Tema 2: Herencia y Polimorfismo en C++

Diseño modular: .hpp y .cpp

A la hora de programar es recomendable seguir un diseño modular, creando un fichero .hpp para cada declaración de clase y un fichero .cpp para su correspondiente definición.

En un módulo se definirá una única clase o bien clases muy relacionadas. No se debe intentar minimizar el número de archivos fuente sino maximizar la organización y legibilidad de las fuentes.

En el .hpp se debe definir una constante con el preprocesador que nos dirá si el archivo ha sido ya incluido con objeto de no volver a incluir las definiciones, ya que darían un error de compilación. Se suele elegir una constante con el mismo nombre del archivo de cabecera o de la clase que se define.

Normalmente, diferentes clases relacionadas se empaquetan en bibliotecas. Las bibliotecas se crean mediante un proyecto de librería estática en el que se incluyen todos los módulos que queremos empaquetar en la biblioteca. La compilación genera un archivo de biblioteca (.a) en lugar de un ejecutable (.exe).

En nuestros programas podremos enlazar archivos de código objeto (.o) y bibliotecas (.a)

Punto
int x # int y
+ Punto(int x=0, int y=0) + void ver( )

Div
- int n - int d
+Div(int a=0, int b=1) + void ver( )

Salida:

(0,0)
(7,9)
0/1
3/1
hola

¡Ojo!
Algunas palabras reservadas (static y virtual) se escriben en el .hpp pero no en el .cpp

Los valores por defecto en los argumentos se indican en el .hpp pero no en el .cpp

El operador de resolución de ámbito ::

Punto.hpp

```
#ifndef PUNTO_HPP
#define PUNTO_HPP

using namespace std;

class Punto {
protected:
 int x, y;
public:
 Punto(int x=0, int y=0);
 void ver() const;
};

#endif
```

Div.hpp

```
#ifndef DIV_HPP
#define DIV_HPP

using namespace std;

class Div {
 int n;
 int d;
public:
 Div(int a=0, int b=1);
 void ver() const;
};

#endif
```

Punto.cpp

```
#include <iostream>
#include "Punto.hpp"

using namespace std;

Punto::Punto(int x, int y) {
 this->x = x; this->y = y;
}

void Punto::ver() const {
 cout << "(" << x << ","
      << y << ")\n";
}
```

Div.cpp

```
#include <iostream>
#include "Div.hpp"

using namespace std;

Div::Div(int a, int b):n(a) {
 d = b;
}

void Div::ver() const {
 cout << n << "/"
      << d << endl;
}
```

main.cpp

```
#include <iostream>
#include "Punto.hpp"
#include "Div.hpp"

using namespace std;
void ver() { cout << "hola\n"; }

int main(int argc, char *argv[]) {
 Punto p, q(7,9);
 Div f, h(3);
 p.ver(); q.ver(); f.ver(); h.ver(); ver();
 system("PAUSE"); return EXIT_SUCCESS;
}
```

Sirve para indicar a qué clase corresponde cada método.
Esto permite definir métodos con el mismo nombre para clases distintas y/o para saber si una determinada función es miembro de una clase o es independiente de una clase

La referencia o puntero implícito this (para que sirve)

Cada vez que se crea un objeto el compilador crea un puntero implícito **this** que apunta al propio objeto. Se utiliza para resolver:

-Problemas de ambigüedad por enmascaramiento con locales o argumentos
-Estructuras enlazadas
-Devolver el propio objeto (*this)

```
class Punto{
private:
        int x;
        int y;
public:
        void cambia( int x, int y ) {
          this->x = x;
          this->y = y;
        }
        Punto clonar( ) {
          return (*this);
        }
        Punto operator++(  ) { //++obj
          x += 1;
          return (*this);
        }
        Punto operator++(int i ) { //obj++
          Punto p;
          p.cambia(x, y);
          x += 1;
          return p;
        }
```

```
class NodoArbolBinario{
protected:
…
        NodoArbolBinario *padre;
        NodoArbolBinario *hijoDerecha;
        NodoArbolBinario *hijoIzquierda;
…
public:
        void insertaHijoDerecha( NodoArbolBinario *hd ){
                hijoDerecha = hd;
                hd -> padre = this;
        }
…
};
```

Objetos miembros de otros objetos: construcción anidada
Una clase puede contener miembros que son objetos de otras clases.
El constructor debe llamar a los constructores de los objetos miembros mediante inicializadores.
Si no se hace, el constructor implícitamente invoca los constructores por defecto de los objetos anidados.
(En cuanto al destructor, este invoca automáticamente los destructores de los objetos miembros)
Sintaxis:

```
Clase_Continente::Clase_Continente(argumentos):
  Objeto_OtraClase1(argumentos), Objeto_OtraClase2(argumentos), otras_variables {
   Asignación_de_otras_variables;
}
```

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

class Punto {
 int x, y;
public:
 Punto() { x = 0; y = 0; }
 Punto(int a, int b) { x = a; y = b; }
 void set(int a, int b) { x = a; y = b; }
 friend ostream& operator<<(ostream &s, const Punto &p) { //const no por ser friend
  s << "(" << p.x << "," << p.y << ")";
  return s;
 }
 static float distancia(const Punto &a, const Punto &b) { //const no por ser static
  return sqrt(pow((b.x-a.x),2.0)+pow((b.y-a.y),2.0));
 }
};

class linea {
 Punto x, y; //aunque la clase Punto no tuviera constructor por defecto se puede
public:      //declarar objetos Punto sin parametros
 linea(int a, int b, int c, int d);
 operator float() const { return Punto::distancia(y,x); }
 void ver() const { cout << x << "," << y << endl;}
};

linea::linea(int a, int b, int c, int d) {
 x.set(a,b);
 y.set(c,d);
}

int main(int argc, char *argv[]) {
  linea a(1,1,4,5);
  a.ver();
  cout << "Distancia: " << (float)a << endl;
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

(1,1),(4,5)
Distancia: 5

```
linea::linea(int a, int b, int c, int d)
 : y(), x() {
 x.set(a,b);
 y.set(c,d);
}
ESTE ES EQUIVALENTE AL QUE ESTA EN ROJO
```

El constructor linea llama al constructor por defecto de los puntos p1 y p2 que tomarán ambos el valor (0,0). **Si la clase punto no tuviera definido un constructor por defecto, el constructor línea debería de llamar explícitamente a los constructores de los objetos p1 y p2 con inicializadores**.

```
class linea {
 Punto x, y;
public:
 linea(int a, int b, int c, int d): y(c,d), x(a,b) { //cuerpo vacio }
};
```

```
linea(int a, int b, int c, int d): y(c,d), x(0,0) {
  x.set(a,b);
} ES EQUIVALENTE PERO MENOS EFICIENTE YA QUE CAMBIO x 2 VECES…
```

El orden de creación de los objetos no viene determinado por el orden en el que son llamados en los inicializadores, sino por el orden en que son declarados los atributos dentro de la clase.
En el ejemplo anterior se inicializa antes x que y, aunque el orden esté al revés en los inicializadores.

Arrays de objetos miembros de otros objetos: construcción anidada

Una clase puede contener miembros que son arrays de objetos de otras clases.

Para poder crear arrays de objetos de una determinada clase x, la clase x debe tener un constructor que puede invocarse sin argumentos (ya que no es posible pasar argumentos propios para cada uno de los objetos del array para su inicialización).

El constructor de la clase contenedora de arrays de objetos debe llamar a los constructores de los objetos miembros mediante inicializadores.

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

class Punto {
 int x, y;
public:
 Punto() { x = 0; y = 0; }
 Punto(int a, int b) { x = a; y = b; }
 void set(int a, int b) { x = a; y = b; }
 friend ostream& operator<<(ostream &s, const Punto &p) { //const no por ser friend
  s << "(" << p.x << "," << p.y << ")";
  return s;
 }
 static float distancia(const Punto &a, const Punto &b) { //const no por ser static
  return sqrt(pow((b.x-a.x),2.0)+pow((b.y-a.y),2.0));
 }
};

class triangulo {
 Punto p[3]; //coordenada de los 3 puntos del pentagono
public:
 triangulo(int a, int b, int c, int d, int e, int f)/*:no inicializadores*/ {
   p[0].set(a,b); p[1].set(c,d); p[2].set(e,f);
 }
 float perimetro() const {
   float peri=0;
   for(int i=0; i<3; i++)
     peri+=Punto::distancia(p[i],p[(i+1)%3]);
   return peri;
 }
 void ver() const { cout << p[0] << "," << p[1] << "," << p[2] << endl; }
};

class puntoLinea {
 Punto p;
 Punto l[2]; //coordenada de los 2 extremos de la linea
public:
 puntoLinea(int a, int b, int c, int d, int e, int f):p(a,b) {
   l[0].set(c,d); l[1].set(e,f);
 }
 void ver() const { cout << "Punto: " << p << " Linea: " << l[0] << "-" << l[1] << endl; }
};

int main(int argc, char *argv[]) {
  triangulo a(1,1,4,1,4,5);
  puntoLinea b(2,6,4,1,4,7);
  a.ver();
  cout << "Perimetro: " << a.perimetro() << endl;
  b.ver();
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

(1,1),(4,1),(4,5)
Perimetro: 12
Punto: (2,6) Linea: (4,1)-(4,7)

Herencia simple: Proceso mediante el cual una clase (**clase derivada**) se crea a partir de otra clase (**clase base**). Una clase derivada puede a su vez ser la clase base de otra clase que deriva de ella y así sucesivamente dando lugar a una jerarquía de clases.

Los miembros de una clase pueden ser:

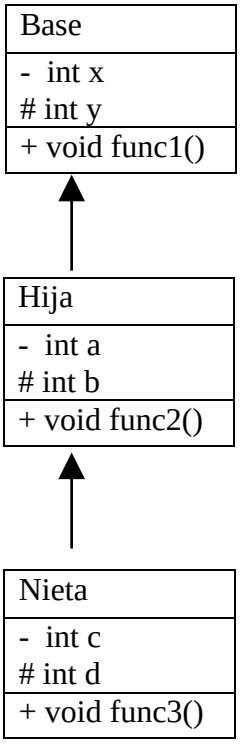
- Privados (**private**): solo son visibles y pueden ser usados por los métodos de la clase. No son accesibles desde el exterior de la clase (ninguna clase derivada, función no miembro, ni resto de clases puede acceder a ellas). Están encapsulados y su uso es a través de la interfaz de la clase
- Protegidos (**protected**): son visibles y pueden ser usados por los métodos de la clase y por las clases derivadas.
- Públicos (**public**): son visibles y pueden ser usados en cualquier ámbito.

La clase derivada tiene (heredado) todos los atributos y funciones miembros (métodos) de la clase base (excepto algunos); además puede agregar atributos y/o métodos adicionales. Que lo herede no significa que pueda acceder directamente a ellos (eso depende de si lo que hereda es privado, protegido o publico). La clase derivada no tiene acceso directo a los **private** de la clase base, pero si a los **protected** y **public**.

La forma en la que los miembros de una clase Base se heredan en la clase Derivada depende si la herencia es **public**, **protected** o **private** (por defecto). Si la clase Base se hereda como:

- Pública (**public**): no se modifica el nivel de acceso indicado en la clase Base.
- Protegida (**protected**): los miembros **public** y **protected** de la clase Base pasan a ser **protected** en la clase Derivada.
- Privados (**private**): los miembros **public** y **protected** de la clase Base pasan a ser **private** en la clase Derivada

Nota: Hemos subrayado y marcado en rojo las sentencias que son erróneas por ser inaccesibles




```
class Base {
 private:    int x;
 protected: int y;
 public:
  void func1() { x=1; y=2;}
};

class Hija: public Base {
  int a; //private
 protected:  int b;
 public:
  void func2() { x=1; y=1;
                 a=2;b=3;
                 func1(); }
};

class Nieta: public Hija {
   int c; //private
 protected:  int d;
 public:
  void func3() { x=1; y=1;
                 a=2; b=3;
                 c=1; d=1;
                 func1();func2();}
};

int main(int argc, char *argv[]) {
  Base b;    Hija h;    Nieta n;
  b.func1();
  h.func1();  h.func2();
  n.func1();  n.func2(); n.func3();
}
```

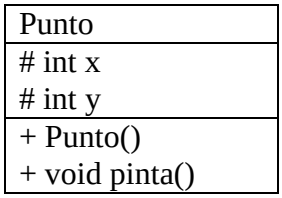
```
class Base {
 private:    int x;
 protected: int y;
 public:
  void func1() { x=1; y=2;}
};

class Hija: protected Base {
  int a; //private
 protected:  int b;
 public:
  void func2() { x=1; y=1;
                 a=2;b=3;
                 func1(); }
};

class Nieta: protected Hija {
   int c; //private
 protected:  int d;
 public:
  void func3() { x=1; y=1;
                 a=2; b=3;
                 c=1; d=1;
                 func1();func2();}
};

int main(int argc, char *argv[]) {
  Base b;    Hija h;    Nieta n;
  b.func1();
  h.func1();  h.func2();
  n.func1();  n.func2(); n.func3();
}
```

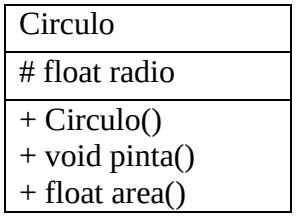
```
class Base {
 private:    int x;
 protected: int y;
 public:
  void func1() { x=1; y=2;}
};

class Hija: Base { //:private Base
  int a; //private
 protected:  int b;
 public:
  void func2() { x=1; y=1;
                 a=2;b=3;
                 func1(); }
};

class Nieta: private Hija {
   int c; //private
 protected:  int d;
 public:
  void func3() { x=1; y=1;
                 a=2; b=3;
                 c=1; d=1;
                 func1();func2();}
};

int main(int argc, char*argv[]) {
  Base b;    Hija h;    Nieta n;
  b.func1();
  h.func1();  h.func2();
  n.func1(); n.func2(); n.func3();
}
```

Acceso a la superclase.

