# POO en C++

Antonio Espín Herranz

# Creación de Clases

- Una clase es un tipo definido por el usuario que se compone de atributos o propiedades y métodos o funcionalidades.
- Los atributos determinarán el estado interno del objeto.
- Los métodos o funcionalidades el conjunto de operaciones que puede realizar los objetos que pertenezcan a la clase.

# Sintaxis

```
// Esta definición se almacenará en un fichero .h.

class nombreClase {
    // Miembros privados.
    private:

    // Miembros protegidos:
    protected:

    // Miembros públicos:
    public:

};  // Miembros pueden ser atributos o métodos.
```



PRÁCTICA CLASE PUNTO2D

# Atributos de la clase

- Constituyen la estructura interna de la clase.
- class Punto2D {
  private:
  int x, y;
  };
- No es posible asociarles un valor inicial a no ser que se definan como **const** o **static**.
- Los atributos pueden ser tipos simples u objetos de otras clases.
- Una clase no puede ser atributo de ella misma pero si puede ser un objeto.

```
class Punto2D {
    private:
        int x, y;
        Punto2D *p;
};
```

# Los métodos de la clase

- Son las operaciones que nos ofrece la clase y sólo podemos trabajar a través de estas.
- Los métodos no se pueden anidar.
- Y tienen los mismo tipos de acceso que lo atributos.

# Acceso a los miembros de la clase

- Una buena practica es **NO permitir** al usuario de la clase acceder directamente a los atributos de la clase.
- Se suele ocultar la representación de la misma y para poder acceder a los atributos de esta se ofrecen métodos para leer y escribir dichos atributos.
- Esto permite también modificar la estructura interna de la clase sin que el usuario se entere, siempre y cuando mantengamos el interfaz de los métodos.

# 3 niveles de acceso

- Los miembros de la clase nos ofrecen 3 niveles de acceso:
  - **private**: Sólo se puede acceder a dichos elementos desde el interior de la propia clase, por ejemplo, desde un método de la propia clase. Tampoco será accedido desde clases heredadas.
  - **protected**:  Igual que el privado, pero en el caso de la clases hijas si que pueden acceder.
  - **public**: puede ser accedido por un objeto de la clase desde cualquier parte de la aplicación.

# Estructura de las clases en C++

- En C++ podemos separar la definición de la clase de la implementación.
- En un fichero de cabecera .h, se define los atributos y los prototipos de los métodos.
- En un fichero .cpp se implementan dichos métodos.
- Tendremos que hacer un #include “fichero.h”.

# Ejemplo

```
#if !defined( _CIRCULO_H_ )
#define _CIRCULO_H_

class Circulo
{
  // miembros privados
  private:
    double x, y;      // coordenadas del centro
    double radio;     // radio del círculo

  // miembros protegidos
  protected:
    void msgEsNegativo();

  // miembros públicos
  public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r); //
       constructor
    double longCircunferencia();
    double areaCirculo();
};

#endif // _CIRCULO_H_
```

```
#include <iostream>
#include "circulo.h"
using namespace std;

void Circulo::msgEsNegativo()
{
  cout << "El radio es negativo. Se convierte a
       positivo\n";
}

Circulo::Circulo(double cx, double cy, double r) {
  x = cx; y = cy;
  if (r < 0)
  {
    msgEsNegativo();
    r = -r;
  }
  radio = r;
}

double Circulo::longCircunferencia(){
  return 2 * 3.1415926 * radio;
}

double Circulo::areaCirculo(){
  return 3.1415926 * radio * radio;
}
```

# La funciones en línea

- Dentro de la definición de la clase se pueden definir funciones **inline**.

- Se suele aplicar cuando el código de la función ocupa una línea, normalmente para las funciones que manejan los atributos de la clase.

- Dentro del .h, en la sección public:

```
inline double getX(){ return x; }
inline double getY(){ return y; }

inline void setX(double unX){ x = unX; }
inline void setY(double unY){ y = unY; }
```

# Sobrecarga de métodos

- Dentro de la POO también está permitido nombrar a dos métodos con el mismo nombre dentro del mismo ámbito (dentro de la misma clase).

- Se deben de llamar igual pero deben diferir en al menos un parámetro ya sea con el tipo del parámetro o en número de parámetros.

- La sobrecarga no se puede indicar sólo en el valor retornado.

# Ejemplo

```
class  MiClase {
 private:
    int a, b;

public:
    void metodo1()   // Es correcta la sobrecarga
    void metodo1(int a);
    void metodo1(int a, int b);

    void metodo2();  // No es correcto, sólo difieren en el
    int metodo2();   // tipo devuelto.
```

# Parámetros por omisión

- Dentro de un método se pueden definir parámetros por defecto u omisión.
- Definición:
  - void asignarPunto(int x = 0, int y = 0);
- Este método se puede llamar con 0, 1 o 2 parámetros.
- En caso de omitir un parámetro se sustituirá por 0.
  - objetoPunto.asignarPunto();
  - objetoPunto.asignarPunto(1);
  - objetoPunto.asignarPunto(2,3);

# El puntero this

- Los atributos de los distintos objetos son independientes unos de otros y cada objeto mantiene una copia de sus atributos.

- En el caso de los métodos se almacenan en la clase y son comunes a todos los objetos de esa clase.

- El puntero this hace referencia al propio objeto que recibe un mensaje.

```
void Punto2D::asignarPunto(int x, int y){
    this→x = x;
    this→y = y;
  }

  // this→x representa la coordenada x del punto (pertenece al objeto),
     x es el
  // parámetro que recibe el método.

  // Un objeto de la clase Punto2D recibe el mensaje: asignarPunto
```

# Métodos y Objetos **const**

- Si definimos un objeto como **const** al compilar la aplicación si intentamos modificarlo nos dará un error de compilación.
  - const Punto2D p;

- Al definir un objeto constante el método que se aplica a dicho objeto también se tiene que definir como const.
  - En la definición del prototipo: (.h)
    - void asignarPunto(int x, int y) **const**;
  - En la implementación (.cpp):
    - void Punto2D::asignarPunto(int x, int y) **const** { … }

- Si el objeto **no es const**, el método puede ser o no.

# Caso especial

- Si aún así queremos tener la posibilidad de poder modificar un atributo en un objeto constante.
- Definir dicho atributo como **mutable**.
- mutable permite la modificación con objetos definidos como constantes.
- class Punto2D {

```
 private:
     mutable int x; // Se permite modificar la componente X.
     int y;
 };
```

# Sobrecarga en métodos **const**

- También se permite la sobrecarga de un método mediante const.
- Es decir, podemos tener dos versiones del mismo método una para objetos const y otra para objetos no const.
  - void asignarPunto(int x, int y) const;
  - void asignarPunto(int x, int y);
- Siendo el compilador el encargado de llamar al método que corresponde.
- Un método const no puede devolver una referencia:
  - int&  Punto2D::asignarPunto(int x, int y) const → **ERROR**
- Pero si puede devolver una referencia a un objeto const.
  - const int& Punto2D::asignarPunto(int x, int y) → **OK**

# Creando objetos

- Los métodos mas importantes en una clase son los constructores / destructores.
- Por defecto, C++ añade un constructor y un destructor por defecto.
- Los constructores se utilizan para inicializar los objetos y los destructores para liberar dichos objetos.
- Estos métodos son de acceso **public**.
- Si el constructor proporcionado no nos vale podremos sobrescribirlo.
- El constructor se invoca cuando definimos una variable de una determinada clase.
- Punto2D miPunto; // Salta el constructor por defecto.
- El destructor también, cuando el objeto deja de utilizarse.

