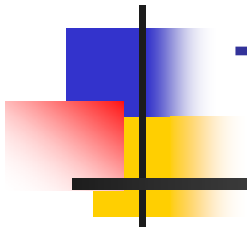


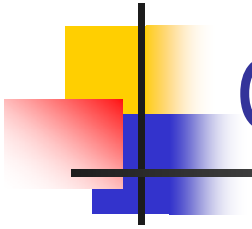
Tema 2: Clase y objetos en C++

Programación Orientada a Objetos

Curso 2008/2009

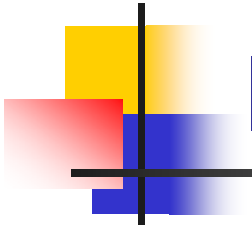
Begoña Moros Valle

DIS
Departamento de
Informática y Sistemas



Contenido

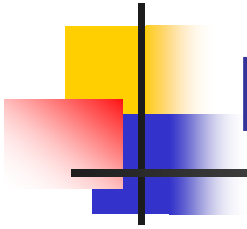
- Introducción
- Módulos en C++:
  - Clases
  - Estructuras (`struct`)
  - Espacios de nombres (`namespace`)
- Semántica referencia
  - Semántica de los operadores “`==`“ e “`=`“
- Métodos y mensajes
- Creación y destrucción de objetos
- Genericidad → `template`



Introducción

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

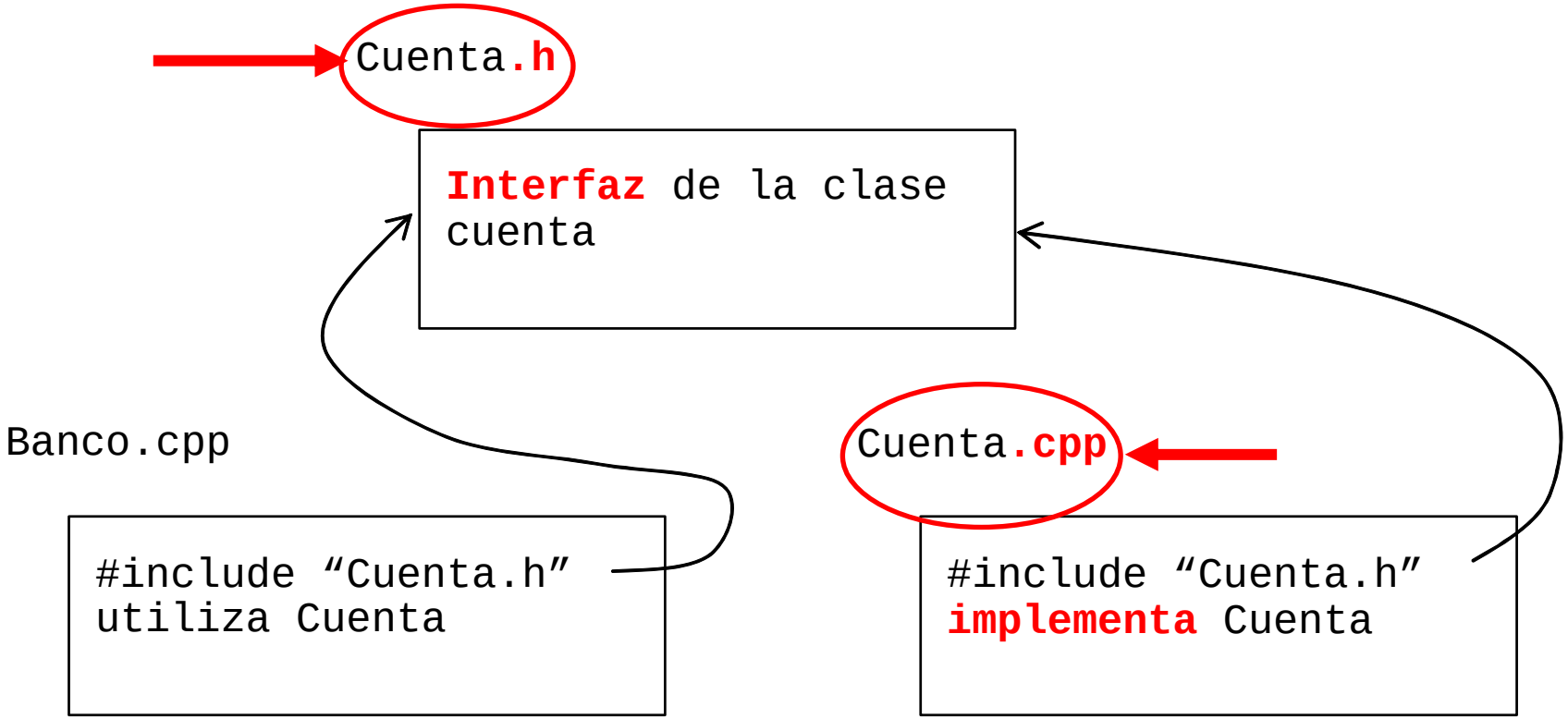
  ➔ Es un **lenguaje híbrido**.
- Ha sido inspiración de lenguajes de programación posteriores como Java y C#.
- A finales de los 90 fue estandarizado: **ISO C++**
- Las librerías del lenguaje son escasas. La librería más notable es **STL** (*Standard Template Library*).
- Actualmente, sigue siendo un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