Para acceder a las superclases, utilizaremos el **operador de resolución de ámbito ::**

Supongamos que en el método pintar, lo que hace *Circulo* es utilizar el método de pintar de su superclase Punto y luego escribir el área:





Pantalla:

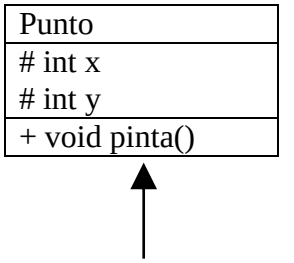
```
0,0 Area: 6.28
```

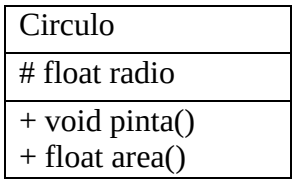
```
class Punto {
protected:
        int x;
        int y;
public:
   Punto() {  x = 0;  y = 0;  }
        void pinta(){
                cout << x << "," <<y;
   }
};

class Circulo: public Punto {
protected:
        float radio;
public:
   Circulo() {  radio = 1;   }
        void pinta() {
                Punto::pinta();
                cout << " Area: " << area();
        }
        float area(){ return 2*3.14*radio;   }
};

int main(int argc, char *argv[]) {
   Circulo c;
   c.pinta();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

- Sintaxis declaración de una clase:

```
class nombre[: <lista_de_clases_de_las_que_hereda>] {

[<definiciones de atributos y métodos miembro> (por defecto: private)
| public:
     <definiciones de atributos y métodos miembro>
| private:
     <definiciones de atributos y métodos miembro>
| protected:
     <definiciones de atributos y métodos miembro>]*
};

<lista_de_clases_de_las_que_hereda>::
[virtual] <privilegio_acceso> <nombre clase>
[, [virtual] <privilegio_acceso> <nombre clase> ]*

<privilegio_acceso>:: private: | protected: | public:
```

Privilegios en la herencia: **public, private, protected**
La clase heredada debe verse como un atributo de clase con el privilegio dado.
De hecho, puede accederse (con restricciones) como un atributo con el . ó ->:





Pantalla:

```
Circulo
Punto
```

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

class Punto{
protected:
        int x;
        int y;
public:
        void pinta( ){
                cout << “Punto\n”;
        }
};

class Circulo: public Punto{
protected:
        float radio;
public:
        void pinta( ){
                cout << “Circulo\n”;
        }
        float area( ){
                return 2*3.14*radio;
        }
};

int main(int argc, char *argv[])
{
   Circulo c;
   c.pinta( );
   c.Punto::pinta( );
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Qué miembros hereda una clase derivada

La clase derivada hereda todos los atributos y todos los métodos de la clase base excepto:

- Los constructores (el de copia tampoco) y los destructores
- Las funciones amigas (friend)
- El operador de asignación sobrecargado (si lo tuviera la clase base)

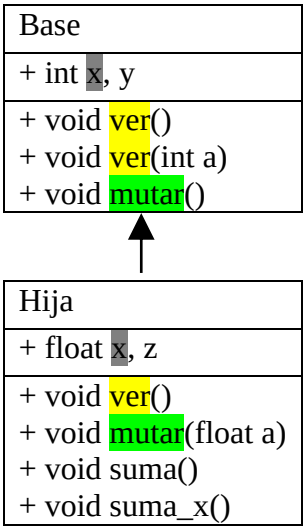
En el caso del operador de asignación y el constructor de copia, si una clase derivada no la redefine C++ genera automáticamente un operador de asignación y un constructor de copia de oficio que se limita a invocar al operador de asignación y constructor de copia de la clase base y hacer una copia binaria de los atributos exclusivos de la derivada.

Una clase derivada puede añadir sus propios atributos o métodos. Si el nombre de alguno de estos miembros coincide con el de un miembro heredado, el heredado queda oculto. En la clase derivada solo hay que listar los nuevos atributos y/o métodos propios de la clase (no heredados) y aquellos métodos heredados que queramos redefinir (para modificar su comportamiento original) o sobrecargar, así como los constructores, destructores y resto de métodos que no se heredan (si son necesarios).

- Para redefinir un método en la clase derivada debemos listarlo con la misma firma (prototipo) que tiene en la clase base.
- Para sobrecargar un método heredado en la clase derivada se lista con una firma distinta a la que tiene en la clase base.
- **Al redefinir o sobrecargar** un método en la clase derivada **ocultamos el original de la clase base y todas las sobrecargas que tuviera**. No obstante podemos acceder a los métodos de la clase base o abuela ocultos asi: **clase_base::**método_redefinido( ) o **clase_abuela::**método_redefinido( )

Pantalla:

```
Hija 1,2 + Base 3,4
Base 3,4
Hola
Hija 1,2 + Base 1,4
Hija 7,2 + Base 0,4
Base 0,4
9
7
```




```
class Base {
public:
  int x, y;
  void ver() { cout << "Base " << x << "," << y << endl; }
  void ver(int a) { cout << "Hola\n"; }
  void mutar() { x=0; }
};

class Hija: public Base {
public:
   float x, z;
   void ver() { cout << "Hija " << x << "," << z << " + ";
                Base::ver(); }
   void mutar(float a) { x=a; Base::mutar(); }
   float suma() {return (x + z); }
   float suma_x() {return (x + Base::x); }
};

int main(int argc, char *argv[]) {
  Hija h;
  h.x=1; h.z=2;
  h.Base::x=3; h.y=4;
  h.ver();   h.Base::ver();
  h.Base::ver(3);   //h.ver(3); esta oculto
  h.Base::mutar(); //h.mutar(); esta oculto
  h.Base::x++; h.ver();
  h.mutar(7);
  h.ver();   h.Base::ver();
  cout << h.suma() << endl;
  cout << h.suma_x() << endl;
  system("PAUSE");  return EXIT_SUCCESS;
}
```

Hija::x oculta el atributo x de Base → para acceder a él hay que poner Base::x
Hija::mutar(float a) sobrecarga y oculta mutar() de Base -> hay que poner Base::mutar() para acceder
Hija::ver() redefine ver() de Base y oculta ver() y ver(int a) de Base → Base::ver() y Base::ver(x)

Constructores en clases derivadas

El constructor de la clase derivada debe encargarse de los atributos que son exclusivos suyos e invocar en la zona de inicializadores al constructor de su clase base, para que éste se encargue de los atributos que le pertenecen (y que la clase derivada hereda).

Esquema del constructor en la clase derivada:

```
Derivada::Derivada(parámetros_base, parámetros_derivada): Base(parámetros_base), otros_inicializadores {
   codigo_adicional; //aqui tratamos los atributos exclusivos de la clase Derivada, no heredados de la Base
   …;
}
```

Si no invocamos al contructor de la clase base, el compilador invocará en la zona de inicializadores al constructor por defecto de la clase base (ídem **a los constructores por defecto de los atributos exclusivos de la derivada que son objetos de otras clases**).

```
Derivada::Derivada(parametros): otros {
  codigo_adicional
}
```

equivale a poner

```
Derivada::Derivada(parametros): Base( ),
   atributoObjeto1( ),  …, atributoObjetoN( ),otros {
  codigo_adicional;
}
```

Ej1: Si la clase Hija (hereda de Base) tiene 3 atributos (const int x; OtraClase y, z;). Al hacer:

```
Hija:: Hija (parametros):y(p) {
  codigo_adicional
}
```

equivale a

```
Hija:: Hija (parametros): Base( ),y(p), z( ) {
  codigo_adicional;
}
```

//constructor de copìa

```
Hija:: Hija (const Hija& h):y(h.y) {
  codigo_adicional
}
```

equivale a

```
Hija:: Hija (const Hija& h): Base( ), y(h.y), z( ) {
  codigo_adicional;
}
```

Si la clase Base o la clase OtraClase no tiene un constructor por defecto (sin parámetros), lo anterior dará error ya que se invoca Base( ) y z( ) sin parámetros

Constructores en clases derivadas

Si en la clase derivada no hay ningún constructor (ni de copia), el compilador crea:

- uno de oficio, que llama al constructor por defecto (ya sea de oficio o no) de su clase base y a los constructores por defecto de los atributos exclusivos de la derivada que son objetos de otras clases

```
Derivada::Derivada( ) {
  //nada
}
```

equivale a

```
Derivada::Derivada( ): Base( ),
   atributoObjeto1( ),  …, atributoObjetoN( ),{
  //nada
}
```

- un constructor de copia (que llama al constructor de copia de su clase base y hace copia binaria de los atributos exclusivos de la clase derivada)

```
Derivada::Derivada(const Derivada& d): Base(d),  atributo1(d.atributo1), …, atributoN(d.atributoN) {
   // nada
 }
```

Destructores en clases derivadas

Cuando se invoca el destructor de una clase derivada, se ejecuta su código y al finalizar invoca automáticamente al destructor de la clase base. No hace falta invocarlo. El destructor de la clase derivada sólo debe liberar la memoria de los atributos dinámicos propios de la clase derivada (el destructor de la clase base se ocupa de sus atributos dinámicos que la clase Derivada hereda).

Los destructores se llaman en orden inverso en el que se llaman a los constructores. Primero se ejecuta el cuerpo del destructor de la clase derivada, después son llamados los destructores de sus atributos objetos miembros (si los tuviera) y por último se ejecuta el destructor de la clase base, que a su vez actúa igual (ejecuta cuerpo destructor, invoca destructores de sus atributos y ejecuta destructor clase base).

Ejercicio: Dada las siguiente clases, implementa **el mínimo número** de métodos necesarios para que el siguiente main( ) funcione correctamente:

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

class Clase {
  int n;
public:
  Clase(int x) { n=x; }
//A RELLENAR POR EL ALUMNO
};


class Clase2 {
  const int tamanio;
  int *tabladinamica; //el constructor crea la tabla con un tamaño fijo indicado por parametro
  Clase y;
public:
  //A RELLENAR POR EL ALUMNO
};


class Clase3: public Clase2 {
  static int n; //para saber cuantos objetos de tipo Clase3 se crean
  Clase z;
public:
  //A RELLENAR POR EL ALUMNO
};


int main() {
  Clase a(2), b(a);
  Clase2 x(2), y(3, a);  //x con un atributo Clase de valor 0 y una tabla de 2 elementos
  Clase3 h(2, a, b), hc(h); //2 es el tamaño de la tabla dinámica, a es y, b es z

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Indica por qué implementas esos métodos y no otros más

Operación de asignación y constructor de copia en clases derivadas

Una clase (base o derivada) que use memoria dinámica por tener un atributo (no heredado) tipo puntero, deberá definir el destructor, el constructor de copia y el operador de asignación.

Cada clase derivada solo tiene que mirar sus atributos exclusivos (no los que hereda) de forma que si sus atributos exclusivos no contienen punteros, no será necesario implementar ni el destructor (no hay nada que liberar) ni el constructor de copia ni el operador de asignación ya que los que genera el compilador (en caso que no se codifiquemos éstos) funcionan correctamente:

- El constructor de copia que genera el compilador llama al constructor de copia de la clase base, y hace una copia binaria de los atributos exclusivos de la derivada, invocando el constructor de copia de dichos atributos exclusivos en la zona de inicializadores.
- El operador de asignación que genera el compilador llama al operador de asignación de la clase base, y hace una copia binaria de los atributos exclusivos de la derivada, invocando el operador de asignación de dichos atributos exclusivos.

Los que genera de oficio el compilador (en caso que no sobrecarguemos) son estos:

Constructor de copia

```
Derivada::Derivada(const Derivada& obj): Base(obj),
  atributo1(obj.atributo1), …, atributoN(obj.atributoN) {
    // nada
 }
```

Operador de asignación

```
Derivada& Derivada::operator=(const Derivada& obj) {
   if (this!=&obj) {
      Base::operator=(obj);
      atributo1=obj.atributo1;
      …;
      atributoN=obj.atributoN;
    }
    return *this;
}
```

Ej2: Si la clase Derivada tiene 2 atributos x de tipo entero y f de tipo Fecha:

Si escribimos esto:

```
Derivada::Derivada(const Derivada& obj) { codigo }

Derivada& Derivada::operator=(const Derivada& obj) {codigo }
```

Es equivalente a escribe esto:

```
Derivada::Derivada(const Derivada& obj)
                  : Base(),  f (obj.f )   { codigo }

Derivada& Derivada::operator=(const Derivada& obj) {codigo }
```

Por ello, si queremos que haga una cosa diferente, debemos seguir el siguiente esquema:

Esquema del constructor de copia en la clase derivada:

```
Derivada::Derivada(const Derivada& objeto): Base(objeto), otros_inicializadores {
   codigo_adicional; //aqui tratamos los atributos exclusivos de la clase Derivada, no heredados de la Base
   …;
}
```

Como vemos en el inicializador invocamos el constructor de copia de la clase Base el cual se encarga de los atributos heredados por la clase Derivada. Base(objeto) es correcto ya que a un objeto Base se le puede asignar un objeto de tipo Derivada.

Esquema del operador de asignación en la clase derivada:

```
Derivada& Derivada::operator=(const Derivada& objeto_lado_derecho) {
   if (this != &objeto_lado_derecho) {
      Base::operator=(objeto_lado_derecho);  //llamada al operador = sobrecargado de la clase Base
      …; //aqui se trata los atributos exclusivos de la clase Derivada, no heredados de la clase Base
   }
   return *this;
}
```

Ejemplo: Crear una clase Base (suponed que tiene atributos dinámicos) que tenga un método ver( ). Crear una clase derivada Hija que sea igual que la Base pero que permita sumar. Crear una clase que derive de Hija que tenga un atributo adicional y que permita hacer lo mismo que Hija.
Solución: Al ser la clase Base dinámica hay que crear el constructor, constructor de copia, destructor y sobrecargar el operador de asignación. Las derivadas de Base tendrán que crearlas si sus atributos adicionales son dinámicos. En caso de no tener atributos adicionales o no ser éstos dinámicos, no tienen necesidad de crear el constructor de copia ni operator= ya que los que genera de oficio el compilador (hace copia binario de los datos exclusivos de la derivada e incova al constructor copia y operator= del padre) funcionan correctamente. Lo mismo ocurre con el destructor (si no hay atributos adicionales o no son dinámicos no hace falta destructor). El constructor si es necesario.

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

class Base {
protected:
  int x;
public:
  Base(int a=0) { x=a; cout << "\nBase"; }
  Base(const Base &c) { x=c.x; cout << "\ncopia Base"; }
  ~Base() { cout << "\nDestruye Base"; }
  Base& operator=(const Base &c) {
    x=c.x;    cout << "\nasigna Base";     return *this;
  }
  void ver() { cout << x << "\n"; }
};

class Hija: public Base {
public:
  Hija(int a=0): Base(a) { cout << " - Hija";}
  Hija(const Hija &c): Base(c) { cout << " - copia Hija"; }
  ~Hija() { cout << "\nDestruye Hija - "; }
  Hija& operator=(const Hija &c) {
     Base::operator=(c); cout << " - asigna Hija";
     return *this;
  }
  Hija operator+(Hija h) { Hija hi; hi.x=h.x+x; return hi; }
};

class Nieta: public Hija {
  int y;
public:
  Nieta(int x, int y): Hija(x), y(y) { cout << " - Nieta";}
  Nieta(const Nieta &c): Hija(c) {
     y=c.y; cout << " - copia Nieta";
  }
  ~Nieta() { cout << "\nDestruye Nieta - "; }
  Nieta& operator=(const Nieta &c) {
     Hija::operator=(c); y=c.y; cout << " - asigna Nieta";
     return *this;
  }
  Nieta operator+(const Nieta &n) {
      Nieta ni(n.x+x, n.y+y);
      return ni;
  }
  void ver() { cout << x << "," << y << "\n"; }
};
```

```
int main(int argc, char *argv[]) {
 { //ponemos llaves para ver los mensajes de destructor
  Base a, b(a);                  //objetos locales
  Hija c(3), d(c);               //objetos locales
  Nieta e(2,3), f(e), g(7,1);  //objetos locales
  cout << "\n----\n";
  a.ver(); b.ver(); c.ver(); d.ver(); e.ver(); f.ver(); g.ver();
  cout << "----";
  d=d+c;
  f=g;
  g=e+Nieta(7,0);
  cout << "\n----\n";
  a.ver(); b.ver(); c.ver(); d.ver(); e.ver(); f.ver(); g.ver();
  cout << "----";
 } //cuando sale de las llaves se destruyen los objetos
 cout << "\n----\n";
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Salida por pantalla

