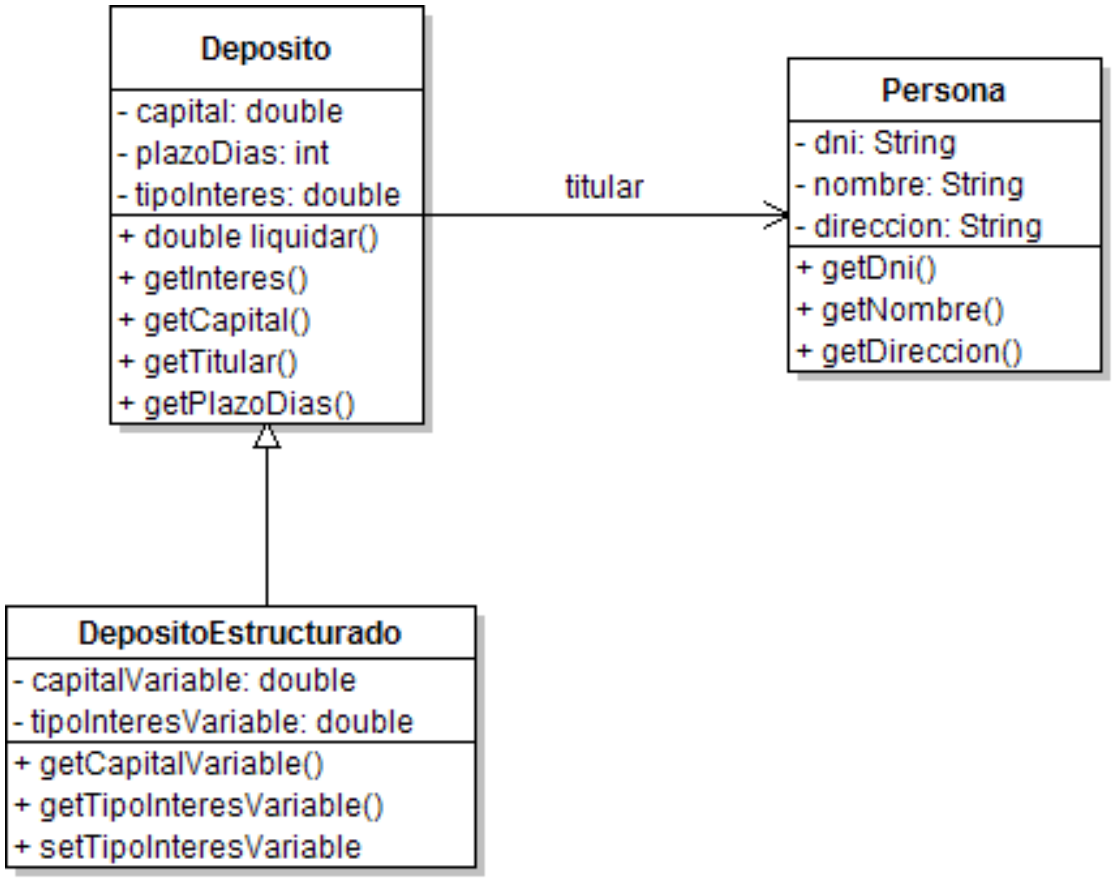
- Por tanto, **se genera código objeto para cada uno de los tipos a los que se instancie la clase genérica**.



Herencia en C++

- 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
- 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

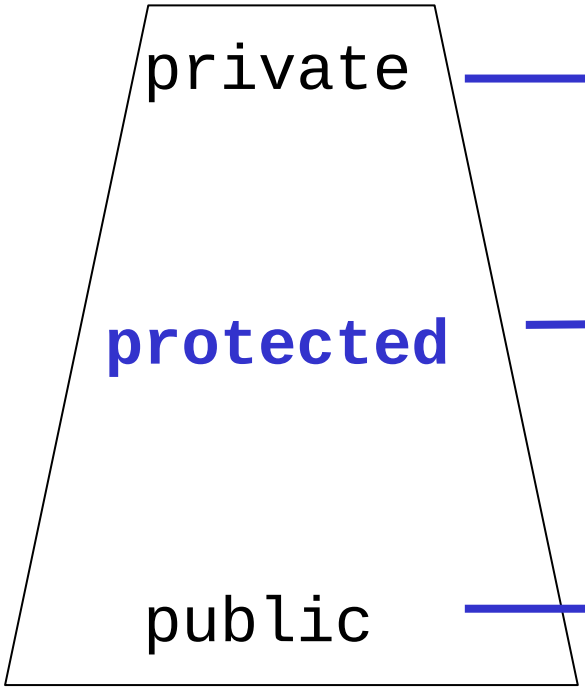
Clase Depósito

```
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();
};
```