# Ejemplo

```
class  Punto2D {
 private:
    int x, int y;

 public:
    Punto2D();
    ~Punto2D();

 // A nivel de implementación estos métodos por defecto
    están vacíos.
 };
```

Ambos métodos coinciden en nombre con la clase.

# Constructores

- Se utilizan para definir los valores iniciales de los objetos.
- Los constructores no se heredan, no pueden devolver ningún tipo (tampoco void).
- No pueden ser declarados como const, static o virtual.
- El constructor también se puede sobrecargar.

# Mas sobre Constructores

- Pueden tener valores por defecto como cualquier otro método.
- Cuando definimos un constructor este sustituye al constructor que añade el compilador por defecto.
- Lo podemos mantener pero habrá que definirlo.
  - Por ejemplo:

    Punto2D(int x=0, int y=0);

  - Invocación:

```
Punto2D p;        // Se le invoca sin parámetros.
Punto2D p(3)      // Se creará el punto (3,0);
Punto2D p(3,4)    // Se creará el punto (3,4);
```

# Formas de invocar al Constructor

- Al declarar un object o Global:
  - Punto2D p;
- Declarando un objeto local dentro de un método:
  - Punto2D p(-1, 5);
  - Es lo mismo que:
    - Punto2D p = Punto2D(-1, 5);
- Invocando al operador new:
  - Punto2D *p = new Punto2D(6,7);
- Al retornar de un método:
  - return Punto2D(7,8); // Crea una copia y la retorna.

# Desde un constructor a otro

- Ejemplo (sólo en el .CPP):
  - CFecha::CFecha(int dd, int mm, int aaaa) **:**
    **dia(dd), mes(mm), anyo(aaaa){ … }**

El propio constructor llama a los constructores de los atributos

Disponiendo de la clase:

```
class Cfecha {
  // Atributos
  private:
    int dia, mes, anyo;
 … };
```

# Operador = de una clase

- Por defecto C++ también incorpora un operador de asignación.
- Ejemplo:

```
Punto2D p(1, 4);
Punto2D q;
q = p;
```

- Copia los atributos uno a uno.
- Devuelve una ref. por eficiencia.
- Su cabecera es:

```
Punto2D& Punto2D::operator=(const Punto2D& punto){
    x = punto.x;
    y = punto.y;
    return *this;
}
```

# Constructor copia

- Otra forma de iniciar un objeto es asignándole otro objeto de su misma clase en el momento de su creación.

- C++ proporciona para cada clase un constructor público por omisión, recibe el nombre de constructor copia y que recibe como parámetro otro objeto de la misma clase.

- Su cabecera es:

```
Punto2D::Punto2D(const Punto2D& punto){
    *this = punto;
}
```

# Destrucción de objetos

- Todos los objetos que creamos mediante el operador new tienen que ser destruidos mediante el operador delete.

- El destructor:
  - Coincide con el nombre de la clase precedido de una tilde. ~
  - Coincide con el nombre de la clase.
  - No puede retornar ningún tipo (ni void).
  - No puede ser declarado como const ni static, pero si puede ser virtual (podremos destruir objeto sin conocer su tipo).
  - Se añade uno por defecto.

```
Punto2D *p = new Punto2D(8, 6);
delete p;
```

PRACTICA COMPOSICIÓN

# Punteros como atributos de una clase

- Si definimos una clase que tiene un atributo que es un puntero vamos a necesitar en algún punto:
  - Reservar memoria → constructor
  - Y liberar memoria → destructor

- En este tipo de clases tenemos que tener cuidado para implementar los constructores, constructor copia, operador = y el destructor.

- Al trabajar con punteros no podremos hacer asignaciones directamente de los atributos del objeto, en ese caso habrá que hacer nuevas reservas de memoria y copiar los elementos.

# Ejemplo

- Podemos implementar un array dinámico.
- Lo suyo es definirlo como un puntero y no limitar el espacio.
- class Vector {

```
 private:
    int n;
    double *v;
 };
```

El constructor debe asignar memoria con new.

Y el destructor liberarla con delete [] porque es un array.

# La reserva de memoria

- Se puede hacer así:
- try {

```
double *p = new double[n];
return p;
} catch (bad_alloc e){
   cout << “sin memoria”;
   exit (-1);
}
```

PRACTICA VECTOR

# Atributos y métodos static

- Los atributos static son variables compartidas por todos los objetos de una clase.
  - Se puede acceder a ellos a través de la clase sin necesidad de crear un objeto.

- O acceder a un método sin crear el objeto previamente → métodos static.

# Atributos y métodos static

- Los atributos static tienen los tres niveles de acceso: public, protected y private.
- Si tienen acceso public pueden ser accedidas desde fuera (de la clase) mediante el nombre de la clase.
  - nombreClase::varStatic

- En el caso de los métodos no se dispone de this.
- Hay que tener en cuenta que se les llaman sin crear un objeto de la clase.
- Igual que las variables se les llama a través del nombre de la clase y el operador de resolución de ámbito.
  - nombreClase::nombreMetodoStatic

# Ejemplo

```
// Definición:
class CFecha {
        private:
            int dia, mes, anyo;
            static CFecha fechaPorOmision;
        public:
             // resto de métodos.
             static void obtenerFechaActual(int&, int&, int&);
        };

        // Implementación:
        void CFecha::obtenerFechaActual(int& dd, int& mm, int& aaaa){
          // Obtener la fecha actual.
          struct tm *fh;
          time_t segundos;
          time(&segundos);
          fh = localtime(&segundos);
          dd = fh->tm_mday; mm = fh->tm_mon; aaaa = fh->tm_year+1900;
        }

        // Llamada:
        int d, m, a;
        Cfecha::obtenerFechaActual(d,m,a);
```

# Ejemplo Singleton

```
class IdiomaSingleton
{
    private:
        static IdiomaSingleton *instance;
        map<string,string> mapa;
        string id;

    public:
        static IdiomaSingleton *getInstance(string id);
        string get(string clave);
        virtual ~IdiomaSingleton();

    protected:
        IdiomaSingleton(string idioma);
};
```

En el CPP → IdiomaSingleton *IdiomaSingleton::instance = NULL;

Después la implementación de todos los métodos!!