```
Base                                  //a
copia Base                            //b(a)
Base - Hija                           //c(3)
copia Base - copia Hija               //d(c)
Base - Hija - Nieta                   //e(2,3)
copia Base - copia Hija - copia Nieta    //f(e)
Base - Hija - Nieta                        //g(7,1)
----
0  0  3  3  2,3  2,3  7,1
----
copia Base - copia Hija                //d=d+c → operator+(Hija h)
Base – Hija                            //d=d+c →  Hija hi;
asigna Base - asigna Hija              //d=d+c →  =
Destruye Hija - Destruye Base          //d=d+c →  hi local operator+
Destruye Hija - Destruye Base          //d=d+c →  h parametro operator+
asigna Base - asigna Hija - asigna Nieta   //f=g
Base - Hija - Nieta                    //g=e+Nieta(7,0); → Nieta(7,0);
Base - Hija - Nieta                    //Nieta ni(n.x+x, n.y+y);
asigna Base - asigna Hija - asigna Nieta        // g=e+Nieta(7,0); → =
Destruye Nieta - Destruye Hija - Destruye Base  //ni de operator+
Destruye Nieta - Destruye Hija - Destruye Base  // Nieta(7,0);
----
0  0  3  6  2,3  7,1  9,3
----
Destruye Nieta - Destruye Hija - Destruye Base     //g
Destruye Nieta - Destruye Hija - Destruye Base     //f
Destruye Nieta - Destruye Hija - Destruye Base     //e
Destruye Hija - Destruye Base                      //d
Destruye Hija - Destruye Base                      //c
Destruye Base                                      //b
Destruye Base                                      //a
----
```

En el ejemplo hemos implementado todos por motivos pedagógicos, para que se vea como se hace…

Ejemplo: Crear una clase Base para guardar una palabra con un método set para cambiar la letra indicada por un guión y un método ver para mostrar la palabra por pantalla. Crear una clase derivada Hija que sea igual que la Base pero que permita comparar con el signo = =. Crear una clase que derive de Hija que tenga un atributo adicional y que permita hacer lo mismo que Hija.
Solución: Al ser la clase Base dinámica hay que crear el constructor, constructor de copia, destructor y sobrecargar operator=. Las derivadas de Base las crearán si sus atributos exclusivos son dinámicos.

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

class Base {
protected:
  char *s;
public:
  Base(char *s="nada") {
     this->s=new char[strlen(s)+1];
     strcpy(Base::s,s); cout << "\nBase";
  }
  Base(const Base &c) {
    s=new char[strlen(c.s)+1];
    strcpy(s,c.s); cout << "\ncopia Base";
  }
  ~Base() { delete [] s; cout << "\nDestruye Base"; }
  Base& operator=(const Base &c) {
    delete [] s;    s=new char[strlen(c.s)+1];
    strcpy(s,c.s); cout << "\nasigna Base";
    return *this;
  }
  void set(char a) { for (int i=0; s[i]!=0; i++)
                          if (s[i]==a) s[i]='-'; }
  void ver() { cout << s << "\n"; }
};

class Hija: public Base {
public:
  Hija(char *s="nada"): Base(s) { cout << " - Hija";}
  Hija(const Hija &c): Base(c) { cout << " - copia Hija"; }
  ~Hija() { cout << "\nDestruye Hija - "; }
  Hija& operator=(const Hija &c) {
    Base::operator=(c); cout << " - asigna Hija";
    return *this;
  }
  bool operator==(Hija h) { return (strcmp(s, h.s)==0); }
};

class Nieta: public Hija {
  int n;
public:
  Nieta(char *s, int i): Hija(s), n(i) { /*n=i;*/ cout << " - Nieta"; }
  Nieta(const Nieta &c): Hija(c) {
     n=c.n; cout << " - copia Nieta";
  }
  ~Nieta() { cout << "\nDestruye Nieta - "; }
  Nieta& operator=(const Nieta &c) {
     Hija::operator=(c); n=c.n; cout << " - asigna Nieta";
     return *this;
  }
  bool operator==(const Nieta &ni) {
     return (Hija::operator==(ni) && n==ni.n);
  }
  void ver() { cout << s << "(" << n << ")\n"; }
};

int main(int argc, char *argv[]) {
 { //ponemos llaves para ver los mensajes de destructor
  Base a; //objetos locales
  Hija b("si"), c(b);
  Nieta d("hola",1), e(d), f("bienvenido",2);
  a.set('a'); b.set('i'); f.set('n');
  cout << "\n----\n";
  a.ver(); b.ver(); c.ver(); d.ver(); e.ver(); f.ver();
  cout << "----";
  if (d==e)
    f=d;
  if (!(b==c))
    b=c;
  f.set('h'); c.set('s');
  cout << "\n----\n";
  a.ver(); b.ver(); c.ver(); d.ver(); e.ver(); f.ver();
  cout << "----";
 } //al salir de las llaves se destruyen los objetos locales
 cout << "\n----\n";
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

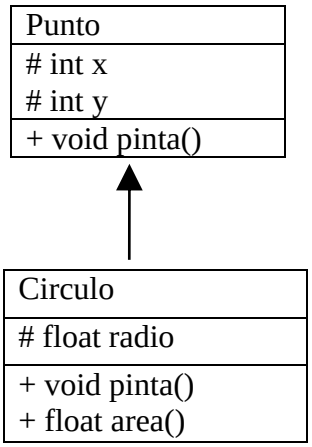
Salida por pantalla

```
Base
Base - Hija
copia Base - copia Hija
Base - Hija - Nieta
copia Base - copia Hija - copia Nieta
Base - Hija - Nieta
----
n-d-    s-    si    hola(1)    hola(1)    bie-ve-ido(2)
----
copia Base - copia Hija
Destruye Hija - Destruye Base
asigna Base - asigna Hija - asigna Nieta
copia Base - copia Hija
Destruye Hija - Destruye Base
asigna Base - asigna Hija
----
n-d-    si    -i    hola(1)    hola(1)    -ola(1)
----
Destruye Nieta - Destruye Hija - Destruye Base
Destruye Nieta - Destruye Hija - Destruye Base
Destruye Nieta - Destruye Hija - Destruye Base
Destruye Hija - Destruye Base
Destruye Hija - Destruye Base
Destruye Base
```

Las lineas grises se pueden eliminar sin problemas porque:
-Los destructores Hija y Nieta no hacen nada
-El constructor de copia de Hija solo invoca al copia del padre y el de Nieta invoca al copia del padre y hace n=c.n (copia binaria) y el de oficio que el compilador genera si nosotros no lo definimos invoca al copia del padre y hace copia binaria de los datos (si hay)
-idem con operator=

Adicionalmente

Importante: **privilegio por defecto, private…**




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

class Punto {
protected:
     int x;
     int y;
public:
     void pinta( ) {
          cout << "(" << x << "," << y << ")";
     }
};

class Circulo: public Punto {
protected:
     float radio;
public:
     void pinta() {
          Punto::pinta();
          cout << "-" << radio;
     }
     float area( ) {
          return 2*3.14*radio;
     }
};

int main(int argc, char *argv[])
{
      Circulo c;
      c.pinta(); cout << endl;
      system("PAUSE");
      return EXIT_SUCCESS;
}
```

Al no haber constructor el compilador crea uno por defecto y uno de copia. El de por defecto se limita a llamar al constructor de su clase base (si la tiene) y el de copia llama a su constructor padre y hace copia binaria de los atributos exclusivos de la clase.
Como tampoco hay destructor el compilador crea uno de oficio que no hace nada. Lo mismo ocurre con la sobrecarga del operador de asignación: el compilador crea uno de oficio que se limita a llamar al operator= de la clase base (si la tiene) y hace copia binaria de los atributos exclusivos de la clase, devolviendo una referencia del propio objeto:

```
Punto::Punto ( ) {  }
Punto::Punto (const Punto & obj)
  {  x=obj.x; y=obj.y; }
Punto::~ Punto ( ) {  }
Punto & Punto::operator= (const Punto & obj ) {
   x=obj.x; y=obj.y;
   return *this;
}

Circulo::Circulo ( ): Punto( ) {  }
Circulo::Circulo (const Circulo & obj) :Punto (obj)
       {  radio=obj.radio; }
Circulo::~ Circulo ( ) {  }
Circulo&Circulo::operator= (const Circulo& obj ) {
   Punto::operator=(obj);
   radio=obj.radio;
   return *this;
}
```

Pantalla:

```
(2009116333,2088810217)-5.31691e+036
```

Habría que añadir **lo que está en negrita** para corregir los errores (eso hace que lo marcado en rojo el compilador no lo genere):

```
class Punto {
public:
  Punto (int x=0, int y=0);
};

Punto::Punto (int x, int y)
  { this->x=x; Punto::y=y;  }

class Circulo: public Punto {
public:
  Circulo (int x=0, int y=0, float r=0 );
}

Circulo::Circulo (int x, int y, float r )
     : Punto(x, y) { radio=r; }
```

Pantalla: (añadiendo lo que está en **negrita**)

```
(0,0)-0
```

Conversiones entre objetos de clases base y clases derivadas

Un ***objeto de una clase derivada*** se puede asignar a un ***objeto de la clase base***, lo contrario no:

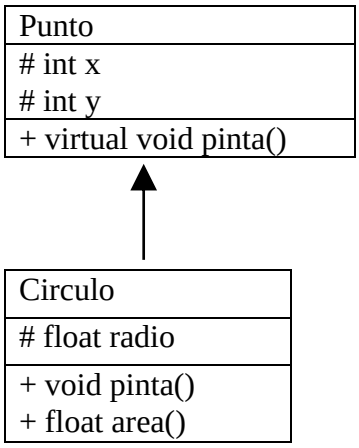
objeto_Base = objeto_Derivada; //correcto: copia en objeto_Base los valores de los atributos comunes
objeto_Derivada = objeto_Base; //ERROR: Derivada puede tener más atributos que Base

Punteros y referencias

Un puntero o una referencia de una clase Base puede apuntar a un objeto de su propia clase o a cualquier objeto **de una clase derivada** de ella (hija, nieta, etc.).

- El tipo del puntero o de la referencia determina que métodos y a qué atributos puede acceder.
- El método que se ejecuta es el de la clase de la que es el puntero o la referencia

Base ba, *b=&ba;
Derivada de, *d=&de;
b=&de; //el puntero de la clase Base puede apuntar a un objeto de la clase derivada porque todos los métodos
//que pueden invocar el puntero Base (que son los que hay en la Base) existen en la clase derivada
d=&ba; //el puntero d de la clase Derivada **no debe** apuntar a un objeto de la clase Base porque podría
// hacer esto: **d->método_que_no_existe_en_clase_base();**



Pantalla:

```
(1,1)-5  (3,5)-2  (1,1)-5  (2,3)  (2,3)
(1,1)
31.4
(3,5)
(1,1)  (1,1)
(1,1)  (3,5)
(1,1)  (3,5)
```

Cuando un puntero o referencia de una clase base apunta a un objeto de una clase derivada, los métodos que se invocan son los de la clase a la que pertenece el puntero o referencia no los de la clase a la que pertenece el objeto.

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

class Punto {
protected:     int x, y;
public:
     Punto (int x=0, int y=0)  { this->x=x; Punto::y=y;  }
     void pinta( ) {  cout << "(" << x << "," << y << ")\n";   }
};

class Circulo: public Punto {
protected:     float radio;
public:
    Circulo (int x=0, int y=0, float r=0 ): Punto(x, y) { radio=r; }
     void pinta( ) {  cout << "(" << x << "," << y << ")-" << radio << "\n"; }
     float area( ) { return 2*3.14*radio; }
};

void noMiembro1(Punto &p) { p.pinta(); }
void noMiembro2(Punto p) { p.pinta(); }

int main(int argc, char *argv[]) {
   Circulo x(1,1,5), z(3,5,2), *px;
   Punto y(2,3), *py;
   px=&x; py=&y;
   x.pinta(); z.pinta(); px-> pinta(); y.pinta(); py->pinta();
   px-> Punto::pinta();
   cout << px->area() << endl;
   y=z; // No necesita casting. Hace copia binaria de los atributos comunes
  //x=y;    ERROR no sabe convertir un Circulo en un Punto
        //el ERROR se quita sobrecargando Circulo::operator=(Punto p)
   py=&x; // No necesita casting  (idem py=px)
   y.pinta(); py->pinta();
   py-> Punto::pinta();
   //py-> area();  ERROR la clase Punto no tiene un método area( )
   noMiembro1(x); noMiembro1(z);
   noMiembro2(x); noMiembro2(z);
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Punteros y referencias: Métodos virtuales (Polimorfismo)

Un puntero o una referencia de una clase Base puede apuntar a un objeto de su propia clase o a cualquier objeto **de una clase derivada** de ella (hija, nieta, etc.).

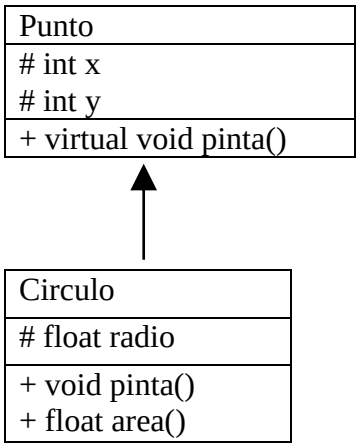
- El tipo del puntero o de la referencia determina que métodos y a qué atributos puede acceder.
- El método que se ejecuta es el de la clase de la que es el puntero o la referencia

Para hacer que el método que se ejecuta por un puntero o una referencia no **sea** el de la clase del puntero o la referencia, sino **el del objeto al que apunta el puntero o la referencia hay que declarar dicho método como virtual.**

Cuando un **método** se declara **virtual en una clase Base**, las redefiniciones de dicho método son **también virtual en las clases derivadas aunque explícitamente no se indique**.

IMPORTANTE: El polimorfismo sólo se puede conseguir con punteros y referencias. Además la clase debe ser polimórfica, es decir, debe tener al menos un método declarado como virtual.

Métodos virtuales: ejemplo.



Salida con virtual:

```
(1,1)-5  (3,5)-2  (1,1)-5  (2,3)  (2,3)
(1,1)
31.4
(3,5)
(1,1)-5  (1,1)
(1,1)-5  (3,5)-2
(1,1)    (3,5)
```

Salida sin virtual:

```
(1,1)-5  (3,5)-2  (1,1)-5  (2,3)  (2,3)
(1,1)
31.4
(3,5)
(1,1)  (1,1)
(1,1)  (3,5)
(1,1)  (3,5)
```

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

class Punto {
protected:     int x, y;
public:
     Punto (int x=0, int y=0)  { this->x=x; Punto::y=y;  }
     virtual void pinta( ) {  cout << "(" << x << "," << y << ")\n";   }
};

class Circulo: public Punto {
protected:     float radio;
public:
    Circulo (int x=0, int y=0, float r=0 ): Punto(x, y) { radio=r; }
     void pinta( ) {  cout << "(" << x << "," << y << ")-" << radio << "\n"; }
     float area( ) { return 2*3.14*radio; }
};

void noMiembro1(Punto &p) { p.pinta(); }
void noMiembro2(Punto p) { p.pinta(); }

int main(int argc, char *argv[]) {
   Circulo x(1,1,5), z(3,5,2), *px;
   Punto y(2,3), *py;
   px=&x; py=&y;
   x.pinta(); z.pinta(); px-> pinta(); y.pinta(); py->pinta();
   px-> Punto::pinta();
   cout << px->area() << endl;
   y=z; // No necesita casting. Hace copia binaria de los atributos comunes
  //x=y;    ERROR no sabe convertir un Circulo en un Punto
        //el ERROR se quita sobrecargando Circulo::operator=(Punto p)
   py=&x; // No necesita casting  (idem py=px)
   y.pinta(); py->pinta();
   py-> Punto::pinta();
   //py-> area();  ERROR la clase Punto no tiene un método area( )
   noMiembro1(x); noMiembro1(z);
   noMiembro2(x); noMiembro2(z);
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Nota: Por eso a veces interesa que una función tenga el parámetro por referencia, aunque no haga falta, como por ejemplo en **noMiembro1(Punto &p)**. En ese caso como medida de seguridad se debería pasar como referencia constante, y los métodos que no modifican la clase etiquetarlos como **const**

Utilidad del polimorfismo:
Permite programar métodos y funciones no miembros que usan objetos cuyo tipo no es conocido hasta el momento de la ejecución. En tiempo de ejecución se ejecuta el método correspondiente al objeto correcto
IMPORTANTE: El polimorfismo sólo se puede conseguir con punteros y referencias. Además la clase debe ser polimórfica, es decir, debe tener al menos un método declarado como virtual.