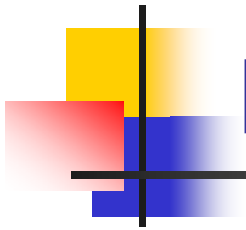
Clase Depósito Estructurado

```
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

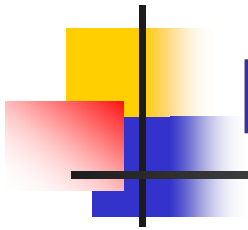


Herencia pública

```
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 y ligadura dinámica

- La clase padre debe indicar que sobre los métodos que se van a redefinir que se podrá aplicar la ligadura dinámica 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 y que se podrá aplicar ligadura dinámica.
  - **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::nombreMetodo`
  - `Deposito::getCapital();`

Redefinición de métodos

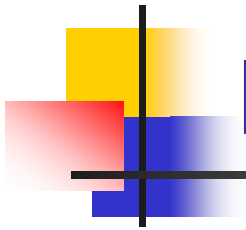
```
class Deposito {
private:
  Persona* titular;
  double capital;
  int plazoDias;
  double tipoInteres;
public:
  Deposito(…);
  double liquidar();
  virtual double getIntereses();
  virtual double getCapital();
  int getPlazoDias();
  double getTipoInteres();
  Persona* getTitular();
};
```

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++

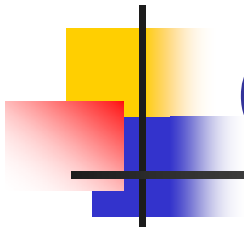
```
class Deposito {
…
public:
    virtual double getCapital();
};
class DepositoEstructurado: public Deposito{
…
public:
   double getCapital(bool tipo);
};
```

- `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);
}
```

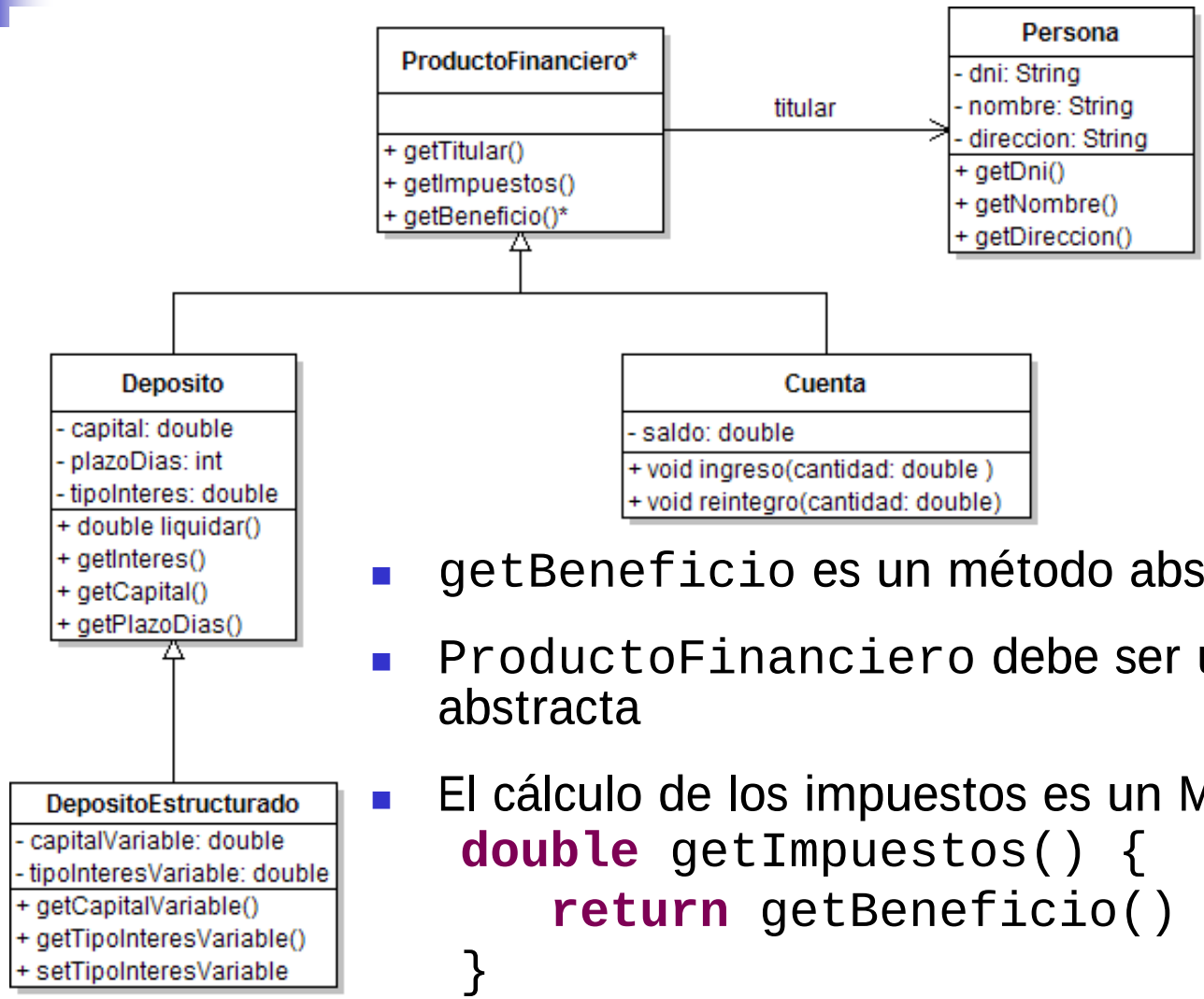
Consulta del tipo dinámico

- 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



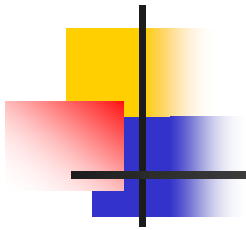
- `getBeneficio` es un método abstracto
- `ProductoFinanciero` debe ser una clase abstracta
- El cálculo de los impuestos es un Método Plantilla