# Atributos que son objetos

- Podemos tener objetos de otras clases que son atributos en este caso estamos estableciendo una **relación de composición** con los objetos.

- Por ejemplo, podemos definir un objeto recta a partir de dos puntos, es decir, una recta se compone de dos puntos.

# Ejemplo

- La clase Punto:

```
class Punto  {

private:
    int x,y;

public:
    Punto();
    Punto(int x, int y);
    inline int getX(){ return x; }
    inline int getY(){ return y; }
    ~Punto();

};
```

- La clase Recta:

```
class Recta  {
private:
    Punto A, B;

public:
    Recta();
    Recta(Punto a, Punto b);
    ~Recta();

};

// El constructor: puede ser así:
Recta::Recta(Punto a, Punto b): A(a), B(b){}
// O:
Recta::Recta(Punto a, Punto b){
    this->A=a;
    this->B=b;
}
```

# Restricciones

- Una clase no puede estar ella misma de atributo:
- A no ser que sea un atributo static.
- O un puntero.

```
class Punto {
private:
    Punto a; // ERROR
    Punto *p // OK
    static Punto O; // OK
};
```

# Clases internas

```
class Circulo {
  private:
    class Punto    {
      private:
        double x, y;
      public:
        Punto(double cx = 0, double cy = 0) { x = cx; y = cy; }
        double X() const { return x; }
        double Y() const { return y; }
    };
    Punto centro; // coordenadas del centro
    double radio; // radio del círculo
     …
};
```

Una clase interna es una clase que es miembro de otra clase.

Solo hacerlo cuando tenga sentido.

Una ventana define su propio cursor

La clase Punto tendrá acceso a todos los miembros de Circulo

# Integridad de los datos

- Una forma de garantizar la integridad de los datos es definir los atributos como privados y métodos públicos para poder acceder a estos.
- Mediante estos métodos podemos llevar el control, de los valores, solo permitir ciertos valores, etc.
- Si en los métodos de asignación del objeto devolvemos referencias, vamos a poder utilizarlos también en expresiones a parte de asignaciones.
  - Cfecha& fechaNacimiento(){ return fecha_nacimiento; }

# Devolver puntero / Devolver referencia

- Cuidado, si un método de una clase devuelve un puntero a un atributo privado de la clase, se vulnera la seguridad y la encapsulación.

```
// Siendo nombre un atributo de la clase.
char *Ccumpleaños::obtenerNombre() const { return nombre; }
```

- Podríamos:
  - char *pnom = p2.obtener
  - strcpy(pnom, “hola”);
  - Lo mejor: devolver un puntero a un objeto constante.
  - const char * …
  - Si devolvemos una referencia que sea una referencia a un objeto constante.

# Matrices de Objetos

- Podemos definir matrices de objetos como cualquier otro tipo.
- Pueden ser estáticas o dinámicas.
- Las dinámicas se crean con new y se destruyen con delete.
- En orden inverso al de creación.

# Ejemplo

**// MATRIZ ESTÁTICA DE OBJETOS**
**// PUNTO:**

```
Punto puntos1[10];
int i;

for (i=0; i < 10 ; i++)
        puntos1[i] = Punto(i,i);

cout << "listado de puntos1: " << endl;
for (i= 0; i < 10; i++){
        puntos1[i].imprimir();
        cout << endl;
}
```

**// MATRIZ DINÁMICA DE OBJETOS**
**// PUNTO:**

```
Punto *puntos2 = new Punto[10];
for (i=0; i < 10 ; i++)
        puntos2[i] = Punto(i,i);

cout << endl << "listado de puntos2: "
<< endl;
for (i= 0; i < 10; i++){
        puntos2[i].imprimir();
        cout << endl;
}

delete [ ]puntos2;
```

# Ejemplo

**// MATRIZ DE PUNTEROS A OBJETOS**
**// PUNTO:**

```
Punto *puntos3[10];

for (i=0; i < 10 ; i++)
        puntos3[i] = new Punto(i,i);

cout << endl << "listado de puntos3: "
<< endl;
for (i= 0; i < 10; i++){
        puntos3[i]→imprimir();
        cout << endl;
        delete puntos3[i];
}
```

**// MATRIZ DINAMICA DE PUNTEROS A**
**// OBJETOS PUNTO:**

```
Punto **puntos4 = 0;

puntos4 = new Punto *[10];
for (i=0 ; i < 10 ; i++)
        puntos4[i] = new Punto(i,i);

cout << endl << "listado de puntos4: "
<< endl;
for (i= 0; i < 10; i++){
        puntos4[i]→imprimir();
        cout << endl;
        delete puntos4[i];
}

delete [ ]puntos4;
```

# Funciones amigas

- Métodos ordinarios de una clase:
  - Acceden al resto de miembros.
  - Pertenecen al ámbito de la clase.
  - Debe invocarse para un objeto de su misma clase, a través de this.
- Los estáticos no pueden hacer uso de this pero las dos primeras si las cumplen.
- En el caso de **funciones externas** no cumplen ninguna de las tres.
  - Pero si necesitamos que una función externa tenga acceso a los miembros de una clase la podemos definir en la clase como **friend**.
- Las funciones friend no se ven afectadas por los modificadores de acceso: private, protected y public.

# Ejemplo

- En la clase Vector:
  - Podemos implementar una función externa que reciba un objeto Vector e imprima el contenido.
  - Dentro de la clase Vector:
  - // Definición:

**friend** void visualizar(const Vector& v);

// En la implementación de la función (fuera de la clase, por ej en el fichero principal donde esté main()):

```
void visualizar(const Vector &v){
    for (int i=0 ; i < v.longitud ; i++)
        …
} // Accede a los miembros de la clase Vector.
```

**// Este mecanismo se utiliza en la sobrecarga de operadores.**

# Funciones amigas entre clases

- Una función externa puede ser amiga de varias clases, si fuera necesario.
- Podemos definir métodos friend de otras clases, no tienen porque ser siempre funciones externas.
- En la clase C1: friend C2:metodo(C1&);
- O tener acceso a todos los miembros de otra clase:

```
class C1 {
    ...
};
```

```
class C2 {
        friend class C1;
};
```

# Punteros a miembros de una clase

- La asignación y declaración de funciones a métodos varia con los punteros a funciones externas:

- En una función externa:
  void funcion(int *)
  El prototipo es: void (int *)
  Un puntero a la función **void (*)(int *)**

- Ejemplo:

```
int x;
void funcion(int *p);          // Prototipo de la función.
void (*pfun)(int *);           // Puntero a la función.
pfun = funcion;                // Asignación de la función al puntero.
```

# Punteros a miembros de una clase

- En el caso del método: No se puede asignar igual.

- El puntero tiene que ser del mismo tipo.