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

class Date {
  int dia, anio;
  char *mes;
public:
  Date(int d, const char *m, int a) {
   dia=d; anio=a; mes=new char[strlen(m)+1]; strcpy(mes, m);
  }
  ~Date() { delete [] mes; }
  int getDia() const { return dia; }
  const char *getMes() const { return mes; }
  int getAnio() const { return anio; }
};

class Vehiculo {
  char *modelo;
  Date fecha;
 public:
  Vehiculo(char *m, Date &f)
     :fecha(f.getDia(), f.getMes(), f.getAnio()) {
    modelo=new char[strlen(m)+1];    strcpy(modelo, m);
  }
  Vehiculo(const Vehiculo &v):fecha(v.fecha.getDia(),
                v.fecha.getMes(), v.fecha.getAnio()) {
    modelo=new char[strlen(v.modelo)+1];
    strcpy(modelo, v.modelo);
  }
  ~Vehiculo() { delete [] modelo; }
  virtual void ver(ostream &s) const;
};

void Vehiculo::ver(ostream &s) const {
  s << "modelo: " << modelo << " fecha: " << fecha.getDia()
  << "/"  << fecha.getMes() << "/" << fecha.getAnio() << endl;
}

class Camion: public Vehiculo {
  char *empresa;
  int tara;
public:
  Camion(char *m, Date &f, char *e, int t):Vehiculo(m,f) {
    tara=t; empresa=new char[strlen(e)+1];  strcpy(empresa, e);
  }
  Camion(const Camion &v): Vehiculo(v) {
    empresa=new char[strlen(v.empresa)+1];
    strcpy(empresa, v.empresa);    tara=v.tara;
  }
  ~Camion() { delete [] empresa; }
  void ver(ostream &s) const;
};

void Camion::ver(ostream &s) const {
  Vehiculo::ver(s);
  s << "empresa: " << empresa << " peso: " << tara << endl;
}
```

```
class Coche: public Vehiculo {
  char *amo;
public:
  Coche(char *m, Date &f, char *amo):Vehiculo(m,f) {
    this->amo=new char[strlen(amo)+1];
    strcpy(this->amo, amo);
  }
  Coche(const Coche &v): Vehiculo(v) {
    amo=new char[strlen(v.amo)+1];
    strcpy(amo, v.amo);
  }
  ~Coche() { delete [] amo; }
  void Coche::ver(ostream &s) const {
    Vehiculo::ver(s);
    s << "propietario: " << amo << endl;
  }
};
void info1(Vehiculo v) {  v.ver(cout); } //no polimorfismo

void info2(Vehiculo &v) { v.ver(cout); } //si polimorfismo

void info3(Vehiculo *v) { v->ver(cout); }//si polimorfismo

int main(int argc, char *argv[]) {
  Date f1(10,"abril",1990);
  Camion c("Ford",f1,"Bimbo",1000);
  Coche p("Opel", f1, "juan");
  c.ver(cout);       //(1)
  p.ver(cout);       //(2)
  info1(c);          //(3)
  info2(c);          //(4)
  info3(&c);         //(5)
  Vehiculo v(p); //Vehiculo v=p
  v.ver(cout);       //(6)   //no hay polimorfismo
  Vehiculo *pv;
  pv=&p;
  pv->ver(cout);  //(7)    //hay polimorfismo
 cout << "\n----\n";
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Pantalla:

```
modelo: Ford fecha: 10/abril/1990          (1)
empresa: Bimbo peso: 1000
modelo: Opel fecha: 10/abril/1990          (2)
propietario: juan
modelo: Ford fecha: 10/abril/1990          (3)
modelo: Ford fecha: 10/abril/1990          (4)
empresa: Bimbo peso: 1000
modelo: Ford fecha: 10/abril/1990          (5)
empresa: Bimbo peso: 1000
modelo: Opel fecha: 10/abril/1990          (6)
modelo: Opel fecha: 10/abril/1990          (7)
propietario: juan
----
Presione una tecla para continuar . . .
```

Clase abstracta:
virtual
Método virtual puro.

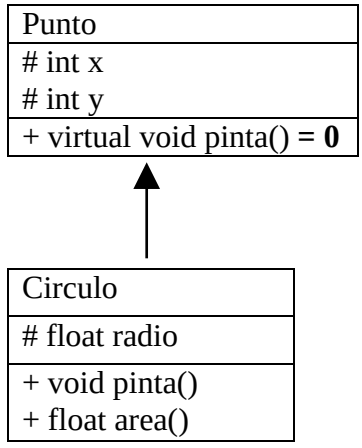
Una clase abstracta es una clase que **no se puede instanciar** (no se pueden crear objetos de dicha clase). Una clase es abstracta si **tiene al menos un método virtual puro**.
Un método virtual puro es un método virtual que **está definido pero no implementado**. Se declara así:

```
virtual metodo( ) const=0; //metodo virtual puro
```

Las clases que deriven de una clase abstracta deben implementar **todos** los métodos virtuales puros que heredan, si no redefinen **todos** se convierten también en clases abstractas.

Utilidad de las clases abstractas:
Proporcionar una plantilla modelo o interface con los métodos que deben implementar las clases que deriven de ella.

Aunque no se puede crear objetos de una clase abstracta, ***se puede crear punteros de la clase abstracta, pues es a través de ellos como será posible manejar objetos de todas las clases derivadas***.



Ojo: virtual se pone en el .hpp pero no en el .cpp

```
class Punto{ //clase abstracta
protected:
        int x;
        int y;
public:
        virtual void pinta() = 0;
};

class Circulo: public Punto{
protected:
        float radio;
public:
        void pinta(){
                cout << "Circulo\n";
        }
        float area(){
                return 2*3.14*radio;
        }
};

int main(int argc, char *argv[])
{
   Circulo c;
   Punto *p; // Se permiten punteros
   p = &c;    // No necesita casting
   p-> pinta();
   //p->Punto::pinta(); No permitido
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Static y Dynamic cast

A la hora de hacer una conversión de tipo (cast) se puede hacer estática o dinámicamente.

La conversión dinámica sólo se puede hacer con punteros o referencias y además el tipo al que se quiere convertir debe ser un tipo polimórfico, es decir, **la clase debe tener métodos virtuales o tenerlo la clase de la que hereda** (cuando una clase hereda un método virtual, si la clase derivada la sobreescribe también es virtual aunque no se indique explícitamente).

Para hacer la conversión estática se utiliza el operador static_cast
Para hacer la conversión dinámica se utiliza el operador dynamic _cast

Sintaxis: tipo representa el tipo al que se quiere convertir el objeto

Conversión estática: static_cast<tipo>(objeto)
(tipo)objeto

Conversión dinámica: dynamic_cast<tipo *>(puntero_a_objeto)
dynamic_cast<tipo &>(referencia_a_objeto)

Conversión estática

```
int a;
float b;
b = static_cast<float>(a);      //conversion explicita
b = (float) a;                  //conversion explicita (equivalente)
b = a;                          //conversion implícita
```

```
#include <iostream>
using namespace std;
class Tiempo {
public:
        Tiempo(int h=0, int m=0) : hora(h), minuto(m) {}
        void Mostrar() { cout << hora << ":" << minuto << endl; }
        operator int() {
                return hora*60+minuto;
        }
private:
        int hora;
        int minuto;
};

int main() {
        Tiempo Ahora(12,24);
        int minutos;
        Ahora.Mostrar();
        minutos = static_cast<int> (Ahora);
        minutos = (int) Ahora;
        // minutos = Ahora; // Igualmente legal, pero implícito
        cout << minutos << endl;
}
```

Static y Dynamic cast

Conversiones entre objetos de clases base y clases derivadas

Un ***objeto de una clase derivada*** se puede asignar a un ***objeto de la clase base***, lo contrario no:

objeto_Base = objeto_Derivada; //correcto: copia en objeto_Base los valores de los atributos comunes
objeto_Derivada = objeto_Base; //ERROR: Derivada puede tener más atributos que Base

Punteros y referencias

Un puntero o una referencia de una clase Base puede apuntar a un objeto de su propia clase o a cualquier objeto **de una clase derivada** de ella (hija, nieta, etc.).

- El tipo del puntero o de la referencia determina que métodos y a que atributos puede acceder.
- El método que se ejecuta es el de la clase de la que es el puntero o la referencia

```
Base ba, *b=&ba;
Derivada de, *d=&de;
b=&de; //el puntero de la clase Base puede apuntar a un objeto de la clase derivada porque todos los métodos
b=d;   //que pueden invocar el puntero Base (que son los que hay en la Base) existen en la clase derivada
d=&ba; //el puntero d de la clase Derivada no debe apuntar a un objeto de la clase Base porque podría
d=b;   // hacer esto: d->método_que_no_existe_en_clase_base();
```

Un puntero o referencia de una clase Base puede apuntar a una clase Derivada, es decir, podemos convertir un puntero (referencia) a Derivada en un puntero (referencia) a Base (conversión ascendente) porque todos los métodos que se pueden invocar con el puntero Base existen en la clase Derivada (ya que la Derivada hereda todo lo de la Base y añade lo suyo propio).

Un puntero o referencia de una clase Derivada NO DEBE apuntar a una clase Base, es decir, no debemos convertir un puntero (referencia) a Base en un puntero (referencia) a Derivada (conversión descendente) porque una vez convertido, podemos invocar con el puntero a Derivada métodos exclusivos de la Derivada que no existen en la clase Base.

NO DEBE no implica que esté prohibido, de hecho podemos forzar la conversión explícitamente. ¿Por qué? Porque puede darse estas 2 situaciones:

```
Base *b;
Derivada de, *d;
b=&de; //b apunta a un objeto Derivada
//d=b; //no lo permite implícitamente
d=(Derivada *)b;                  //permitido
d=static_cast< Derivada *>(b);    //permitido
d->metodoExclusivoDerivada( ); //OK
```

```
Base ba,*b;
Derivada de, *d;
b=&ba; //b apunta a un objeto Base
//d=b; //no lo permite implícitamente
d=(Derivada *)b;                  //permitido
d=static_cast< Derivada *>(b);    //permitido
d->metodoExclusivoDerivada( ); //ERROR!!!
```

En la derecha se produce error porque el puntero a derivada d apunta a un objeto Base (que era a lo que apuntaba el puntero Base convertido) y hemos invocado un método exclusivo de la derivada que el objeto Base no tiene. En la izquierda no hay error porque el puntero a derivada d apunta a un objeto Derivada (que era a lo que apuntaba el puntero Base convertido).

Por ello, en caso de querer convertir de puntero Base a puntero Derivada (conversión descendente), es decir, hacer que un puntero Derivada apunte a un puntero Base debemos saber a qué es lo que apunta el puntero Base.

Eso se puede saber en tiempo de ejecución con los operadores **dynamic_cast** y **typeinfo**.

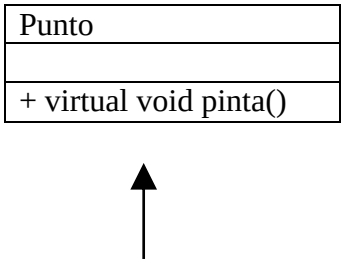
Static y Dynamic cast

Información de tipo (clase) en tiempo de ejecución: #include <typeinfo>
typeid y dynamic_cast. **(para usarlos la clase debe tener al menos un método virtual OJO!!!)**

- El operador **dynamic_cast permite detectar en tiempo de ejecución si una conversión entre punteros o referencias se ha efectuado correctamente o no**.
- El operador **typeid permite preguntar en tiempo de ejecución si un puntero apunta a un objeto de un tipo determinado**.

Dado un puntero p: **¿es el objeto apuntado por p de tipo T?**
typeid(*p) = = typeid(T); → true si el objeto apuntado por p es de tipo T, false en caso contrario.
dynamic_cast <T *>(p); → (T *) si el objeto apuntado por p es de tipo T, 0 (NULL) si no

Utilidad:
Un puntero a una clase Base puede apuntar a cualquier objeto de las clases Derivadas, pero solo puede invocar los métodos (virtuales o no) definidos en la clase Base. Si necesitamos invocar un método exclusivo de la clase Derivada, podemos hacerlo convirtiendo el puntero Base a Derivada haciendo que un puntero a Derivada apunta al puntero a Base con un cast, pero para ello debemos asegurarnos (con **typeid** o **dynamic_cast**) que realmente el puntero a Base apunta a un objeto Derivada.



Pantalla:

```
No existe en Punto
No existe en Punto
7Circulo
---
Circulo
array[0] es un Circulo
No existe en Punto
Circulo
array[1] es un Circulo
No existe en Punto
Circulo
array[2] es un Circulo
No existe en Punto
Punto
array[3] es un Punto
```

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

class Punto {
public:
   virtual void pinta(){ cout << "Punto\n"; }
};

class Circulo:public Punto {
public:
   virtual void pinta(){ cout << "Circulo\n"; }
   void otroMetodo(){ cout << "No existe en Punto\n"; }
};

int main(int argc, char *argv[]) {
   Punto **array = new Punto *[4];
   array[0] = new Circulo;
   array[1] = new Circulo( ); //los ( ) no son necesarios...
   array[2] = new Circulo;
   array[3] = new Punto;
   if (typeid(*array[0])==typeid(Circulo)) {
      ((Circulo *)array[0])->otroMetodo();
      static_cast<Circulo *>(array[0])->otroMetodo();
      cout << typeid(*array[0]).name() << "\n";
   }
   cout << "---\n";
   for (int i=0; i<4; i++) {
      array[i]->pinta();
      if ( Circulo *c = dynamic_cast<Circulo*>(array[i]) ) {
         cout << "array[" << i << "] es un Circulo\n";
         c->otroMetodo();
      }else
         cout << "array[" << i << "] es un Punto\n";
   }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Permite acceder con un puntero base a funciones sólo existentes en la derivada.

Static y Dynamic cast

Información de tipo (clase) en tiempo de ejecución: #include <typeinfo>
typeid y dynamic_cast. **(para usarlos la clase debe tener al menos un método virtual OJO!!!)**

- El operador **dynamic_cast permite detectar en tiempo de ejecución si una conversión entre punteros o referencias se ha efectuado correctamente o no**.
- El operador **typeid permite preguntar en tiempo de ejecución si un puntero apunta a un objeto de un tipo determinado**.

Dado un puntero p: **¿es el objeto apuntado por p de tipo T?**
typeid(*p) = = typeid(T); → true si el objeto apuntado por p es de tipo T, false en caso contrario.

dynamic_cast <T *>(p); → (T *) si el objeto apuntado por p es de tipo T, 0 (NULL) si no

Dada una referencia r: **¿es el objeto r de tipo T?**
typeid(r) = = typeid(T); → true si el objeto r es de tipo T, false en caso contrario.

```
try {
  dynamic_cast <T &>(r);  → (T &) si el objeto r es de tipo T, excepción bad_cast si no
} catch (bad_cast) {
  //código a ejecutar si r no es una referencia a T
}
```

Ejemplo:

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

class Punto {
public:
   virtual void pinta(){ cout << "Punto\n"; }
};

class Circulo:public Punto {
public:
   virtual void pinta(){ cout << "Circulo\n"; }
   void otroMetodo(){ cout << "No existe en Punto\n"; }
};

void f1(Punto *p) { //dynamic_cast con punteros
  Circulo *c;
  c=dynamic_cast<Circulo*>(p);
  if (c) {
    cout << "Es un Circulo\n";
    c->otroMetodo();
  }
  else
    cout << "Es un Punto\n";
}

void f2(Punto &p) { //dynamic_cast con referencias
  try {
    Circulo &c=dynamic_cast<Circulo&>(p);
    cout << "Es un Circulo\n";
    c.otroMetodo();
  }
  catch (bad_cast) {
    cout << "Es un Punto\n";
  }
}

int main(int argc, char *argv[]) {
   Punto **array = new Punto *[2];
   array[0] = new Circulo;
   array[1] = new Punto;
   for (int i=0; i<2; i++) {
      f1(array[i]);
      f2(*array[i]);
   }
   system("PAUSE");    return EXIT_SUCCESS;
}
```

Pantalla:

```
Es un Circulo
No existe en Punto
Es un Circulo
No existe en Punto
Es un Punto
Es un Punto
Presione una tecla para continuar . . .
```

Con typeid quedaría así:

```
void f1(Punto *p) { //typeid con punteros
  if (typeid(*p)==typeid(Circulo)) {
    cout << "Es un Circulo\n";          //Circulo *c=(Circulo*)p;
    ((Circulo *)p)->otroMetodo();   //c->otroMetodo();
  }
  else
    cout << "Es un Punto\n";
}

void f2(Punto &p) { //typeid con referencias
  if (typeid(p)==typeid(Circulo)) {
    cout << "Es un Circulo\n";         //Circulo &c=(Circulo &)p;
    ((Circulo &)p).otroMetodo();  //c.otroMetodo();
  }
  else
    cout << "Es un Punto\n";
}
```

Comportamiento polimórfico en funciones amigas

Algunos operadores obligatoriamente hay que implementarlos con funciones no miembros (ej: <<) y otros (ej: = =) se suelen implementar como funciones no miembros en lugar de cómo métodos.
El polimorfismo sólo se puede lograr en aquellos métodos que son declarados virtual, no en funciones no miembros. Podemos hacer que una función no miembro tenga comportamiento polimórfico ¿cómo?
Simplemente haciendo dentro de la función no miembro una llamada interna a un método virtual
Ejemplo:

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

class Date {
  int dia, anio;
  char *mes;
public:
  Date(int d, const char *m, int a) {
   dia=d; anio=a; mes=new char[strlen(m)+1]; strcpy(mes, m);
  }
  ~Date() { delete [] mes; }
  int getDia() const { return dia; }
  const char *getMes() const { return mes; }
  int getAnio() const { return anio; }
  friend ostream& operator<<(ostream &s, const Date &d) {
    s << d.dia << "/" << d.mes << "/" << d.anio;      return s;
  }
};

class Vehiculo {
  char *modelo;
  Date fecha;
 public:
  Vehiculo(char *m, Date &f)
     :fecha(f.getDia(), f.getMes(), f.getAnio()) {
    modelo=new char[strlen(m)+1];    strcpy(modelo, m);
  }
  Vehiculo(const Vehiculo &v):fecha(v.fecha.getDia(),
              v.fecha.getMes(), v.fecha.getAnio()) {
    modelo=new char[strlen(v.modelo)+1];
    strcpy(modelo, v.modelo);
  }
  ~Vehiculo() { delete [] modelo; }
  virtual void ver(ostream &s) const;
  friend ostream& operator<<(ostream &s, const Vehiculo &v) {
    v.ver(s); //llamamos a un método virtual → polimorfismo
    return s;
  }
};

void Vehiculo::ver(ostream &s) const {
  s << "modelo: " << modelo << " fecha: " << fecha.getDia()
 << "/"  << fecha.getMes() << "/" << fecha.getAnio() << endl;
}

class Camion: public Vehiculo {
  char *empresa;
  int tara;
public:
  Camion(char *m, Date &f, char *e, int t):Vehiculo(m,f) {
    tara=t; empresa=new char[strlen(e)+1];  strcpy(empresa, e);
  }
  Camion(const Camion &v): Vehiculo(v) {
    empresa=new char[strlen(v.empresa)+1];
    strcpy(empresa, v.empresa);    tara=v.tara;
  }
  ~Camion() { delete [] empresa; }
  void ver(ostream &s) const;
};

void Camion::ver(ostream &s) const {
  Vehiculo::ver(s);
  s << "empresa: " << empresa << " peso: " << tara << endl;
}

class Coche: public Vehiculo {
  char *amo;
public:
  Coche(char *m, Date &f, char *amo):Vehiculo(m,f) {
    this->amo=new char[strlen(amo)+1];
    strcpy(this->amo, amo);
  }
  Coche(const Coche &v): Vehiculo(v) {
    amo=new char[strlen(v.amo)+1];
    strcpy(amo, v.amo);
  }
  ~Coche() { delete [] amo; }
  void Coche::ver(ostream &s) const {
    Vehiculo::ver(s);
    s << "propietario: " << amo << endl;
  }
};

int main(int argc, char *argv[]) {
  Date f1(10,"abril",1990);
  Camion c("Ford",f1,"Bimbo",1000);
  Coche p("Opel", f1, "juan");
  cout << c;    //(1)
  cout << p;    //(2)
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Pantalla:

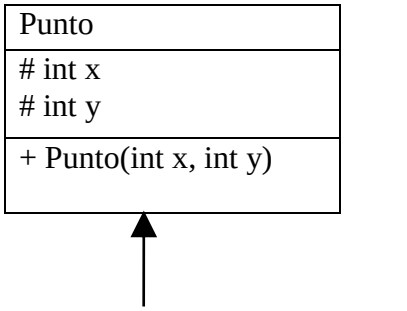
```
modelo: Ford fecha: 10/abril/1990     //(1)
empresa: Bimbo peso: 1000
modelo: Opel fecha: 10/abril/1990     //(2)
propietario: juan
```

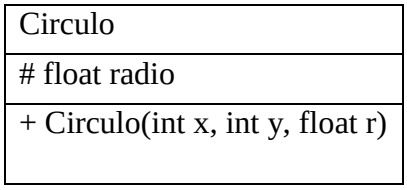
Dentro de función no miembro operator<< llamamos al método virtual ver(s) pasándole el ostream &s

Como v se ha pasado por referencia a operator<< y ver es virtual tenemos una referen cia que invoca un método virtual → comportamiento polimórfico.

Además no es necesario definir la función no miembro operator<< en las clases hijas. Solo hay que sobrecargar en las hijas el método virtual ver(s).
El operator<< es usado por objetos de la clase Vehiculo y por objetos que deriven de Vehiculo.

Instanciación: Constructores/Destructores
Constructores no por defecto

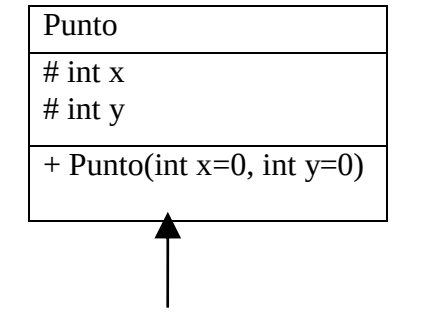


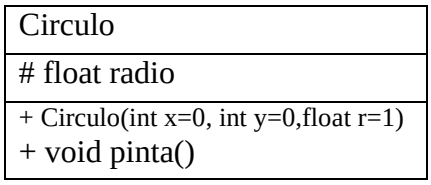


Necesario ya que no hay
un constructor por defecto

```
class Punto{
protected:
        int x;
        int y;
public:
   Punto( int nuevox, int nuevoy ){
      x = nuevox;
      y = nuevoy;
   }
};

class Circulo: public Punto{
protected:
        float radio;
public:
   Circulo( int x, int y, float r ): Punto( x,y ){
      radio = r;
   }
};

int main(int argc, char *argv[])
{
   Circulo c(0,0,1);
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Instanciación: Constructores/Destructores
Argumentos por defecto





No es necesario ya que sí hay un constructor por defecto:
Atención a la instanciación con argumentos

```
class Punto{
protected:
        int x;
        int y;
public:
   Punto( int nuevox = 0, int nuevoy = 0){
      x = nuevox;
      y = nuevoy;
   }
};

class Circulo: public Punto{
protected:
        float radio;
public:
   Circulo( int x = 0, int y = 0, float r = 1 ): Punto( x,y ){
      radio = r;
   }
   void pinta(){
      cout << x <<":"<< y <<" "
          << radio <<"\n";
   }
};

int main(int argc, char *argv[])
{
   Circulo c, c2(1,1);
   c.pinta();
   c2.pinta();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Instanciación: Constructores/Destructores
new y delete

new se utiliza para la instanciación dinámica de objetos:

Circulo *c = new Circulo(); //new Circulo
Circulo *c = new Circulo(10,20,1.5); Después de new se especifica un tipo: new **tipo**

con arrays: new **tipo[num]** ← crea un array de **num** elementos de tipo **tipo**

char *cadena = new char[20]; → char cadena[20]; el estático el compilador libera la memoria
Circulo *array = new Circulo[10]; → Circulo array[10]; **el dinámico la memoria no se libera**
Hay que usar delete para hacerlo

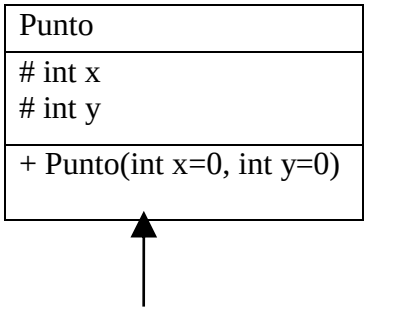
Nota: Para utilizar new con arrays es necesario que exista un constructor por defecto

No confundir con: **Circulo **array = new Circulo*[10];** //array de punteros a Circulo
Si no hay constructor por defecto esto si se permite porque no hemos creado ningún Circulo aun y Ej:
Crear 10 circulos concéntricos de radio 1 a 5: con constructor por defecto (izquierda) y sin él (derecha):

```
Circulo *array = new  Circulo[5];
for(int i=0; i<5; i++) {
    array [i].set(0,0,i+1);
    array [i].pinta( );
}
```

```
Circulo **array = new  Circulo*[5];
for(int i=0; i<5; i++) {
   array [i]=new Circulo (0,0,i+1);
   array [i]->pinta( );
}
```

Recuerde: si se crea un array estático, se llama a los constructores de cada elemento.
Si es un array de referencias, esto será responsabilidad del programador.



Pantalla:

```
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
```

```
class Punto{
protected:
        int x, y;
public:
   Punto( int nuevox = 0, int nuevoy = 0) {
      x = nuevox;       y = nuevoy;
   }
};

class Circulo: public Punto{
protected:
        float radio;
public:
   Circulo( int x = 0, int y = 0, float r = 1 ): Punto( x,y ){
      radio = r;
   }
   void set(int a, int b, float r) { x=a; y=b; radio=r; }
   void pinta(){
      cout << x <<":"<< y <<" "  << radio <<"\n";
   }
};

int main(int argc, char *argv[])
{
   Circulo *c = new Circulo();   // Círculo
   Circulo *c2 = new Circulo[10];  // Array de círculos!!
   c->pinta();
   for (int i=0; i<10;i++){
      c2[i].pinta();
   }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Instanciación: Constructores/Destructores
new y delete

delete se utiliza para la destrucción de objetos creados dinámicamente

Circulo c = new Circulo(); //creo un objeto dinámicamente y hago que c apunte a dicho objeto
delete c; //destruyo el objeto al que apunta c

para arrays:

Circulo *c = new Circulo[10]; ////creo un array de objetos dinámicamente (creo los objetos c[0],…,c[9]
delete [] c; ¡¡ Atención, esto llama al destructor de cada uno de los 10 círculos !!
delete c no falla pero no llama a los destructores correctamente (llama al destructor del primero) !!

delete [] c; es equivalente a poner **delete c[0]; deletec[1]; delete c[2]; …**

Lo que hace delete es llamar al destructor y luego liberar la memoria asignada

- Cada clase tiene un único destructor
- Los destructores no tienen argumentos
- Se llaman como la clase seguida de ~

para arrays de referencias (punteros):

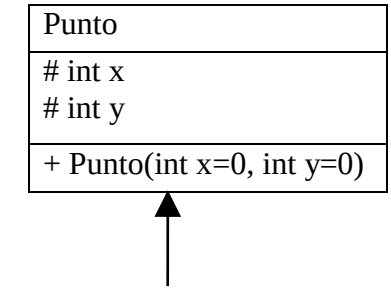
```
Circulo **c = new Circulo*[10]; //crea un array de punteros a Circulo (no he creado ningún objeto)
for(int i=0; i<10; i++)
  c[i]=new Circulo (0,0,i+1);     //creo objetos Circulo en cada iteracion del bucle (aqui si los creo)
…                                 //creo c[0], c[1], … , c[9]
for(int i=0; i<10; i++)
  delete c[i];                    //libero la memoria (destruyo) del Circulo i-esimo
delete [] c;                      //libero (destruyo) el array c de punteros
```

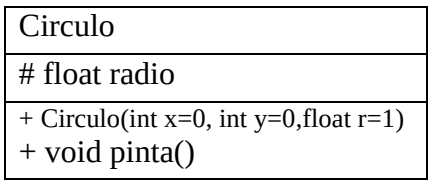
En resumen:

```
clase *puntero = new clase(argumentos);   //crea dinámicamente un objeto de tipo clase
puntero->método(argumentos);              //para invocar un método del objeto
puntero->atributo;                        //para acceder a un atributo del objeto
delete puntero;                           //para destruir el objeto

clase *tabla = new clase[n];              //crea un array de objetos de tipo clase (sin argumentos)
tabla [i].metodo(argumentos);             //para invocar un método del objeto i-ésimo de la tabla
tabla [i].atributo();                     //para acceder a un atributo del objeto i-ésimo de la tabla
delete [] tabla;                          //para destruir el array de objetos (destruye todos )

clase **tabla = new clase*[n];            //crea un array de punteros a objetos clase (no crea objetos)
for(int i=0; i<n; i++)
  tabla[i]=new clase(argumentos);         //crea un objeto de tipo clase (con argumentos)
tabla [i]->metodo(argumentos);            //para invocar un método del objeto i-ésimo de la tabla
tabla [i]->atributo();                    //para acceder a un atributo del objeto i-ésimo de la tabla
for(int i=0; i<n; i++)
  delete tabla[i];                        //destruyo el objeto i-ésimo al que apunta tabla[i]
delete [] tabla;                          //para destruir el array de punteros a objetos
```

Instanciación: Constructores/Destructores
new y delete

Ojo con la reserva dinámica (para trabajar con referencias):





Pantalla:

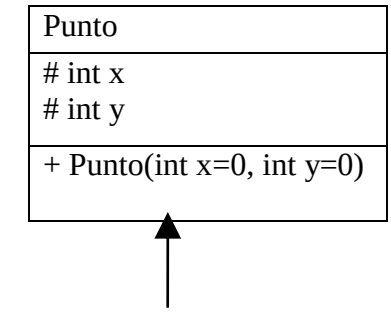
```
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
```

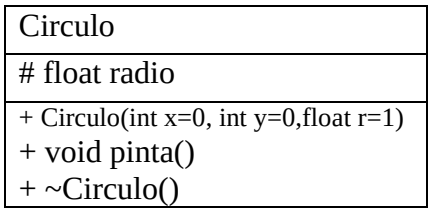
Hemos creado objetos pero al terminar el programa los objetos no se destruyen porque lo que se crea dinámicamente con **new** hay que destruirlo explícitamente con **delete**

```
class Punto{
protected:
        int x;
        int y;
public:
   Punto( int nuevox = 0, int nuevoy = 0) {
      x = nuevox;
      y = nuevoy;
   }
};

class Circulo: public Punto{
protected:
        float radio;
public:
   Circulo( int x = 0, int y = 0, float r = 1 ): Punto( x,y ){
      radio = r;
   }
   void pinta(){
      cout << x <<":"<< y <<" "
         << radio <<"\n";
   }
};

int main(int argc, char *argv[])
{
   Circulo **c = new Circulo*[10];   // Array de referencias

for (int i=0; i<10;i++){
      c[i] = new Circulo(); //new Circulo
   }

   for (int i=0; i<10;i++){
      c[i]->pinta();
   }

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Instanciación: Constructores/Destructores
new y delete

Para realizar correctamente el ejemplo anterior:





Hemos creado objetos con **new** y antes de terminar el programa hemos destruido los objetos explícitamente con **delete**

Pantalla:

```
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
0:0 1
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
DESTRUCTOR
```

```
class Punto {
protected:
        int x;
        int y;
public:
   Punto( int nuevox = 0, int nuevoy = 0) {
      x = nuevox;
      y = nuevoy;
   }
};

class Circulo: public Punto {
protected:
        float radio;
public:
   Circulo( int x = 0, int y = 0, float r = 1 ): Punto( x,y ){
      radio = r;
   }
   ~Circulo(){
      cout << "DESTRUCTOR\n";
   }
   void pinta(){
      cout << x <<":"<< y <<" "
         << radio <<"\n";
   }
};

int main(int argc, char *argv[])
{
   Circulo **c = new Circulo*[10];   // Array de referencias

   for (int i=0; i<10;i++){
      c[i] = new Circulo(); //new Circulo
   }

   for (int i=0; i<10;i++){
      c[i]->pinta();
   }

   for (int i=0; i<10;i++){
      delete c[i]; // porque es un array de referencias…
   }

   delete [] c;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Instanciación: Constructores/Destructores
Destructores polimórficos

Los destructores polimórficos tienen sentido. Los constructores, no.

Punteros y referencias: Métodos virtuales

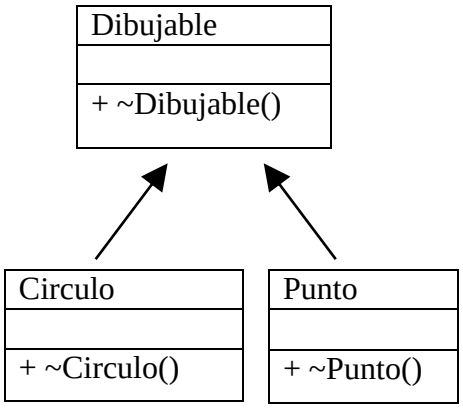
Un puntero o una referencia de una clase Base puede apuntar a un objeto de su propia clase o a cualquier objeto **de una clase derivada** de ella (hija, nieta, etc.).

- El tipo del puntero o de la referencia determina que métodos y a que atributos puede acceder.
- El método que se ejecuta es el de la clase de la que es el puntero o la referencia

Para hacer que el método que se ejecuta por un puntero o una referencia no **sea** el de la clase del puntero o la referencia, sino **el del objeto al que apunta el puntero o la referencia hay que declarar dicho método como virtual.**

Cuando un objeto de una clase derivada se crea dinámicamente y está apuntado por un puntero de la clase base, al aplicar el operador **delete** al puntero de la clase base se ejecuta el destructor de la clase base, a menos que hayamos declarado el destructor como **virtual.**

Si el objeto de la clase derivada tuviera un atributo propio dinámico (que no tuviera la clase base) y el destructor no fuera **virtual**, la memoria no se liberaría al ejecutarse directamente el destructor de la Base




```
class Dibujable{
public:
   ~Dibujable(){cout << "DESTRUCTOR: Dibujable\n";}
};

class Punto: public Dibujable{
public:
   ~Punto(){cout << "DESTRUCTOR: Punto\n";}
};

class Circulo: public Dibujable{
public:
   ~Circulo(){cout << "DESTRUCTOR: Circulo\n";}
};

int main(int argc, char *argv[])
{
   Dibujable *d;
   Punto *p = new Punto(); //new Punto
   Circulo *c = new Circulo(); //new Circulo
   d = p;  //el puntero Dibujable apunta a Punto
   delete d; //se ejecuta el destructor de Dibujable
   d = c;  //el puntero Dibujable apunta a Circulo
   delete d; //se ejecuta el destructor de Dibujable
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Comportamiento estático
ERROR!!!!