```
double getImpuestos() {
    return getBeneficio() * 0.18;
}
```

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();
};
```

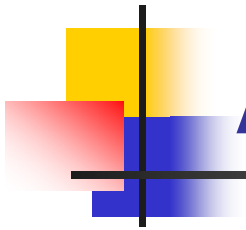


Interfaces

- C++ **no define el concepto de interfaz** de Java.

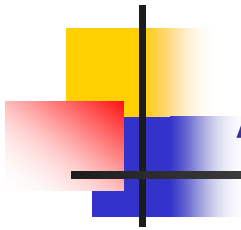
- No es necesario, ya el lenguaje ofrece herencia múltiple.
  - Si una clase quiere ser compatible con varios tipos, basta con que herede públicamente de otras clases.

- 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

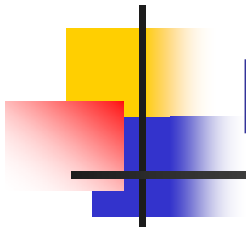
```
class CondicionCapital: public Condicion<ProductoFinanciero*>{
 private:
  double capitalUmbral;
 public:
  CondicionCapital(double capital);
  bool test(ProductoFinanciero* elemento);
};
```

```
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);
…
```



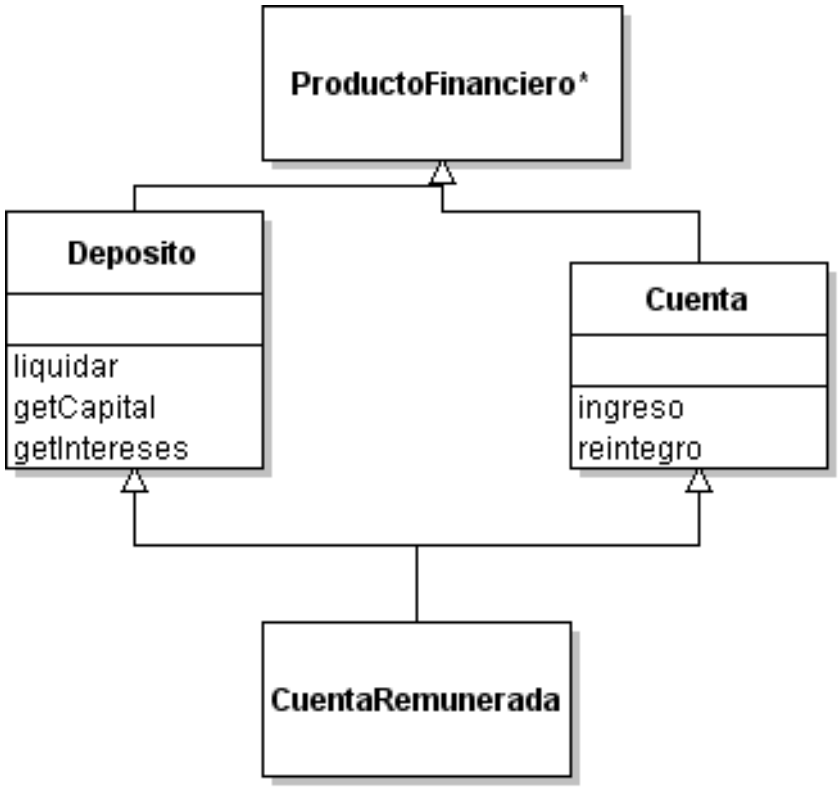
Herencia múltiple

- 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

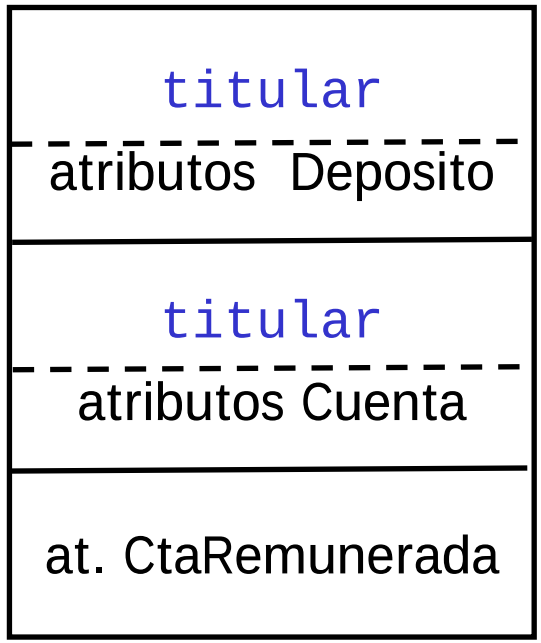
```
class CuentaRemunerada: public Cuenta, public Deposito{
        …
};
```



- `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”

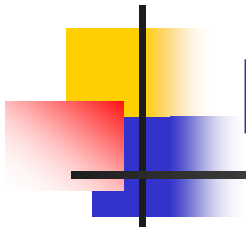
```
ProductoFinanciero* pf;
CuentaRemunerada* cr = new CuentaRemunerada(…);


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();
};
```