- Ejemplo:

  En la clase C, tenemos el método:

  void C::funcion(int *)

  El tipo es: void C:: (int *)

  Y el puntero es: **void (C::*)(int *)**

# Ejemplo

```
class CNotas {
      private:
            float nota;
      public:
        CNotas(float n=0): nota(n){};
        void asignarNota(float n);
        float ObtenerNota() const;
};
```

- No se pueden definir puntero a miembros static.

```
// Definimos el tipo puntero
// al método asignarNota
typedef void (CNotas::*pf)(float)

// El puntero al método:
pf punteroF = &CNotas::asignarNota;

// Acceso desde un objeto: .*
// objeto.*puntero_a_miembro

CNotas alumno;
(alumno.*punteroF)(nota);

// Acceso desde un puntero a objeto: .→
// puntero_a_objeto→*puntero_a_miembro

CNotas *palumno = new Notas();
(palumno→*punteroF)(nota);
```

# Sobrecarga de Operadores

- Sobrecargar un operador significa que puede desarrollar su función en varios contextos diferentes sin necesidad de otras operaciones adicionales.

- A+B supone operaciones diferentes si estamos trabajando con números enteros o complejos → El operador + está sobrecargado.

- Los operadores >> y << utilizados en los flujos cin y cout también están sobrecargados.

# Sobrecargar un operador

- En C++ podemos asociar una función o método a un operador estándar.
- El compilador llama a la función cuando detecta un operador en un determinado contexto.
- Está sobrecargado si podemos operar con mas de un tipo de objetos.

# Sintaxis

- tipo operator operador([parámetros]);
- Tipo: valor retornado de la función.
- Operador: cualquiera **de la tabla**. Salvo:
  **::** op. De ámbito.
  **.** Selección de un miembro.
  **.*** idem pero mediante un puntero.
  **?:** operador condicional.
  **sizeof**: tamaño de.
  **typeid**: Id. De tipo.

OJO, tampoco nos podemos inventar operadores nuevos

# Parámetros

- Si se sobrecarga un operador unario utilizando una función externa, esta debe tomar un parámetro.

- Y dos cuando se sobrecarge un operador binario. De la misma forma con una función externa.

# Ejemplo

```
class C { … }
// Funciones externas:
   C operator – (C) ; // - Unario.
   C operator – (C, C); // - Binario.
   int main(){
        C a, b, c;
        b = -c;        // Invoca a la función – unario.
        c = a – b;     // Invoca a la función – Binario.
   }
```

# Dentro de la clase

- Si la sobrecarga del operador se realiza con métodos de la clase, perdemos un parámetro de la función. Ya que actúa el puntero implícito this.

- class C {

```
public:
    C operator – ();
    C operator –(C); // La llamadas se aplican igual que
                     // antes.
}
```

# Dos formas de llamar al operador

- Estas dos formas son válidas:
  b = -c; → b = c.operator-();
  c = a – b; → c = a.operator-(b);

- **OJO**, al sobrecargar los operadores conservan su prioridad de evaluación y su asociatividad.

- No cambiar las operaciones a los operadores: + para sumar, etc.

- Si asignamos ^ la exponenciación no es buena idea, tiene una prioridad muy baja → puede dar resultados no esperados dentro de una expresión.

- Si sobrecargamos los operadores =, [], → y () deben ser métodos de una clase y no funciones externas.

- La sobrecarga es muy útil con clases que trabajan con tipos abstractos de datos.

# Ejemplo

- class Complejo {

```
private:
    double real, img;
public:
    Complejo(double r=0, double i=0) : real(r), img(i){}

    Complejo Complejo::operator+(Complejo x){
      return Complejo(real+x.real, img+x.img);
    }

 // Intentar definir:
Complejo operator+(Complejo x, Complejo y);
// Dentro de nuestra clase daría error, está implícito this.
```

# Sobrecarga con una función externa

- Esta definición:
  Complejo operator+(Complejo x, Complejo y);
  Como una función externa no daría error.

- **Problema**: Dentro de la función necesitamos acceder a los atributos de los objetos, pero estamos en una función externa.

- **Solución**: Definir la función externa como **friend** dentro de la clase complejo.

- Se puede utilizar: d = d + Complejo(3,3);

- **Restricción**: No podemos sobrecargar un operador binario para realizar un operador unario.

- **Restricción**: No podemos inventarnos nuevos operadores. **

# Operador de igualdad

- El operador de = ya está implementado en C++.
- Se puede implementar así:
  - CRacional se compone de numerador y denominador.

```
CRacional& CRacional::operator=(const CRacional &c){
    numerador = c.numerador;
    denominador = c.denominador;
}

// Devuelve el mismo objeto que recibe el mensaje.
// Otra veces nos puede interesar devolver un nuevo objeto: return CRacional ( … );
```

# Operadores Aritméticos

- Si queremos que nuestro operador devuelva un nuevo objeto, podemos definir algo así:

```
const CRacional CRacional::operator+(const CRacional &r){
        CRacional tmp( … operaciones …)
        return tmp;
    }
```

// Al devolver **const** podemos evitar: a+b = c;

# Aritmética mixta

- Dada una clase Racional definida por un numerador y denominador, puede interesarnos operaciones de este tipo:
- ¾ + 5; // Sería sumar la fracción con un entero.
- Lo podemos solucionar con parámetros por defecto en el constructor:
  - CRacional(long num=0, long den =1);
- El operador + sería:
  - const CRacional CRacional::operator+(const CRacional &r);
- El siguiente código no daría problemas:

```
CRacional r(3,4), c;  // La llamada con 5 se convierte en 5/1.
int n = 5;
c = r + n;        // Equivale a: c = r.operator+(n);
                  // El entero n se convierte a long y luego a CRacional.
```

No hay problema, ¿pero si sumamos: c = n + r ? → ERROR

# Aritmética mixta II

- Para hacer que la operación funcione como conmutativa debe recibir dos objetos CRacional. → implementarla con una **función externa** y definirla como **friend** dentro de la clase CRacional.

- class CRacional {

```
    friend const CRacional operator+(
    const CRacional &, const CRacional &);
}
```

# Sobrecarga de ==

- Si queremos que soporte la propiedad conmutativa tenemos que hacer lo mismo que antes.

- friend bool operator==(const CRacional &r1, const CRacional &r2);

# Sobrecarga de <<

- C++ incluye la clase ostream obtenida de basic_stream.
- Para sobrecargar el operador << tenemos dos soluciones:
  - cout.operator<<(r) En la librería no tenemos este método.
  - operator<<(cout,r) A través de una función externa.
  - La idea es poderlo utilizar de esta forma:
  - cout  << r1 << r2;
  - Se resuelve:
    - operator <<(cout.operator<<(r1)).operator<<(r2);

# Sobrecarga de <<

- Para poder encadenar las expresiones: cout << r1 << r2; // necesitamos que nuestra función devuelva un flujo.

- Dentro de la clase, será friend.