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)

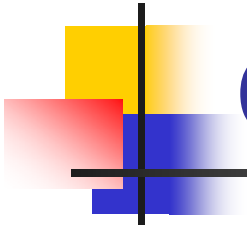
```
#include "Persona.h“


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);
};
```

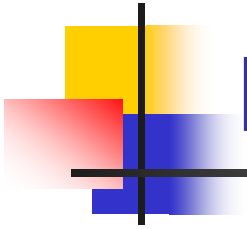
Cuenta.cpp

```
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++

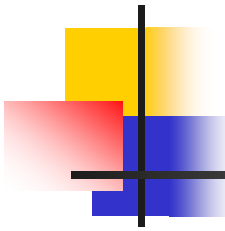
- Se pueden definir tanto atributos y métodos de clase (**`static`**) como de instancia (= Java).
- Palabra reservada **`const`**
  - Indica que un **atributo** es **inmutable**
    - Equivalente a atributos `final` en Java
    - **`const`** `Persona* titular;` → puntero inmutable
    - `Persona*` **`const`** `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`

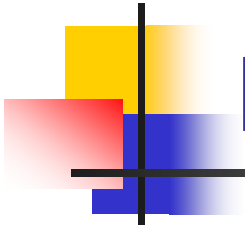
```
class NodoArbol {
        friend class Arbol;
        private:
          int valor;
          NodoArbol decha;
          NodoArbol izda;
                …
};
```

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

Concepto de estructura

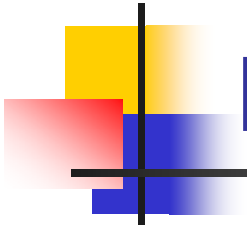
- Unidad modular heredada de C
  - en C++ se amplía con la definición de funciones
- Totalmente equivalente al concepto de clase salvo:
  - Cambia la palabra `class` por **`struct`**
  - Por defecto todos los elementos son públicos salvo que se diga lo contrario.

```
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 banco;`

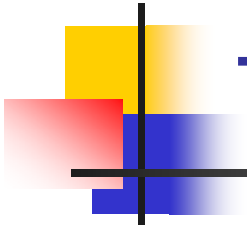
Espacio de nombres

- En `Cuenta.h`

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

- En `Banco.h`

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

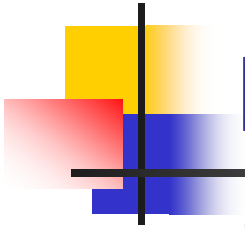


Tipos del lenguaje

- Tipos de datos primitivos:
  - `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `bool`, `etc.`
- Enumerados:
  - `enum {OPERATIVA, INMOVILIZADA, NUM_ROJOS};`
- Objetos embebidos:
  - Subojetos
- 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