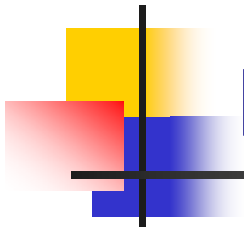
- Se evita al redefinir el método eligiendo una de las versiones:

```
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`

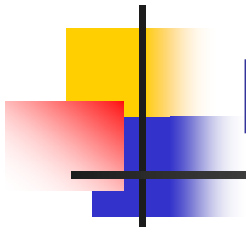


Función dominante

- Si un método de la clase `ProductoFinanciero` se redefine sólo en una de las clases hijas, no existe ambigüedad

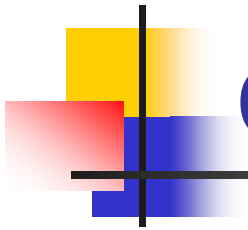
- Se dice que la versión redefinida domina sobre la versión original

- 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.



Corrección y robustez en C++

- Asertos en C++
- Mecanismo de excepciones:
  - Declaración de excepciones
  - Lanzar excepciones
  - Manejo de excepciones
  - Definición de excepciones
  - Excepciones de la librería estándar

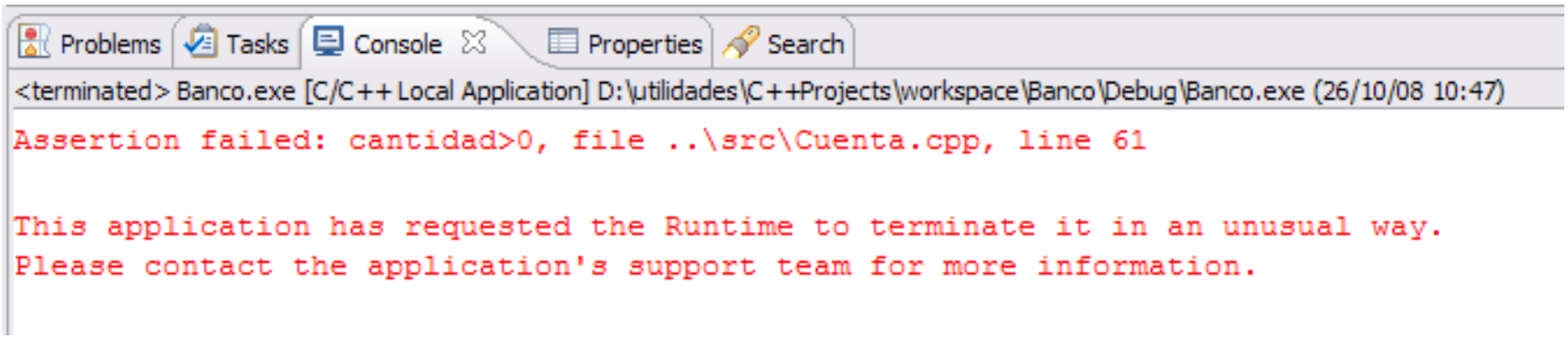
Asertos en C++

- Es posible definir puntos de chequeo en el código utilizando los asertos
  - Ayuda útil para la depuración
- Ofrece la macro `assert()` en `<assert.h>`
- `assert()` evalúa su parámetro y llama a `abort()` si el resultado es cero (`false`).

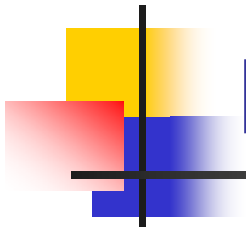
```
void Cuenta::ingreso(double cantidad){
  assert(cantidad>0);
  assert(estado == OPERATIVA);
  saldo = saldo + cantidad;
}
```

Asertos en C++

- Antes de abortar `assert` escribe en la salida de error el nombre del archivo fuente y la línea donde se produjo el error.

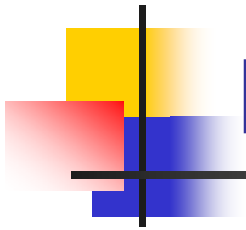


- Si se define la macro `NDEBUG`, se desactiva la comprobación de todos los asertos definidos.



Limitaciones asertos

- Aunque los asertos se pueden utilizar para controlar la corrección del código (precondiciones, postcondiciones, invariantes) tienen las mismas limitaciones que los `assert` de Java.
  - Las precondiciones no se deben evaluar con `assert`
  - No se debe abortar un programa como consecuencia de un fallo en la postcondición