- ostream& operator <<(ostream &os, const CRacional &r){

```
    return os << r.numerador << "/" << r.denominador;
}

// Devolvemos el flujo, y al ser friend podemos acceder
   a los atributos.
```

# Sobrecarga de >>

- C++ incluye la clase istream a partir de la plantilla basic_stream.
- Al igual que antes se implementa con una función externa.
- class Cracional {

  friend std::istream& operator>>(std::istream &, Cracional &);

  };

// Se podría utilizar cin >> a >> b;

// Siendo a y b objetos de la clase CRacional.

# Sobrecarga de Unarios

- Sobrecargar el operador ! en la clase CRacional:

- Uso: if (!a){ …

- Implementación:

```
bool Cracional::operator!(){
    return !numerador;
}
```

# Incremento / Decremento

- Estos dos operadores se pueden utilizar de sufijo / prefijo.
- La operación ++a devuelve a incrementado.
- La operación a++ devuelve a y luego incrementa.
- CRacional CRacional::operator++(){

```
    numerador+=denominador;
    return *this;
}
```

# Incremento / Decremento II

- Para hacer la distinción de prefijo y sufijo añadió un parámetro.

- CRacional CRacional::operator++(int){

```
    CRacional temp = *this; // Guardar una copia.
    numerador += denominador;
    return temp;
}
```

# Operadores unarios / binarios

- Unario – cambia el signo.

```
CRacional Cracional::operator-(){
    CRacional temp(-numerador, denominador);
    return temp;
}
// OJO devuelve un nuevo objeto. No cambia el original.
```

- Binario – resta.

  Será una función externa definida como friend en la clase.

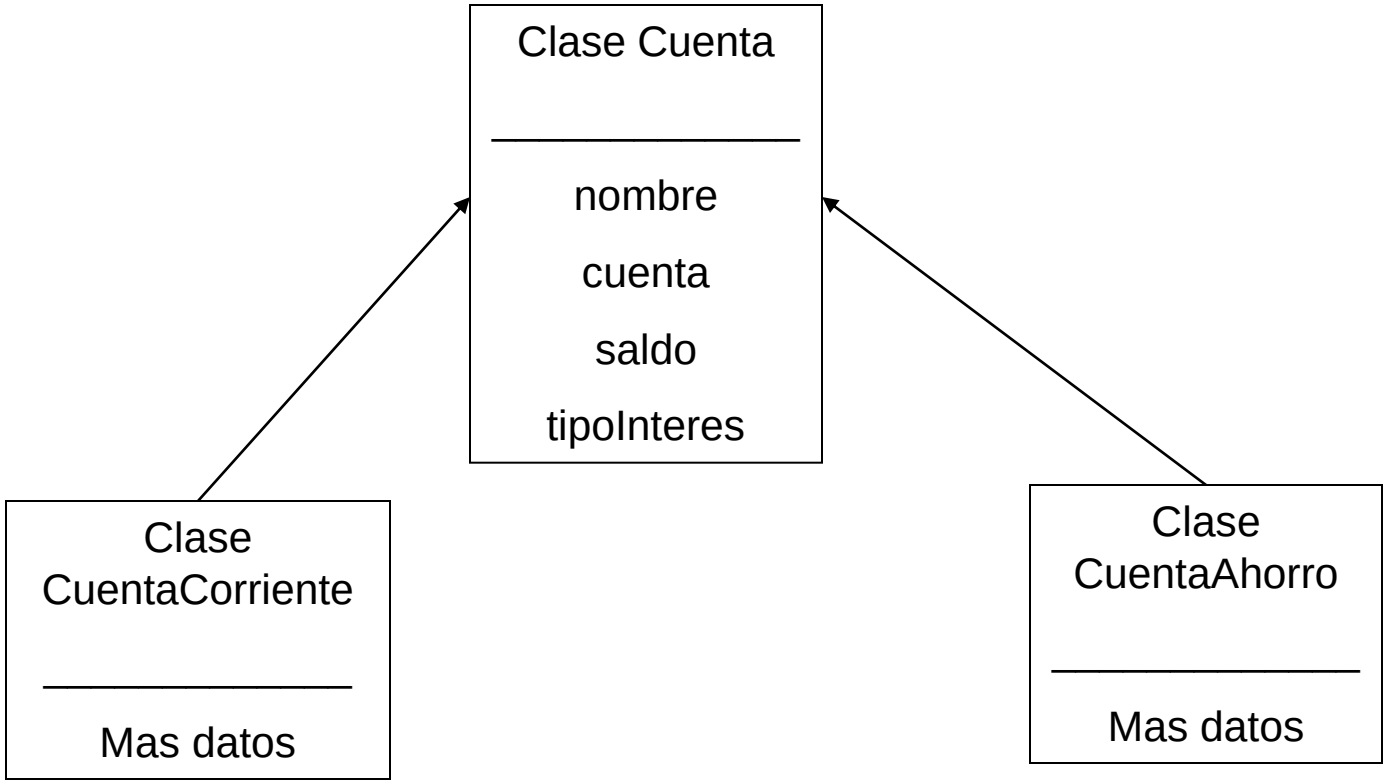
# Herencia

- La herencia es un mecanismo mediante el cual podemos construir clases a partir de otras.
- La nueva clase recibe los nombres de: subclase, clase derivada, clase hija.
- La clase existente recibe los nombres de: superclase, clase base y clase padre.

# Herencia Simple

- Con la herencia simple podemos crear jerarquías de clases.
- A la hora de diseñar una jerarquía podemos pensar en lo más genérico e ir bajando a lo mas particular.
- O ir centrándonos en los detalles e ir subiendo hacia los mas genérico.
- Agrupando cosas y comportamientos comunes de las clases hijas podemos obtener las clases mas generales.

# Ejemplo


Clase Cuenta
nombre
cuenta
saldo
tipoInteres
Clase CuentaCorriente
Mas datos
Clase CuentaAhorro
Mas datos

# Definir Clases Derivadas

- La herencia de la podemos ver como una especialización.

- Al definir la clase derivada tenemos que indicar el nombre de la clase base.

- class CuentaAhorro **:** public **Cuenta** {

  // Lista de métodos y atributos …

  }

# Control de acceso a la clase base

- También podemos indicar que tipo de acceso queremos tener a la clase base.
- Tipos de acceso: private, protected y public.
- Si indicamos public (es el mas habitual, como en el ej anterior). Todos los métodos y atributos de la clase base mantiene su tipo de acceso, es decir:
  - Lo que era privado se mantiene privado.
  - Lo protegido idem
  - y en lo público idem.

# Control de acceso a la clase base

- class Cuenta {} // Mi clase base.
- class CuentaCorriente : private  Cuenta {};
  - El tratamiento que le damos a la clase Base es privado. Todo lo público y protected de la clase pasa a ser private. Sólo se puede acceder a las funciones **amigas**.
- class CuentaCorriente : protected Cuenta {}
  - Los miembros public y protected pasan a ser protected y los privados se mantienen privados.
- Class CuentaCorriente : public Cuenta {}
  - Este caso mantiene todo como estaba.
- Si no podemos nada por defecto es **private**.