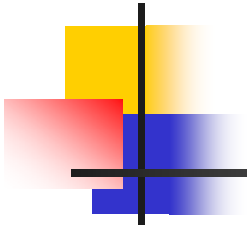
```
namespace banco{
  enum EstadoCuenta{
       OPERATIVA, INMOVILIZADA, NUM_ROJOS
     //OPERATIVA == 0, INMOVILIZADA == 1, NUM_ROJOS == 2
  };

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

Enumerados. Uso

- En `Cuenta.cpp`

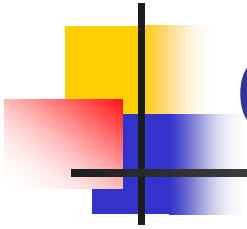
```
#include "Cuenta.h"

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



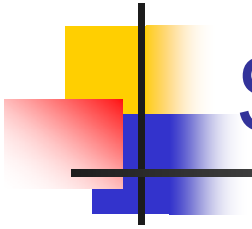
Constructores

- 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

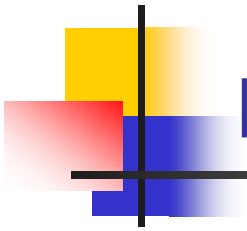
```
Cuenta::Cuenta (Persona *persona){
 codigo = ++ultimoCodigo;
 titular = persona;
 saldo = SALDO_MINIMO;
 ultimasOperaciones = new double[Cuenta::MAX_OPERACIONES];
 numOperaciones = 0;
 estado = OPERATIVA;
}
Cuenta::Cuenta (Persona *persona, double saldoInicial){
 codigo = ++ultimoCodigo;
 titular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



Sobrecarga de constructores

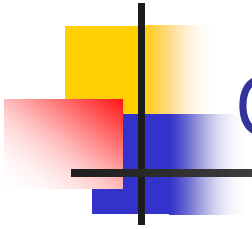
- A diferencia de Java, **`this` no se puede utilizar como una función** para reutilizar el código de los constructores

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



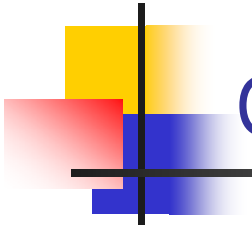
Reutilización de código en constructores

```
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

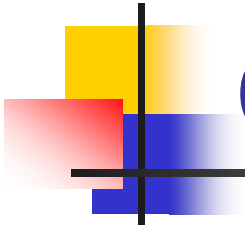
- Para la clase `Cuenta` definimos **un único constructor** con dos argumentos:
  - `titular`: obligatorio pasarlo como parámetro
  - `saldo`: que tiene como valor por defecto el saldo mínimo.
- No necesitamos dos constructores sobrecargados en la clase `Cuenta`.

Constructor para la clase Cuenta

- Fichero cabecera:

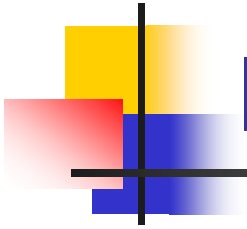
```
class Cuenta {
public:
Cuenta (Persona *titular, double saldoInicial= SALDO_MINIMO);
…
};
```

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



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

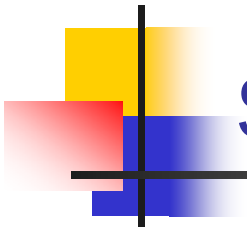
Destructor para la clase Cuenta

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

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

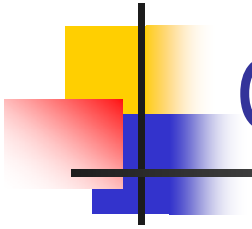
- Implementación (Cuenta.cpp):

```
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



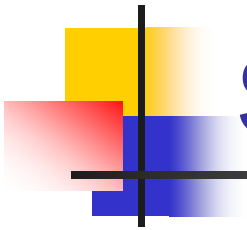
Creación de objetos compuestos

- 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;
…
};
```

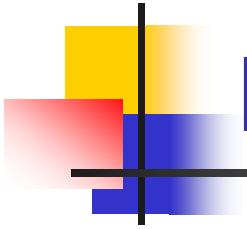
- Si no existe el constructor por defecto en la clase `Persona` este código no compila