- Para ayudar a la verificación de los programas C++ también existen entornos para la definición de pruebas unitarias
  - **`CPPUnit`** es uno de los entornos más utilizado



Excepciones en C++

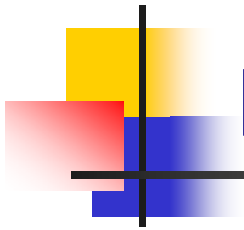
- En C++ se utiliza el mecanismo de excepciones para notificar que se ha producido una situación excepcional.
- A diferencia de Java, una excepción no tiene por qué ser un objeto de una clase:
  - Se puede lanzar “cualquier cosa” (un entero, una cadena de texto, ...)
  - No existen distintas categorías de excepciones
  - Puede ser útil definir una jerarquía de excepciones, aunque no es obligatorio

Lanzamiento de excepciones

- Se utiliza la palabra reservada **`throw`**
- Podemos lanzar un número o una cadena de texto para informar de un fallo en la precondición

→ Se desaconseja esta práctica en un programa OO

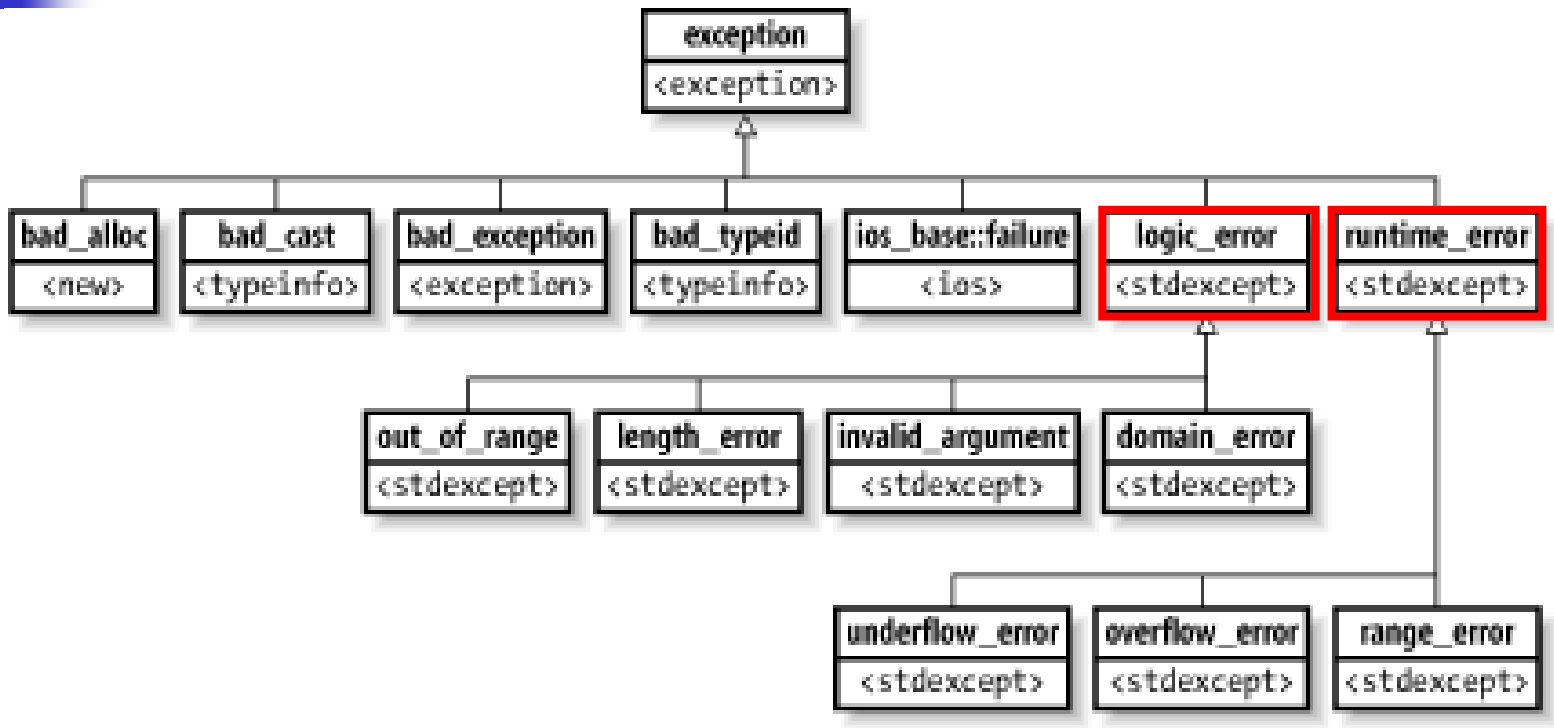
```
void Cuenta::ingreso(double cantidad){
  if (cantidad<0)
       throw cantidad;
  if (estado!= OPERATIVA)
       throw "Fallo pre. Estado incorrecto";
  saldo = saldo + cantidad;
}
```



Excepciones en la librería estándar

- Existe un conjunto de excepciones predefinidas en el espacio de nombres `std` (`<stdexcept>`)
- Todas ellas heredan de la clase `std::exception` (`<exception>`)
  - Disponen del método `what()` que devuelve la cadena de texto con el informe del error
- La clase `exception` se puede utilizar como raíz de una jerarquía de excepciones definidas por el programador, aunque no es obligatorio
  - Las nuevas excepciones deben redefinir el método `what` para que lancen el mensaje de error más conveniente.

Excepciones en la librería estándar



- **`logic_error`**: fallo en las precondiciones
- **`runtime_error`**: fallo en las postcondiciones

Uso de excepciones estándar

- Para el **control de precondiciones** se lanzan excepciones compatibles con **`logic_error`**