# Control de acceso a la clase base

- Ojo, se puede dar esta situación:
  - class A { … }
  - **class B : A** { … }
  - En este caso la herencia es **private**.

- A parte de la limitación de los métodos que no será visibles desde un objeto de la clase B.

- Tampoco podremos hacer esto:
  - ***A *a = new B(…)***
  - ***Dará un error porque la clase A (es inaccesible)***

# Control de acceso

- Las funciones externas solo pueden acceder a los métodos públicos.
- Desde la misma clase y desde funciones amigas puedo acceder a los 3 niveles.
- Desde una clase hija solo accedo a lo público y lo protegido.
- Desde cualquier otra clase y que no sea amiga solo accede a lo público.

# Miembros a heredar

- La clase derivada hereda todos los miembros de la clase base.
- **OJO los constructores NO.**
- Una clase derivada no accede a lo privado de la clase base.
- La clase derivada añade sus propios métodos y atributos.
- Incluso los puede llamar igual que los de la clase Base, pero quedarían ocultos los miembros de la clase Base que se llamaran igual.
- **ESTO, EN POO SE CONOCE CON EL NOMBRE DE REDEFINICIÓN O SOBREESCRIBIR EL MIEMBRO (MÉTODO o ATRIBUTO) DE LA CLASE BASE.**
- Si volvemos a heredar se sigue propagando por la jerarquía de clases → **propagación de la herencia.**

# Redefinir Atributos

- Cuando definimos un atributos en la clase hija que se llama igual que el de la padre, este último se **sobreescribe**.

- Si desde la clase hija hacemos referencia al atributo, primero se comprueba si está definido en la clase hija y si no se localizará el de la clase padre.

- Siempre tiene preferencia el atributo de la clase hija.

# Ejemplo

```
class ClaseA {
  protected:
    int atributo_x;

  public:
    ClaseA(int x = 1) : atributo_x(x) {}

    int metodo_x()  {
      return atributo_x * 10;
    }

    int metodo_y()    {
      return atributo_x + 100;
    }
};
```

```
class ClaseB : public ClaseA {
  private:
    int atributo_x;
  public:
    ClaseB(int x = 2) : atributo_x(x) {}

    int metodo_x()    {
      return atributo_x * -10;
    }

    int metodo_z()   {
      atributo_x = ClaseA::atributo_x + 3;
      return ClaseA::metodo_x() +
      atributo_x;
    }
};
```

```
int main()
{
  ClaseB objClaseB;
  cout << objClaseB.metodo_x() << endl;
  cout << objClaseB.metodo_y() << endl;
  cout << objClaseB.metodo_z() << endl;
}
```

¿Qué salida tiene?

¿Cómo se ejecutan los métodos?

# Acceso a la clase Base

- Aunque hayamos sobrescrito un método o atributo de la clase base tenemos una forma de poder acceder al atributo / método de la clase padre en vez ala clase hija.

- // En la clase Hija: Accedemos directamente …

```
int metodo_x(){
        return ClaseA::atributo_x * -10;
}
```

# Acceso a la clase Base

- Desde la clase hija:

  ClaseA::atributo_x; Accede a la clase Base.

  this→atributo_x; Accede al atributo de la clase hija, estoy referenciando mediante this.

- ¿Cómo accedo mediante this a la clase Base?

  static_cast<**ClaseA** ***>(this)→atributo_x;

  Hacemos una conversión del puntero.

# Redefinición de métodos

- Diferenciar bien entre redefinición y sobrecarga (No es lo mismo).
- Para sobreescribir un método lo tenemos que volver a definir en la clase hija.
- Al sobreescribirlo se oculta el método de la clase base y todas las sobrecargas que existan en la clase Base.

# Control de Acceso en Redefinición

- Podemos cambiar el tipo de acceso al redefinir un método en la clase hija.
- Le podemos hacer público o hacerle mas restrictivo.
- Para llamar a un método de la clase base desde la hija. **Clase_Base::metodo()**;
- Para referenciar al método de la clase hija desde la propia clase hija: con this.

  **this→metodo();**

# Constructores en las clases derivadas

- Cada vez se crea un objeto se invoca al constructor.
- Los constructores no se heredan
- Por la relación de herencia primero se invocan los constructores de la clase Base y después el de la clase Derivada.
- Si tenemos mas de dos clases en la jerarquía, la construcción del objeto de la clase hoja (de la última de la jerarquía), provocará una serie de llamadas por la jerarquía hasta alcanzar el clase de mas alto nivel y retornará el flujo de llamadas hasta el constructor de la clase hoja.

# Constructores en las clases derivadas

- Con los constructores por defecto este proceso se realiza de forma automática.

- Si hemos implementado un constructor con parámetros en la clase hija a partir de este tendremos que llamar al constructor de la clase base.

- El constructor de la clase derivada tendrá tantos parámetros como atributos tienen entre la clase derivada y los heredados de la clase base.

# Sintaxis

Nombre_clase_derivada(parámetros) : lista de iniciadores {

// Cuerpo del constructor de la clase derivada.

}

Proceso de ejecución:

- 1) Saltan los constructores de las clases Base.
- 2) Se construyen los atributos de la clase derivada.
- 3) Se ejecuta el cuerpo del constructor de la clase Derivada.

# Ejemplo

- Sabiendo que cuenta CuentaAhorro (1 atributo, cuotaManteniemiento) hereda de Cuenta (4 atributos).

```
CuentaAhorro::CuentaAhorro(string nom, string cue, double saldo, double tipo,
    double mant) :
Cuenta(nom, cue, saldo, tipo), cuotaMantenimiento(mant){
      // Cuerpo del constructor…
      }

      main() { // podemos definir…
          CuentaAhorro c1; // saltarán los constructores por defecto.
          CuentaAhorro c2("cliente2", "10101102", 2000, 1.75, 10);
      }
```

# Copia de Objetos

- La copia de objetos se lleva a cabo mediante el operador = y el constructor copia.

```
main(){
    CuentaAhorro c1;
    CuentaAhorro c2(“cliente2”,”0101292”,2000,
    1.75, 10);

    c1 = c2; // Salta el operador de asignación.
    CuentaAhorro c3 = c2; // Constructor por copia.
```

# Implementar constructor copia

- Si necesitáramos explicítamente implementar el constructor por copia en la clase CuentaAhorro:

```
CuentaAhorro::CuentaAhorro(const
   CuentaAhorro& ca) : Cuenta(ca){
   cuotaMantenimiento = ca.cuotaMantenimiento;
}
```

# Implementar operador de Asignación

```
CuentaAhorro& CuentaAhorro::operator=(const CuentaAhorro& ca){
    Cuenta::operator=(ca);
    cuotaMantenimiento = ca.cuotaMantenimiento;
    return *this;
}
```

- Cuando trabajamos con la herencia tenemos que tener en cuenta que cada objeto tiene su responsabilidad, de tal forma que todo lo que gestione la clase Base debe realizarlo ella.