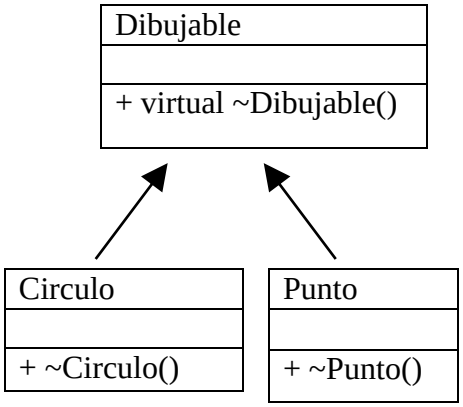
Salida por pantalla:

```
DESTRUCTOR: Dibujable
DESTRUCTOR: Dibujable
```

Instanciación: Constructores/Destructores
Destructores polimórficos

Los destructores polimórficos tienen sentido. Los constructores, no.

Solucion:
Siempre que haya memoria dinámica se debe declarar el destructor de la clase base como virtual, para evitar el error anterior y hacer que el destructor que se ejecute sea el de la clase al que apunta el puntero y no el de la clase del puntero.




```
class Dibujable{
public:
   virtual ~Dibujable(){cout << "DESTRUCTOR: Dibujable\n";}
};

class Punto: public Dibujable{
public:
   ~Punto(){cout << "DESTRUCTOR: Punto\n";}
};

class Circulo: public Dibujable{
public:
   ~Circulo(){cout << "DESTRUCTOR: Circulo\n";}
};

int main(int argc, char *argv[])
{
   Dibujable *d;
   Punto *p = new Punto(); //new Punto
   Circulo *c = new Circulo();
   d = p;  //el puntero Dibujable apunta a Punto
   delete d; //se ejecuta el destructor de Punto, que llama después
             // automáticamente al destructor de Dibujable
   d = c;  //el puntero Dibujable apunta a Circulo
   delete d; //se ejecuta el destructor de Circulo, que llama después
             // automáticamente al destructor de Dibujable

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Comportamiento dinámico
CORRECTO!!!

Salida por pantalla:

```
DESTRUCTOR: Punto
DESTRUCTOR: Dibujable
DESTRUCTOR: Circulo
DESTRUCTOR: Dibujable
```

Instanciación: Constructores/Destructores
Constructores polimórficos

Los destructores polimórficos tienen sentido. Los constructores, no.
Aunque no se pueden crear constructores virtuales, se pueden simular así:

1. Implementar en las clases (Base y Derivada) métodos virtuales que creen dinámicamente objetos de dicha clase y devuelvan punteros a dicho objeto.
2. Crear un objeto de la clase Base y de la Derivada y:
   a. Crear un puntero de la clase Base y hacer que el puntero apunte al objeto Base o Derivada e invocar los métodos virtuales que crean dinámicamente objetos, y/o
   b. Crear un método o función no miembro que tenga un parámetro puntero o **referencia** a la clase base, que invoque los métodos virtuales que crea dinámicamente objetos

```
class Base {
public:
   Base( ) {cout << "BASE\n";}
   Base(const Base &b ) {cout << "COPIA BASE\n";}
   virtual ~Base() {cout << "~BASE\n";}
   virtual Base *nuevo ( ) const {  return new Base; } //invoca al constructor
   virtual Base *clonar ( ) const { return new Base(*this); } //invoca al
};                                                              // constructor copia

class Hija: public Base {
public:
   Hija ( ) {cout << "HIJA \n";}
   Hija (const Hija &b ):Base(b) {cout << "COPIA HIJA\n";}
   ~ Hija (){cout << "~HIJA \n";}
   Hija *nuevo ( ) const { return new Hija; } //invoca al constructor
   Hija *clonar ( ) const { return new Hija (*this); } //invoca al constructor copia
};

Base *crearObjeto(Base *p) { return p->nuevo( ); }
Base *clonarObjeto(const Base &r) { return r.clonar( ); }

int main(int argc, char *argv[]) {
   Base b, *pb, *T[5];
   Hija d, *pd;
   cout << "---\n";
   pb=&b; //hago que el puntero Base apunte a un objeto Base
   Base *p1 = pb->clonar(); //crea (clona) el objeto Base
   T[0] = pb->nuevo(); //crea un objeto Base nuevo
   cout << "---\n";
   pb=&d; //hago que el puntero Base apunte a un objeto Hija
   T[1] = pb->clonar(); //crea (clona) el objeto Hija
   T[2] = pb->nuevo(); //crea un objeto Hija nuevo
   cout << "---\n";
   T[3] = crearObjeto(&b); //crea un objeto Base
   T[4] = crearObjeto(&d); //crea un objeto Hija
   cout << "---\n";
   T[5] = clonarObjeto(b); //crea (clona) un objeto Base
   T[6] = clonarObjeto(d); //crea (clona) un objeto Hija
   cout << "---\n";
   delete p1;
   for(int i=0; i<6; i++)
     delete T[i];
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Comportamiento dinámico

Salida por pantalla:

```
BASE                          //b
BASE  HIJA                    //d
---
COPIA BASE                    //p1
BASE                          //T[0]
---
COPIA BASE COPIA HIJA         //T[1]
BASE  HIJA                    //T[2]
---
BASE                          //T[3]
BASE  HIJA                    //T[4]
---
COPIA BASE                    //T[5]
COPIA BASE COPIA HIJA         //T[6]
---
~BASE                         //p1
~BASE                         //T[0]
~HIJA ~BASE                   //T[1]
~HIJA ~BASE                   //T[2]
~BASE                         //T[3]
~HIJA ~BASE                   //T[4]
~BASE                         //T[5]

~HIJA ~BASE                   //d
~BASE                         //b
```

Ejercicio:
Crear una clase **Base** abstracta con un atributo privado **int a**, y un método **ver( )** que muestra por pantalla el valor de a.
Crea 3 clases derivadas Hijo1, Hijo2 y Hijo3:

- Hijo1 tiene un atributo adicional **int b** y un método **calcular( )** que devuelve la suma de a y b.
- Hijo2 solo tiene sobrecargado el operador ++ de forma que permite incrementar en una unidad el atributo que hereda.
- Hijo3 tiene un método saludo( ) que muestra pon pantalla el mensaje "Hola\n".

Todas las clases deben tener un único constructor con tantos parámetros (sin valores por defecto) como atributos tengan (heredados o no).
Crea un programa que guarde en un array 10 elementos de tipo Hijo1, Hijo2 y Hijo3 creados aleatoriamente y los muestre por pantalla indicando de qué tipo es cada elemento mostrado. A continuación debe guardar en otro array una copia exacta del primero y mostrarlo por pantalla. Finalmente debe guardar en otro array una copia de los elementos del array original que son de tipo Hijo1 (y mostrarlo por pantalla) e incrementar en 2 unidades (utilizando el operador ++) los elementos del array original que son de tipo Hijo2.

Solución: (**sin constructores virtuales**)

```
#include <cstdlib>
#include <iostream>
#include <typeinfo>
#include <time.h>
using namespace std;

class Base {
  int a;
public:
  Base(int a) { Base::a=a; }
  virtual void ver()=0; //virtual puro clase abstracta
  int get() { return a; }
  void set(int a) { this->a=a; }
};

class Hijo1:public Base {
  int b;
public:
  Hijo1(int x, int y):Base(x), b(y) { }
  void ver() { cout << get() << "," << b << endl; }
  int calcular() { return get()+b; }
};

class Hijo2:public Base {
public:
  Hijo2(int a):Base(a) {  }
  Hijo2 operator++(); //++obj
  Hijo2 operator++(int i); //obj++
  void ver() { cout << get() << endl; }
};

Hijo2 Hijo2::operator++() {
  set(get()+1);
  return *this;
}

Hijo2 Hijo2::operator++(int i) {
  Hijo2 copia(*this); //constructor copia
  set(get()+1);
  return copia;
}

class Hijo3:public Base {
public:
  Hijo3(int a):Base(a) {  }
  void saludo() { cout << "Hola\n"; }
  void ver() { cout << get() << endl; }
};
```

```
int main(int argc, char *argv[]) {
    Base *T[10], *clonT[10];
    srand(time(0)); //inicializa la semilla de numeros aleatorios
    for(int i=0; i<10; i++) {
      int n=rand()%3;
      if (n==0) {// si el numero aleatorio es 0
        cout << "Hijo1: ";
        T[i]=new Hijo1(rand()%10+1,rand()%5);
      }
      else if (n==1) {
        cout << "Hijo2: ";
        T[i]=new Hijo2(rand()%5); //aleatorio entre 0 y 4
      }
      else {
        cout << "Hijo3: ";
        T[i]=new Hijo3(rand()%5); //aleatorio entre 0 y 4
      }
      T[i]->ver();
    }
    cout << "---clon---\n";
    for(int i=0; i<10; i++) {
      if (Hijo1 *h = dynamic_cast<Hijo1*>(T[i]))
        clonT[i]=new Hijo1(*h); //constructor de copia
      else if (Hijo2 *h = dynamic_cast<Hijo2*>(T[i]))
        clonT[i]=new Hijo2(*h); //constructor de copia
      else if (Hijo3 *h = dynamic_cast<Hijo3*>(T[i]))
        clonT[i]=new Hijo3(*h); //constructor de copia
      clonT[i]->ver();
    }
    cout << "---\n";
    Hijo1 **H1=new Hijo1 *[10];
    int n=0;
    for(int i=0; i<10; i++) {
      if (Hijo1 *h = dynamic_cast<Hijo1*>(T[i])) {
        H1[n]=new Hijo1(*h); //constructor de copia
        H1[n]->ver();
        n++;
      }
      if (typeid(*T[i])==typeid(Hijo2)) {
          ((Hijo2 *)T[i])->operator++();
          (*((Hijo2 *)T[i]))++;
      }
    }
    cout << "---\n";
    for(int i=0; i<10; i++)
      T[i]->ver();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Pantalla:

```
Hijo2: 4  Hijo2: 3  Hijo2: 4  Hijo1: 2,3  Hijo2: 4  Hijo3: 1  Hijo1: 9,0  Hijo3: 1  Hijo3: 0  Hijo3: 2
---clon---
4  3  4  2,3  4  1  9,0  1  0  2
---
2,3  9,0
---
6  5  6  2,3  6  1  9,0  1  0  2
```

Solución: (**con constructores virtuales**)

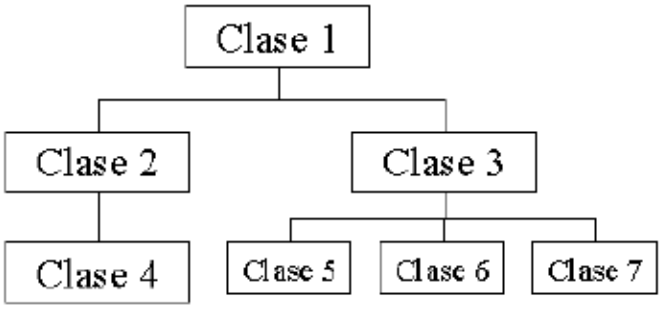
```
#include <cstdlib>
#include <iostream>
#include <typeinfo>
#include <time.h>
using namespace std;

class Base {
  int a;
public:
  Base(int a) { Base::a=a; }
  virtual void ver()=0; //virtual puro clase abstracta
  int get() { return a; }
  void set(int a) { this->a=a; }
  virtual Base *clonar() const =0; //virtual puro
};

class Hijo1:public Base {
  int b;
public:
  Hijo1(int x, int y):Base(x), b(y) { }
  void ver() { cout << get() << "," << b << endl; }
  int calcular() { return get()+b; }
  Hijo1 *clonar() const { return new Hijo1(*this); }
//invoca constructor copia
};

class Hijo2:public Base {
public:
  Hijo2(int a):Base(a) {  }
  Hijo2 operator++(); //++obj
  Hijo2 operator++(int i); //obj++
  void ver() { cout << get() << endl; }
  Hijo2 *clonar() const { return new Hijo2(*this); }
//invoca constructor copia
};

Hijo2 Hijo2::operator++() {
  set(get()+1);
  return *this;
}

Hijo2 Hijo2::operator++(int i) {
  Hijo2 copia(*this); //constructor copia
  set(get()+1);
  return copia;
}

class Hijo3:public Base {
public:
  Hijo3(int a):Base(a) {  }
  void saludo() { cout << "Hola\n"; }
  void ver() { cout << get() << endl; }
  Hijo3 *clonar() const { return new Hijo3(*this); }
//invoca constructor copia
};
```

```
int main(int argc, char *argv[]) {
   Base *T[10], *clonT[10];
   srand(time(0)); //inicializa la semilla de numeros aleatorios
   for(int i=0; i<10; i++) {
    int n=rand()%3;
    if (n==0) {// si el numero aleatorio es 0
      cout << "Hijo1: ";
      T[i]=new Hijo1(rand()%10+1,rand()%5);
    }
    else if (n==1) {
      cout << "Hijo2: ";
      T[i]=new Hijo2(rand()%5); //aleatorio entre 0 y 4
    }
    else {
      cout << "Hijo3: ";
      T[i]=new Hijo3(rand()%5); //aleatorio entre 0 y 4
    }
    T[i]->ver();
   }
   cout << "---clon---\n";
   for(int i=0; i<10; i++) {
    clonT[i]=T[i]->clonar();
    clonT[i]->ver();
   }
   cout << "---\n";
   Hijo1 **H1=new Hijo1 *[10];
   int n=0;
   for(int i=0; i<10; i++) {
    if (Hijo1 *h = dynamic_cast<Hijo1*>(T[i])) {
      H1[n]=new Hijo1(*h); //constructor de copia
      H1[n]->ver();
      n++;
    }
    if (typeid(*T[i])==typeid(Hijo2)) {
       ((Hijo2 *)T[i])->operator++();
       (*((Hijo2 *)T[i]))++;
    }
   }
   cout << "---\n";
   for(int i=0; i<10; i++)
    T[i]->ver();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

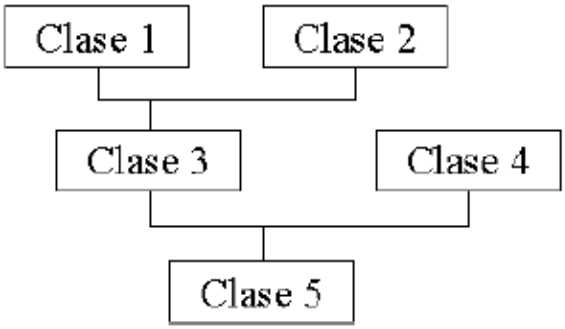
El resultado en pantalla es el mismo que el anterior

Herencia simple y herencia múltiple:

Una clase derivada puede tener una o más clases bases directas
Herencia simple: Cuando una clase derivada tiene una sola clase base directa
Herencia múltiple: Cuando una clase derivada tiene más de una clase base directa



Herencia Simple: Todas las clases derivadas tienen una única clase base



Herencia Múltiple: Las clases derivadas tienen varias clases base

Herencia múltiple:
- La clase derivada hereda todos los atributos y métodos de las clases bases (excepto constructores, destructores y operator=), más los atributos y/o métodos adicionales que ella misma cree.
- Al igual que en la herencia simple, aunque herede solo tiene accedo directo a los **protected** y **public,** pero no tiene acceso directo a los **private** de las clases bases.