```
void Cuenta::ingreso(double cantidad) {
  if (cantidad < 0)
       throw invalid_argument(“Cantidad negativa");
  if (estado != OPERATIVA)
       throw logic_error("Estado incorrecto");
  saldo = saldo + cantidad;
}
```

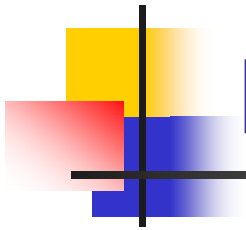
Excepciones de usuario

- Cualquier tipo de datos puede ser lanzado en una excepción.
- Se recomienda crear una clase que herede de **`runtime_error`**:

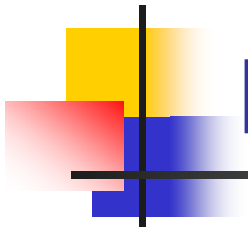
```
class RedNoDisponible: public runtime_error{
public:
 RedNoDisponible(const char* m);
};

RedNoDisponible::RedNoDisponible(const char* msg):
runtime_error(msg) {}
```

- Las excepciones que heredan de `exception` disponen del método `what()` que retorna el mensaje de error.

Declaración de excepciones



- Se utiliza también la palabra reservada **`throw`**
- Se puede especificar el conjunto de excepciones que puede lanzar un método
  - `void f(int a) throw (E1, E2);`
  - `f` puede lanzar excepciones de tipo `E1` y `E2` pero no otras
- Si no se dice nada, significa que podría lanzar cualquier excepción (o ninguna)
  - `int f();`
  - `f` podría lanzar cualquier excepción
- Se puede indicar que un método no lanzará ninguna excepción
  - `int f() throw();`
  - `f` no puede lanzar ninguna excepción (lista vacía)



Declaración de excepciones

- A diferencia de Java, **el compilador ignora la declaración de las excepciones**
  - Si en tiempo de ejecución se intenta lanzar una excepción no declarada, se detecta la violación y termina la ejecución.
  - Se aplica la política “confía en el programador”:
    - El compilador no obliga al código cliente a manejar las excepciones que puede lanzar un método
    - Si ocurre una excepción y el programador no ha definido como manejarla, la excepción escaparía del método
- Un método redefinido no puede lanzar más excepciones que las especificadas en el método de la clase padre.
  - La concordancia entre las especificaciones de la clase padre e hija si es controlada por el compilador

Declaración de las excepciones

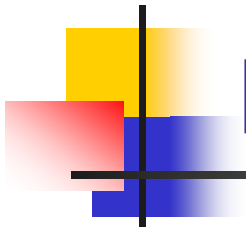
```
void Cuenta::ingreso(double cantidad) throw (logic_error){
  if (cantidad<0)
       throw invalid_argument("cantidad negativa");
  if (estado!= OPERATIVA)
       throw logic_error("Estado incorrecto");
  saldo = saldo + cantidad;
}
```

- Declaramos que el método ingreso SÓLO puede lanzar la excepción `logic_error` (y compatibles)
- Si en el cuerpo del método se produjese alguna otra excepción sería incompatible con la declaración.

Manejo de excepciones

- Como en Java, se debe definir un manejador (`catch`) por cada excepción que se espera que pueda lanzar la ejecución de un bloque de código (`try`).
  - A diferencia de Java, **no es obligatorio**, el compilador no comprueba si esto se hace o no

- Las excepciones se pueden pasar al manejador por valor o por referencia
  - Cuando las excepciones son objetos **se deben pasar por referencia para asegurar el comportamiento polimórfico**



Manejo de excepciones

- Igual que en Java, cuando ocurre una excepción se evalúan los tipos definidos en los manejadores y se ejecuta el que sea compatible
  - Hay que tener en cuenta el orden en el caso de utilizar una jerarquía de excepciones
- Se puede definir un manejador para cualquier tipo de excepción
  - `catch(…)`
- Es posible relanzar la misma excepción que se está manejando
  - `throw;`

Método visualizar del navegador web

```
void Navegador::visualiza(string url){
  Conexion* conexion;
  int intentos = 0;
  while (intentos < 20) {
    try {
        conexion = new Conexion(url);
         break;
    } catch (RedNoDisponible& e) {
         intentos++;
         if (intentos == 20) throw; //relanza
    }
  }
  //Se ha abierto la conexión y se leen las líneas …
}
```

- Si al crear la conexión ocurren las excepciones `ServidorNoEncontrado` o `RecursoNoDisponible` (no manejadas) pasarán al cliente

Código cliente

```
int main(){
  Navegador navegador;
  try{
     navegador.visualiza("http://www.poo.c++");
  }catch(ErrorConexion& e){
     cout<<"Fin del programa. "<<e.what()<<endl;
  } //maneja cualquiera de las subclases
  catch(...){  //maneja cualquier otra excepción
     cout<<"Fin por excepción no prevista"<<endl;
  }
}
```