- Con herencia siempre nos apoyamos en lo que ya está hecho o en funcionalidades que me ofrecen, si no, la herencia no tendría sentido.

# Destructores

- El destructor de la clase base NO es heredado por sus clases derivadas.
- El orden en que se van a destruir los objetos es inverso a como se han creado.
- En el caso de la clase CuentaAhorro primero se ejecutará el destructor de esta, después los atributos de esta y por último saltará el destructor de la clase Cuenta.
- El destructor deberá aparecer en ambas clases.

```
~CuentaAhorro(){ }
~Cuenta(){ }
```

PRACTICA SOBRE HERENCIA

# Funciones Amigas

- Tener en cuenta que si tenemos una clase Base con una función amiga (friend) y una clase Derivada que hereda de esta.
- Desde la función friend de la clase Base NO vamos a poder acceder a los miembros protegidos y privado de la clase Derivada.
- **No se hereda la amistad.**
- Y si tenemos una función friend en la clase Derivada tampoco va a poder acceder a los miembros privado de su clase Base.

# Punteros y Referencias

- En cuanto a punteros y referencias con las clases derivadas funcionan exactamente igual que con otras clases.

- Partiendo de una ClaseBase y una ClaseDerivada que hereda de esta.

```
ClaseDerivada obj (param1, param2);
ClaseDerivada *p = &obj; // Extraer dirección.
ClaseDerivada &r = obj; // Extraer  referencia.
```

# Conversiones Implícitas

```
class ClaseBase  {
private:
      int base;
public:
      ClaseBase();
      void metodoBase();
      virtual ~ClaseBase();
};

class ClaseDerivada : public
      ClaseBase {
private:
      int derivada;

public:
      ClaseDerivada();
      void metodoDerivada();
      virtual ~ClaseDerivada();
};
```

- En main()

- ¿Qué es correcto?

```
void main(){

      ClaseDerivada cd1 = ClaseDerivada(); // OK
      ClaseBase cb1 = ClaseBase(); // OK

      ClaseDerivada cd = ClaseBase();  //ERROR
      ClaseBase cb = ClaseDerivada(); // OK

}
```

Herencia: Un perro es un mamífero
El perro hereda de Mamífero.

# Conversiones Explícitas

- Forzar al revés:

```
ClaseDerivada *cd1 = new ClaseDerivada();
ClaseBase *cb1 = new ClaseBase();
cd1 = cb1; // ERROR
cd1 = static_cast<ClaseDerivada *>(cb1);
```

# Métodos Virtuales

- ¿Para que sirven los métodos virtuales?
- Problema: Cuando tenemos una relación de herencia Base ← Derivada, y la clase Derivada sobrescribe métodos de la clase Base y sabemos que podemos referenciar un objeto de la clase Derivada mediante un objeto de la clase Base.
- Cuando llamemos a un método que existe en ambas clases va a saltar el de la clase Base.
- Para que se reconozca el de la clase Derivada tenemos que utilizar **virtual** es la forma que se soluciona en C++

# Ejemplo

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

class Animal {
        public:
        virtual void come() {
             cout << "Yo como como un animal
             genérico.\n";
        }
};

class Lobo : public Animal {
        public:
        void come() {
        cout << "¡Yo como como un lobo!\n";
        }
};

class Pez : public Animal {
        public:
        void come() {
        cout << "¡Yo como como un pez!\n";
        }
};

class OtroAnimal : public Animal { };
```

```
int main() {
      Animal *unAnimal[4];
      unAnimal[0] = new Animal();
      unAnimal[1] = new Lobo();
      unAnimal[2] = new Pez();
      unAnimal[3] = new OtroAnimal();

      for(int i = 0; i < 4; i++) {
              unAnimal[i]->come();
      }

      for (int i = 0; i < 4; i++) {
              delete unAnimal[i];
      } return 0;
}
```

**Salida con el método virtual come:**
Yo como como un animal genérico.
¡Yo como como un lobo!
¡Yo como como un pez!
Yo como como un animal genérico.

**Salida sin el método virtual come:**
Yo como como un animal genérico.
Yo como como un animal genérico.
Yo como como un animal genérico.
Yo como como un animal genérico.

# Ejemplo 1ª parte

```
class CBase  {

public:
    CBase();
    virtual void mVirtual();
    void mNoVirtual();
    virtual ~CBase();

};
```

**VER EJEMPLO CODIGO**

Trabajar con **métodos virtuales** implica trabajar con **punteros o referencias**

**(para las ref. métodos const)**

```
class CDerivada1 :
    public CBase {

public:
    CDerivada1();
    // Sobreescribir métodos
    void mVirtual();
    void mNoVirtual();
    virtual ~CDerivada1();
};

// En la clase Derivada no
    pongo virtual.
// Sería redundante.
```

# Ejemplo 2ª parte

```
#include "Base.h"
#include "Derivada1.h"
#include <iostream>
using namespace std;

int main(){

    // 1ª Prueba con Punteros:
    cout << "Prueba con punteros" << endl;
    CBase *base = new CDerivada1();
    base->mNoVirtual();
    base->mVirtual();
    delete base;


    // 2ª Prueba objetos:
    cout << endl << "Prueba con objetos"  << endl;
    CBase base2 = CDerivada1();
    base2.mNoVirtual();
    base2.mVirtual();


    // 3ª Prueba con Referencias:
    cout << endl << "Prueba con Referencias" <<
    endl;
    CBase& base3 = CDerivada1();
    base3.mNoVirtual();
    base3.mVirtual();
}
```

```
Prueba con punteros
metodo no Virtual de la Base
Metodo virtual de la Derivada