Constructores en clases derivadas con herencia múltiple

Cuando se crea un objeto de una clase derivada, se invoca automáticamente su constructor, y éste invoca a los constructores por defecto de sus clase bases (a menos que indiquemos en los inicializadores otros constructores distintos), que a su vez invocan a los constructores de sus clases bases (si la tuviera) y así sucesivamente. Primero se ejecutan los constructores de las clases bases de arriba a abajo en la jerarquía de clases y finalmente el de la clase derivada.

```
Hija::Hija(parametros): otros {
  codigo_adicional
}
```

equivale a poner

```
Hija::Hija(parametros): Padre( ), Madre( ), otros {
  codigo_adicional;
}
```

Si en la clase derivada no hay ningún constructor, el compilador crea uno por defecto de oficio, que automáticamente llama a los constructores por defecto (ya sea de oficio o no) de sus clases bases.

Si en una clase derivada no hay constructor (ni de copia) porque no hace falta, es como si hubiera

```
Hija:: Hija ( ) {
  //nada
}
```

ó

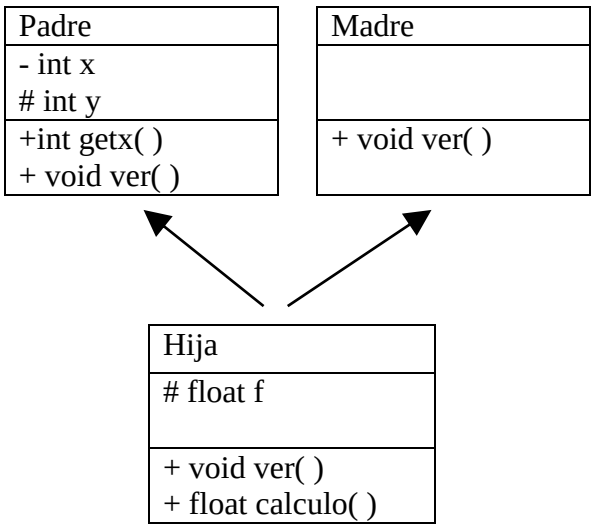
```
Hija:: Hija ( ): Padre( ), Madre( ) {
  //nada
}
```

Destructores en clases derivadas con herencia múltiple

El destructor de una clase derivada invoca automáticamente a los destructores de sus clases bases. El destructor derivada sólo debe preocuparse de liberar la memoria (si tuviera que hacerlo) de los atributos dinámicos propios de la derivada (los destructores de las clases bases se ocupan de los suyos propios).

Los destructores se llaman en orden inverso a los constructores. Primero se ejecuta el cuerpo del destructor de la derivada, después son llamados los destructores de sus atributos objetos miembros (si los tuviera) y por último se ejecuta los destructores de sus clases bases, que a su vez actuarían igual.

Herencia múltiple: llamada a métodos con el mismo nombre en los antecesores:



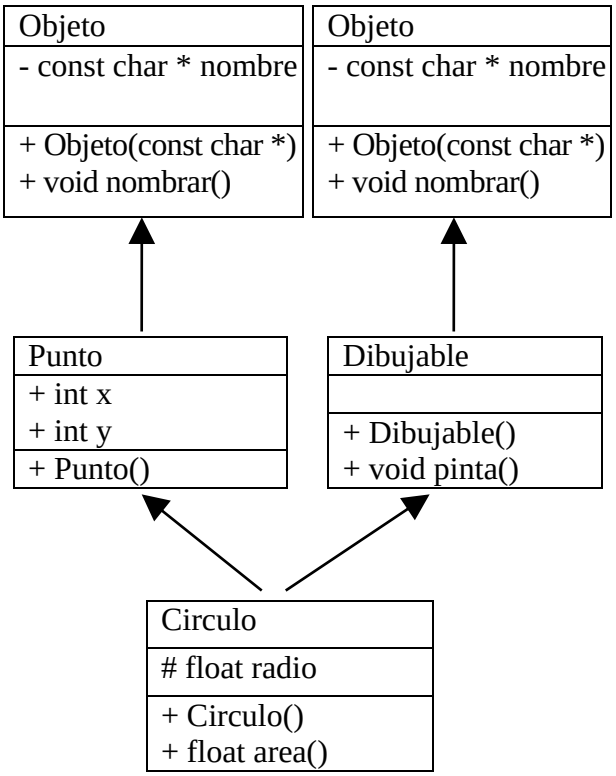
Pantalla:

```
Hija de mi Padre
Padre
Madre
Hija de mi Padre
```

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

class Padre {
   int x;
protected:
   int y;
public:
   int getx( ) { return x; }
   void ver(){  cout << "Padre\n";  }
};

class Madre {
public:
   void ver() {  cout << "Madre\n";  }
};

class Hija: public Padre, public Madre {
protected:
   float f;
public:
   void ver() {
      cout << "Hija de mi ";
      Padre::ver();
   };
   float calculo() {
      return getx( )*y*f;
   }
};

int main(int argc, char *argv[])
{
   Hija h;
   h.ver();
   h.Padre::ver();
   h.Madre::ver();
   h.ver();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Herencia múltiple: llamadas a constructores: caso del antecesor común

Una clase base puede heredarse varias veces indirectamente



Problema:

Si la clase Punto y Dibujable heredan de la clase Objeto y la clase Circulo hereda de Punto y Dibujable:

Los miembros de Objeto estarán duplicados en Circulo!!!!

Habrá 2 copias de **nombre** independientes!!!

Habrá problemas de ambigüedad!!!

Pantalla:

```
reserva Dibujable      //c
reserva Punto
Punto                  //p->nombrar();
Dibujable              //d->nombrar();
(1,2)Punto             //p->pinta();
Dibujable              //d->pinta();
Circulo-5              //c.pinta();
Punto
Dibujable
(1,2)Punto             //c.Punto::pinta();
Dibujable              //c.Dibujable::pinta();
```

```
class Objeto {
   const char * nombre;
public:
   Objeto( const char *nombre ) {
      this->nombre = strdup(nombre); // Reserva memoria!!
      cout << "reserva " << nombre << endl;
   }
   void nombrar( ) {  cout << this->nombre << endl; }
};

class Punto: public Objeto {
public:
   int x, y;
   Punto( ): Objeto("Punto" ), x(0) { y=0; }
   Punto( int x, int y, char *c="Punto");
   void pinta( )
     {  cout << "(" << x << "," << y << ")";    nombrar();  }
};

Punto::Punto( int x, int y, char *c):Objeto(c)
   {   this->x=x;    Punto::y=y;   }

class Dibujable: public Objeto {
public:
   Dibujable( ): Objeto("Dibujable" ) {  }
   void pinta( ) { nombrar(); }
};

class Circulo: public Dibujable, public Punto {
protected:
   float radio;
public:
   Circulo( ) { radio=0; }
// Circulo( ): Dibujable( ), Punto( ), radio(0) { }
   Circulo(int x, int y, float r, char *c)
      :Dibujable( ), Punto(x, y, c) { radio = r; }
   void pinta( ) {
      cout << "Circulo-" << radio << endl;
      //nombrar();  //ERROR ambigüedad Qué nombrar?
      Punto::nombrar();
      Dibujable::nombrar();
   };
   float area( ) { return 2*3.14*radio; }
};

int main(int argc, char *argv[]) {
   Circulo c(1,2,5, "Punto");
   Objeto *o;
   //o = &c; //ERROR Objeto es ambigüo
   //o->nombrar(); //¿Qué nombrar ejecuta?
   Punto *p = &c;
   Dibujable *d = &c;
   p->nombrar();    d->nombrar();
   p->pinta();      d->pinta();
   c.pinta();
   c.Punto::pinta();
   c.Dibujable::pinta();
   //c.Objeto::nombrar(); //ERROR ambigüedad
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Herencia múltiple: llamadas a constructores: caso del antecesor común

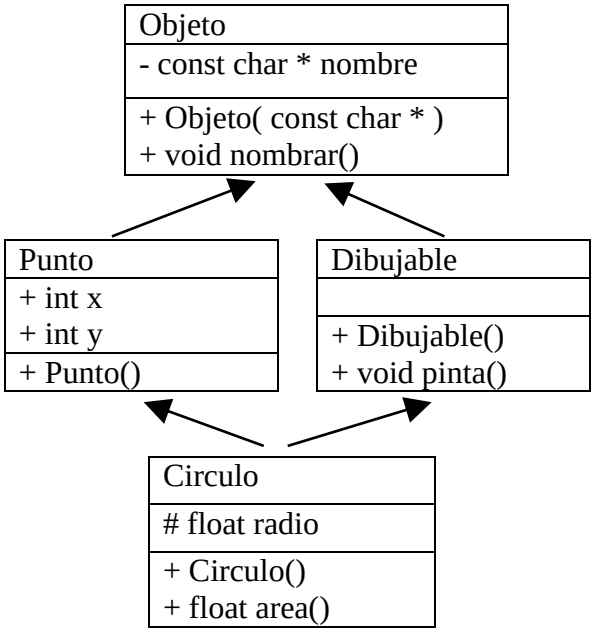
Para evitar que una clase base puede heredarse varias veces indirectamente hay que hacer que la herencia sea virtual:

Al declarar los constructores, hay que declarar la clase común como **virtual en las clases que lo heredan.**

MUY IMPORTANTE!!!!!
El constructor de la clase que hereda de clases con herencia virtual debe invocar (además de a los constructores de sus clases padres) **al constructor de la clase virtual heredada por sus padres.**

Si el constructor no lo hace explícitamente **el compilador invoca los constructores por defecto** (los que falten, sean virtuales o no)**,** de forma que si no hay por defecto -> ERROR

Cuando se construye un objeto de la clase que lo hereda más de una vez, **el compilador invoca 1º los constructores de las clases bases virtuales** que tenga **y luego los constructores de las otras clases bases** no virtuales, **independientemente del orden** en el que estén **en los inicializadores**.



Pantalla:

```
reserva Circulo      //c
Circulo              //o->nombrar();
Circulo              //p->nombrar();
Circulo              //d->nombrar();
(1,2)Circulo         //p->pinta();
Circulo              //d->pinta();
Circulo-5            //c.pinta();
Circulo
Circulo
Circulo
(1,2)Circulo         //c.Punto::pinta();
Circulo              //c.Dibujable::pinta();
Circulo              //c.Objeto::nombrar();
```

```
class Objeto {
   const char * nombre;
public:
   Objeto( const char *nombre ) {
      this->nombre = strdup(nombre); // Reserva memoria!!
      cout << "reserva " << nombre << endl;
   }
   void nombrar( ) {  cout << this->nombre << endl; }
};

class Punto: virtual public Objeto {
public:
   int x, y;
   Punto( ): Objeto("Punto" ), x(0) { y=0; }
   Punto( int x, int y, char *c="Punto");
   void pinta( )
     {  cout << "(" << x << "," << y << ")";    nombrar(); }
};

Punto::Punto( int x, int y, char *c):Objeto(c)
   {   this->x=x;    Punto::y=y;   }

class Dibujable: virtual public Objeto {
public:
   Dibujable( ): Objeto("Dibujable" ) {  }
   void pinta( ) { nombrar(); }
};

class Circulo: public Dibujable, public Punto {
protected:
   float radio;
public:
   //Circulo( ) { radio=0; } //ERROR Objeto() no existe
   //Circulo( ): Objeto(), Dibujable( ), Punto( ), radio(0) { }
   Circulo( ):Objeto("Circulo"),Dibujable(),Punto(){radio=0;}
   Circulo(int x, int y, float r, char *c)
    :Objeto("Circulo"), Dibujable(), Punto(x, y, c) {radio=r;}
   void pinta( ) {
      cout << "Circulo-" << radio << endl;
      nombrar();  //YA NO HAY ambigüedad
      Punto::nombrar();
      Dibujable::nombrar();
   };
   float area( ) { return 2*3.14*radio; }
};

int main(int argc, char *argv[]) {
   Circulo c(1,2,5, "Punto");
   Objeto *o;
   o = &c; //ya Objeto no es ambigüo
   o->nombrar(); //ya hay un solo nombrar
   Punto *p = &c;
   Dibujable *d = &c;
   p->nombrar();    d->nombrar();
   p->pinta();      d->pinta();
   c.pinta();
   c.Punto::pinta();
   c.Dibujable::pinta();
   c.Objeto::nombrar(); //ya no hay ambigüedad
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Miembros estáticos
Cada objeto de una clase tiene su propia copia de cada una de las variables miembros de esa clase. Dichos miembros no existen hasta que no creamos un objeto.

Si queremos definir un miembro que sea común para todos los objetos de la clase (que sólo haya una copia para todos los objetos), de modo que todos compartan el mismo valor, debemos declarar dicho miembro estático. Ej: sueldo base común a todos los empleados, interés ofrecido por un banco sea igual para todas las cuentas de un mismo tipo, un contador de objetos creados para una determinada clase.

Los miembros estáticos (**public** o **private)** pueden ser tanto atributos como métodos.

- Se suele acceder a ellos usando la notación

  `clase::miembro_estatico` (recomendable, posible siempre, exista o no objetos clase)

  Aunque también se puede utilizar la notación:

  `objeto.miembro_estatico` (más confuso, sólo posible cuando ya existe objeto)

  Se recomienda la 1ª forma ya que la 2ª es válida también para atributos no estáticos.

Atributos estáticos (static):

- Son compartidos por todos los objetos de una clase. No aumentan el tamaño de la clase ya que su almacenamiento es global.
- Se utilizan para almacenar información común (que es compartida) a todos los objetos de la clase.
- Existen aunque no se haya creado ningún objeto de dicha clase.
- Son preferibles a las variables globales ya que conservan la encapsulación: son como variables globales pero con ámbito y acceso restringido a nivel de clase (público, protegido o privado).
- Pueden ser tomados como valores por defecto en argumentos de métodos (estáticos o no)
- No se pueden inicializar en un constructor (porque se inicializarían muchas veces, cada vez que creáramos un objeto). Necesitan ser definidos e inicializados fuera del cuerpo de la clase:

  ```
  tipo clase::atributo_estatico = valor_inicial;
  ```

- Pueden ser un objeto de la clase que los contiene

Métodos estáticos (static):

- Se usan generalmente para actuar globalmente sobre todos los objetos de una clase.
- No tienen el argumento implícito **this** que tienen los métodos normales, por lo que sólo pueden acceder y llamar a los atributos y métodos **static** que haya en la clase (aunque puede llamar a cualquier miembro de los objetos pasados por parámetro).
- No pueden ser métodos **const** (no tiene sentido, al ser funciones que no actúan sobre ningún objeto concreto de la clase)
- No permiten sobrecarga de operadores

En resumen, los atributos y métodos **static** resultan útiles en el caso de que se quieran establecer variables y métodos comunes a todos los objetos de una clase. Los métodos **static** pueden recibir como argumentos explícitos objetos de su propia clase (o de cualquier otra), aunque no como argumento implícito. Esto implica que cuando se desee que un método actúe sobre dos objetos de una misma clase, las funciones **static** son una alternativa a las funciones **friend** para conseguir simetría en la forma de tratar a los dos objetos de la clase (que ambos pasen como argumentos explícitos)

Static: atributos y métodos de clase:

Tenemos una clase con una variable común a toda la clase: cuenta de instancias.

Métodos **static**, en referencia a su clase, sólo pueden acceder a variables y métodos de clase **static**

Objeto
+ static int cuenta = 0 - const char *nombre
+ Objeto(const char *nuevoNombre = "") +~Objeto() + static void muestraCuenta() + void muestraNombre()

Ojo: static se pone en el .hpp pero no en el .cpp

Inicialización de variables de clase static (no constantes).

Objeto *array = new Objeto[5]; crea array dinámico de 5 elementos tipo Objeto.

Para poder crear arrays de objetos es necesario que la clase tenga un constructor por defecto (sin argumentos o que pueda invocarse sin argumentos).