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

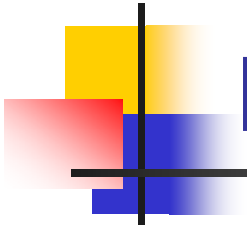
```
Cuenta  objCta;
Cuenta* ptroCta;
```

- **Notación punto** para objetos

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

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

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

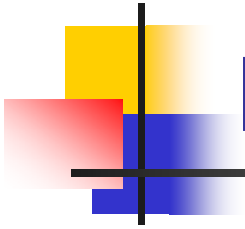


Paso de parámetros

- Soporta paso de parámetros por valor y por referencia

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

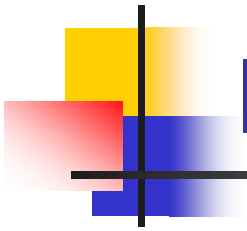
  - `val` se pasa **por valor** → `val` incrementa una copia del parámetro real (= Java)
  - `ref` se pasa **por referencia** → `ref` incrementa el parámetro real



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

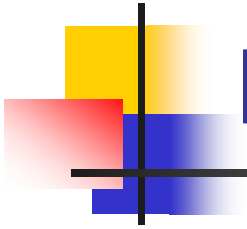
```
void Banco::transferencia(Cuenta* emisor, Cuenta*& receptor,
double cantidad){
  emisor->reintegro(cantidad);
  receptor->ingreso(cantidad);
  receptor = new Cuenta(receptor->getTitular());
}
```

- **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.

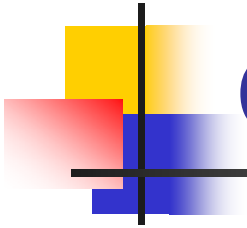


Parámetros `const`

- 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

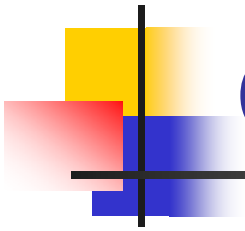
- Es posible definir operadores para las clases implementadas por el usuario
  - Por ejemplo, la suma de matrices
- Algunos operadores (=, ==) tienen un significado predefinido cuando trabaja con objetos que es posible redefinir
- La palabra clave para la definición de un operador es **`operator`**
  - `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;
}
```

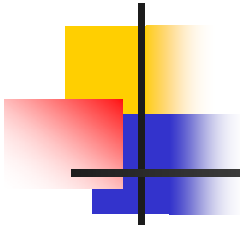
Declaración de operadores

```
namespace banco{
…
class Cuenta {
public:
…
bool operator==(const Cuenta& otraCta);
bool operator!=(const Cuenta& otraCta);
Cuenta& operator=(const Cuenta& otraCta);
…
};
```

Operadores internos

Operadores externos

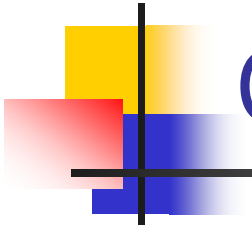
```
ostream& operator<<(ostream& salida, const EstadoCuenta& estado);
ostream& operator<<(ostream& salida, const Cuenta& cuenta);
ostream& operator<<(ostream& salida, const Cuenta* cuenta);
}
```



Genericidad - `template`

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

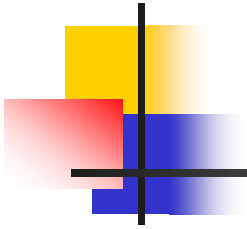
```
template <class T> class Contenedor{
private:
  T contenido;

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



Genericidad - `template`

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

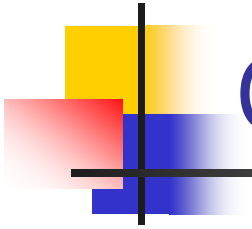
```
template<class T> T Contenedor<T>::getContenido(){
  return contenido;
}

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



Instanciación del tipo genérico

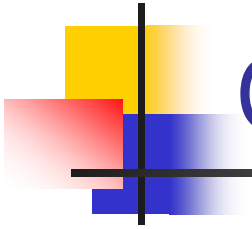
- Se indica el tipo de la clase genérica en su declaración.
- Puede ser aplicada a tipos primitivos

```
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**.