Prueba con objetos
metodo no Virtual de la Base
metodo virtual de la Base

Prueba con Referencias
metodo no Virtual de la Base
Metodo virtual de la Derivada
Press any key to continue_
```

**Objetivo**: Aunque esté referenciando un objeto de la clase Derivada a través de uno de la clase Base,

**Quiero que salte el método de la clase Derivada**

Usar un método **virtual.**

# Resumen Métodos Virtuales

- Un método virtual es un miembro de una clase base que puede ser redefinido en cada una de las clases derivadas, y cuando se redefine puede ser accedido mediante un puntero o una referencia.
- El método virtual se define en el .h indicando virtual delante del método. En el cpp nada.
- Y si la clase derivada sobrescribe el método virtual tampoco tiene que indicarlo, pero debe coincidir en tipo de los parámetros, tipo devuelto y número.
- La clase derivada a su vez puede tener sus métodos virtuales.
- Una clase con métodos virtuales recibe el nombre de **TIPO POLIMÓRFICO.**

# Destructores Virtuales

- Es mejor de definir Destructores Virtuales, la razón es que si tenemos un objeto de la Clase Derivada referenciado por un objeto de la clase Base y salimos del ámbito del objeto, saltará el destructor de la clase Base.

- Esto se soluciona con un Destructor Virtual, por eso C++ lo pone por defecto.

# typeid

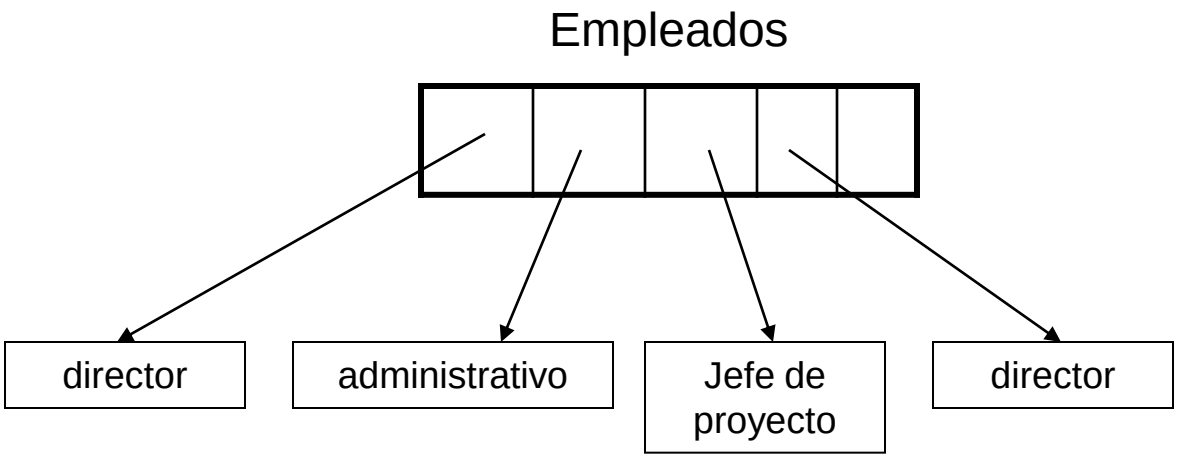
- Con este operador podemos extraer el nombre de la clase a la que pertenece un objeto.

- El operador nos devuelve una referencia constante a type_info:

```
ClaseDerivada *cd1 = new ClaseDerivada();
const type_info& info = typeid(cd1);
cout << info.name() << endl;
```

- Necesario: **#include <typeinfo>**

# Polimorfismo




En C++ un comportamiento polimórfico implica métodos virtuales y trabajar un punteros o referencias

PRACTICA: POLIMORFISMO

# Importante

- Destruir siempre los objetos en el mismo ámbito que se crean.

- Si no puede dar lugar a intentar dos veces liberar el mismo objeto.

# Clases Abstractas

- Hay veces que en una relación de Herencia tenemos una clase Base que no tiene sentido crear objetos de ella porque es genérica, pero de las subclases si.

- Por ejemplo:
  - La figura y de esta pueden heredar:
    - Círculo
    - Cuadrado
    - Rectángulo

# Clases Abstractas

- Para indicar en C++ que se trata de una clase abstracta, tenemos que declarar un método **virtual puro**.

- No hace falta que tenga cuerpo, lo deberían sobrescribir las clases hijas.

- Sintaxis:

  virtual tipo nombre_metodo([parametros]) = 0;

# Clases Abstractas

- Si la clase hija no sobrescribe el método virtual puro se convierte en abstracta.

- Una clase abstracta no se puede instanciar pero si podemos definir punteros y referencias a dicha clase.

  Figura f; → error

  Figura *f ; → ok

  Figura& f; → ok

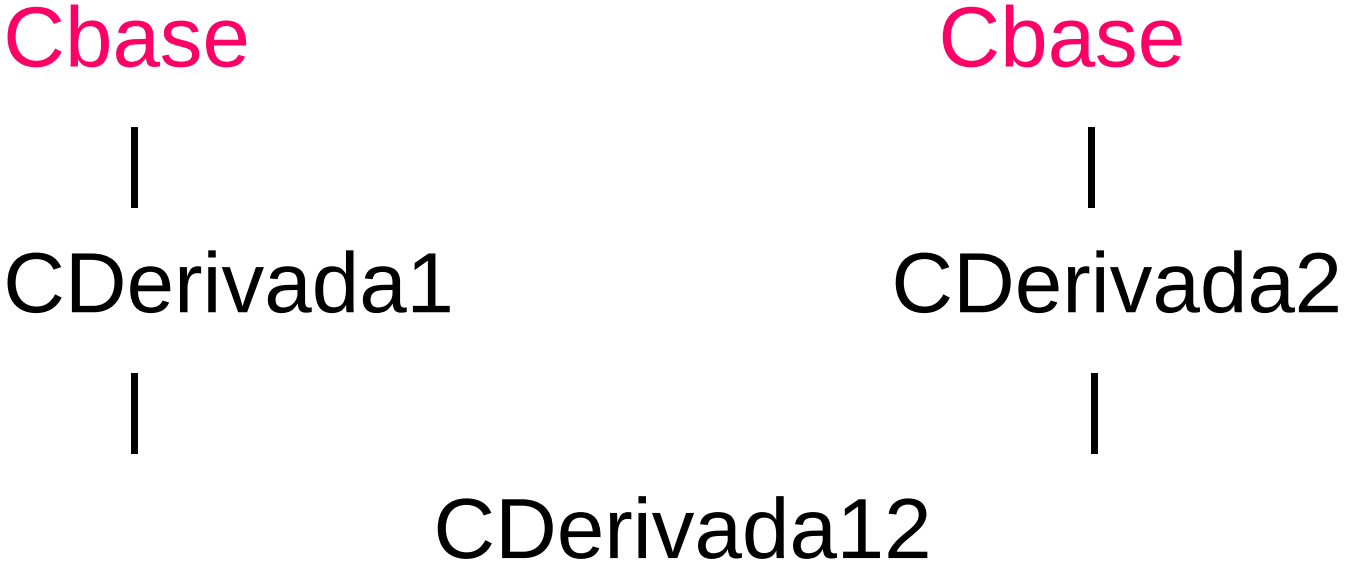
# Herencia Múltiple

- Cuando una clase derivada tiene varias clases base estamos en un caso de Herencia Múltiple.
- Sintaxis:

  class Derivada: public Base1, public Base2 { }
- La herencia múltiple permite que clases hermanas compartan información.
- El acceso a los miembros se hace igual que en herencia simple.

# Problemas

- Que las clases Base sean la misma:




Puede dar lugar a ambiguedades por la herencia, cuando desde las clases derivadas accedamos a la clase Base.

**SOLUCIÓN: CLASES BASE VIRTUALES**

# Clases Base Virtuales

- Sintaxis:
- class Cbase { };

  class CDerivada1 : public virtual Cbase {}

  class CDerivada2 : public virtual Cbase {}

  Class CDerivada12 :

  public CDerivada1, public CDerivada2 {}