Si observamos el constructor es por defecto ya que el único parámetro que tiene es por defecto
Objeto(const char *nuevoNombre = "");

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

class Objeto{
private:
   const char * nombre;
public:
   static int cuenta;
//constructor con parámetro por defecto igual a ""
   Objeto( const char *nuevoNombre = "" ) {
      nombre = strdup(nuevoNombre);
      cuenta++;
   }
    virtual ~Objeto( ){ delete [ ] nombre; cuenta--; }
    static void muestraCuenta( ){
      //cout << nombre; No permitido
      cout << cuenta;
   }
   void ver( ) { cout << cuenta <<":"<< nombre<<"\n"; }
};

int Objeto::cuenta=0;

int main(int argc, char *argv[]) {
   cout << Objeto::cuenta <<"\n";
   Objeto *array = new Objeto[5]; //crea 5 objetos
   Objeto obj("Pieza");
   cout << array[3].cuenta <<"\n";
   obj.muestraCuenta();
   cout << "\n";
   delete [ ] array; //probar con "delete array;"…
   cout << obj.cuenta <<"\n";
   Objeto::muestraCuenta();
   cout << "\n";
   obj.ver();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

```
0
6
6
1
1
1:Pieza
```

No es buena técnica definir atributos estáticos públicos, es mejor definirlos privados y proporcionar un método estático público para acceder a ellos y modificarlos (en el método estático podremos controlar la forma en la que permitimos hacerlo)

```
class Objeto{
   const char * nombre;
   static int cuenta;
public:
   Objeto( const char *nuevoNombre = "" ) {
      nombre = strdup(nuevoNombre);
      cuenta++;
   }
   virtual ~Objeto() { delete [ ] nombre; cuenta--; }
   static void muestraCuenta() { cout << cuenta; }
   void ver() { cout << cuenta <<":"<< nombre<<"\n"; }
   static int getcuenta() { return cuenta; }
   static int setcuenta(int i) { cuenta=i; }
};

int Objeto::cuenta=0;
```

```
int main(int argc, char *argv[]) {
   cout << Objeto::getcuenta() <<"\n";
   Objeto *array = new Objeto[5]; //crea 5 objetos
   Objeto obj("Pieza");
   cout << array[3].getcuenta() <<"\n";
   obj.muestraCuenta();
   cout << "\n";
   delete [ ] array; //probar con "delete array;"…
   cout << obj.getcuenta() <<"\n";
   Objeto::muestraCuenta();
   cout << "\n";
   obj.ver();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Que los atributos estáticos puedan ser objetos de la clase que los contiene no significa que podemos tener datos recursivos, sino que **podemos definir un objeto común para todos los objetos que instancien la clase**

```
#include <iostream>
#include <cmath>
using namespace std;
class Punto {
  int x, y;
public:
  static Punto o; //origen comun para todos los puntos
  static int n;
  Punto(int a=0, int b=0) { x = a; y = b; n++; }
  void set(int a, int b) { x = a; y = b; }
  friend float distancia(const Punto &, const Punto &);
  static float distancia(const Punto &, const Punto &);
  void dibujar() { cout << "(" << x+o.x << "," << y+o.y << ")\n"; }
};//dibujar muestra el punto tomando o como origen de coordenadas

Punto Punto::o; // Punto Punto::o(0,0);
int Punto::n = 0;
//forma de definir e inicializar los miembros estaticos
//fuera de la clase! Estas sentencias son obligatorias!!!
//indicar el tipo (Punto y int) es imprescindible

float distancia(const Punto &a, const Punto &b) {
 return sqrt(pow((b.x-a.x),2.0)+pow((b.y-a.y),2.0));
}

float Punto::distancia(const Punto &a, const Punto &b) {
 return sqrt(pow((b.x-a.x),2.0)+pow((b.y-a.y),2.0));
}

int main(int argc, char *argv[]) {
  cout << "Hay " << Punto::n << " puntos creados\n";
  Punto::o.dibujar();
  Punto::n--; //para que no cuente el punto origen
  Punto p1(1,1), p2(5,4);
  cout << "Hay " << Punto::n << " puntos creados\n";
  p1.dibujar();
  p2.dibujar();
  cout << distancia(p1,p2) << endl;
//cambio el origen de coordenadas a 2,3
  p1.o.set(2,3); //Punto::o.set(2,3); p2.o.set(2,3);
  p1.dibujar();
  p2.dibujar();
  cout << p1.distancia(p1,p2) << endl; //la distancia no cambia
  //Punto::distancia(p1,p2) es lo mismo que p2.distancia(p1,p2) y que p1.distancia(p1,p2)
//cambio el origen de coordenadas al indicado en p2
  p1.o=p2; //Punto::o=p2; p2.o=p2;
  p1.dibujar();
  p2.dibujar();
  cout << distancia(p1,p2) << endl; //la distancia no cambia
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

```
Hay 1 puntos creados
(0,0)
Hay 2 puntos creados
(1,1)
(5,4)
5
(3,4)
(7,7)
5
(6,5)
(10,8)
5
```

Como podemos observar, la distancia la podemos calcular mediante una función miembro estática o mediante una función no miembro (amiga).

La forma en la que se llama en el main() difiere una de otra. Para invocar la función miembro estática hay que utilizar la notación `clase::funcion()` o `objeto.funcion()` mientras que para invocar la función amiga simplemente utilizamos la notación `funcion()`.

No es buena técnica definir atributos estáticos públicos, es mejor definirlos privados y proporcionar un método estático público para acceder a ellos y modificarlos (en el método estático podremos controlar la forma en la que permitimos hacerlo).

También es conveniente utilizar la notación `clase::funcion()` y la notación `clase::atributo` para hacer referencia a miembros estáticos, ya que esta notación es exclusiva de miembros estáticos.

```
#include <iostream>
#include <cmath>
using namespace std;
class Punto {
  int x, y;
  static Punto o; //origen comun para todos los puntos
  static int n;
public:
  Punto(int a=0, int b=0) { x = a; y = b; n++; }
  void set(int a, int b) { x = a; y = b; }
  friend float distancia(const Punto &, const Punto &);
  static float distancia(const Punto &, const Punto &);
  static int getn() { return n; }
  static void setn(int n) { Punto::n=n; } //para distinguir n de Punto::n
  static Punto getorigen() { return o; }
  static void setorigen(Punto p) { o = p; }

void dibujar() { cout << "(" << x+o.x << "," << y+o.y << ")\n"; }
};//dibujar muestra el punto tomando o como origen de coordenadas

Punto Punto::o; // Punto Punto::o(0,0);
int Punto::n = 0;
//forma de definir e inicializar los miembros estaticos
//fuera de la clase! Estas sentencias son obligatorias!!!
//indicar el tipo (Punto y int) es imprescindible

float distancia(const Punto &a, const Punto &b) {
 return sqrt(pow((b.x-a.x),2.0)+pow((b.y-a.y),2.0));
}

float Punto::distancia(const Punto &a, const Punto &b) {
 return sqrt(pow((b.x-a.x),2.0)+pow((b.y-a.y),2.0));
}

int main(int argc, char *argv[]) {
  cout << "Hay " << Punto::getn() << " puntos creados\n";
  Punto::getorigen().dibujar();  Punto::getorigen().set(7,7); //modifica una copia
  Punto::setn(Punto::getn()-1); //para que no cuente el punto origen
  Punto p1(1,1), p2(5,4);
  cout << "Hay " << Punto::getn() << " puntos creados\n";
  p1.dibujar();
  p2.dibujar();
  cout << distancia(p1,p2) << endl;
//cambio el origen de coordenadas a 2,3
  Punto::setorigen(Punto(2,3)); //p2.setorigen(Punto(2,3));
  p1.dibujar();
  p2.dibujar();
  cout << Punto::distancia(p1,p2) << endl; //la distancia no cambia
  //Punto::distancia(p1,p2) es lo mismo que p2.distancia(p1,p2) y que p1.distancia(p1,p2)
//cambio el origen de coordenadas al indicado en p2
  Punto::setorigen(p2); //p1.setorigen(p2);
  p1.dibujar();
  p2.dibujar();
  cout << distancia(p1,p2) << endl; //la distancia no cambia
  system("PAUSE"); return EXIT_SUCCESS;
}
```

Pantalla:

```
Hay 1 puntos creados
(0,0)
Hay 2 puntos creados
(1,1)
(5,4)
5
(3,4)
(7,7)
5
(6,5)
(10,8)
5
```

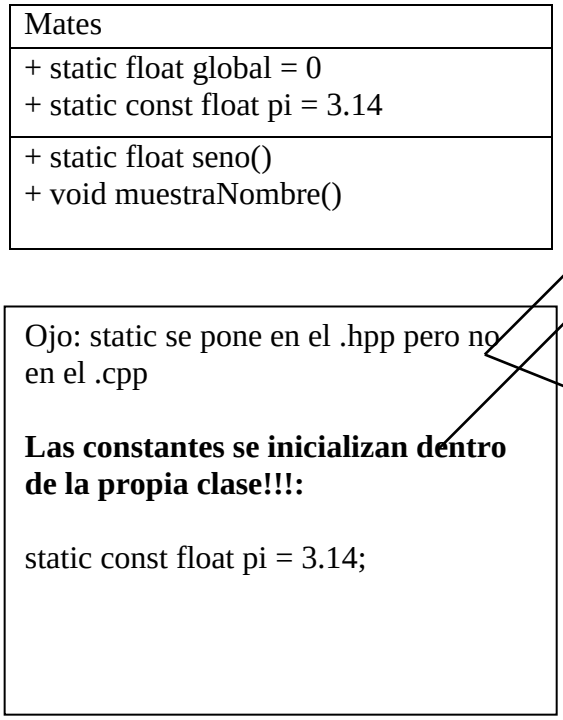
Static: clase privada no instanciable (estática o *singleton*).

Tenemos una clase que no se debe instanciar. Sirve para agrupar semánticamente métodos y agrupar variables globales relacionadas.

Los atributos y los métodos serán static

Evitamos la instanciación declarando el constructor por defecto como privado.

¿Qué pasa con el destructor?




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

class Mates{
private:
   Mates() { cout << "Constructor"; } //constructor privado
public:
   static float global;
   static const float pi = 3.14;
   static float seno( float x ){
      return sin(x);
   }
};
float Mates::global = 0;

int main(int argc, char *argv[]) {
   Mates::global = 10;
   cout << Mates::global <<" : "
        <<Mates::seno( Mates::pi/2 )<<"\n";
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

No es lo mismo clase abstracta que clase privada estática no instanciable.

Una clase que derive de una clase abstracta si se puede instanciar, pero una clase privada que derive de una clase estática no instanciable no se puede instanciar (al tener el contructor en la parte privada, la clase derivada no puede invocar dicho constructor y por tanto no puede instanciarse).

En realidad **una clase privada puede ser instanciada mediante una función no miembro amiga de la clase**, aunque quizás no tenga mucho sentido el hacerlo:

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

class Mates{
   Mates() { cout << "Constructor\n"; } //constructor privado
   ~Mates() { cout << "Destructor\n"; } //destructor privado
public:
   static float global;
   static const float pi = 3.14;
   static float seno( float x ){
      return sin(x);
   }
   friend void creacion( ); //puede acceder a la parte privada
};

float Mates::global = 0;
```

```
void creacion() {
 Mates x; //invoca constructor por defecto
 x.global=5;
 cout << x.global << endl;
}

int main(int argc, char *argv[]) {
   Mates::global = 10;
   cout << Mates::global <<" : "
        <<Mates::seno( Mates::pi/2 )<<"\n";
   creacion();
   cout << Mates::global <<" : "
        <<Mates::seno( Mates::pi/2 )<<"\n";
   system("PAUSE");   return EXIT_SUCCESS;
}
```

Pantalla:

```
10 : 1
Constructor
5
Destructor
5 : 1
```

Objetos y miembros constantes
Se pueden definir objetos constantes del mismo modo que hacemos con las variables tipos normales. Un objeto constante no puede modificarse desde el momento de su construcción hasta su destrucción (las únicas operaciones que podemos hacer es construirlos y destruirlos).

Un objeto constante sólo puede invocar (aparte de sus constructores y destructores) métodos constantes. **Los métodos estáticos no se pueden definir const.**

Atributos constantes
En una clase podemos definir atributos constantes. Un atributo constante no se puede modificar una vez inicializado (sólo se pueden inicializar, no modificar). Para inicializar un atributo constante obligatoriamente hay que hacerlo mediante inicializadores.

Atributos mutables
En una clase podemos definir atributos que pueden cambiar dentro de objetos constantes. Un atributo declarado mutable puede ser modificado dentro de un método constante.

```
#include <iostream>
using namespace std;
class cla {
 int a;
 const float fijo;
 mutable int i;
public:
 cla():a(0),fijo(9.8),i(0) { }
 cla(int x, float v, int y)
   :a(x),fijo(v),i(y) { }
 int geta() const { return a; }
 float getfijo() const { return fijo; }
 int geti() const { return i; }
 void set(int x, int y) { a=x; i=y; }
 void seti(int i) const  { this->i=i; }
 cla operator++() const {i++; return *this;}
 friend ostream& operator<<(ostream &s, const cla &c);
 static const cla& maxfijo(const cla &,const cla &);
};
ostream& operator<<(ostream &s, const cla &c){
 s << c.a << ":" << c.fijo << ":" << c.i;
 return s; //s actúa a modo de cout
}
const cla& cla::maxfijo(const cla &x, const cla &y) {
 if (x.fijo > y.fijo)
  return x;
 return y;
}
int main() {
 const cla a;
 cla b(2, 0.5, 3);
 cout << a.geta() << "," << a.getfijo() << "," << a.geti() << endl;
 cout << b.geta() << "," << b.getfijo() << "," << b.geti() << endl;
 b.set(6,3); //a.set(6,3); //ERROR a es constante
 a.seti(5); b.seti(4);
 ++a; ++b;
 cout << a << endl << b << endl;
 cla::maxfijo(a,b).seti(2);
 ++cla::maxfijo(a,b);
 //cla::maxfijo(a,b).set(1,2); //ERROR a es constante
 cout << a << endl << b << endl;
 system("PAUSE");return EXIT_SUCCESS;
}
```

Lo anterior es equivalente a:

```
class cla {
 ...
public:
 cla():fijo(9.8) { a=0; i=0; }
 cla(int x, float v, int y):fijo(v) {a=x; i=y;}
 ...
 friend const cla& maxfijo(const cla &, const cla &);
};
ostream& operator<<(ostream &s, const cla &c);

ostream& operator<<(ostream &s, const cla &c) {
 s << c.geta() << ":" << c.getfijo() << ":" << c.geti();
 return s; //s actúa a modo de cout
}

const cla& maxfijo(const cla &x, const cla &y) {
 if (x.fijo > y.fijo)
  return x;
 return y;
}
int main() {
 ...
 maxfijo(a,b).seti(2);
 ++maxfijo(a,b);
 //maxfijo(a,b).set(1,2); //ERROR a constnate
 cout << a << endl << b << endl;
 system("PAUSE");return EXIT_SUCCESS;
}
```

Pantalla:

0,9.8,0
2,0.5,3
0:9.8:6
6:0.5:5
0:9.8:3
6:0.5:5

Genéricos (Plantillas)

C++ permite definir funciones y clases parametrizadas o genéricas, o lo que es lo mismo, definir una clase o función en la que los tipos de todos o algunos de los parámetros o miembros es un parámetro más. La forma de definirlos es a través de plantillas (**templates**) y utilizando variables de clase, es decir, variables que pueden tomar como valor un tipo o clase.
El usuario debe indicar el tipo en el momento de usar la función genérica o la clase genérica.
Las plantillas (como las funciones) se pueden sobrecargar.

Función genérica (Plantilla de funciones)

Sintaxis: template <class T1 [,class T2]> declaración_de_la_funcion
template indica al compilador que la definición de clase o función que sigue contiene tipos sin especificar.
Entre < > se pone la lista de parámetros genéricos usados en la declaración_de_la_funcion, puede haber tantos como sea necesarios y todos deben empezar por class

Si alguno de los parámetros genéricos no aparece en los parámetros de declaracion_de_la_funcion, **en la llamada** a la plantilla **habrá que indicarlo** explícitamente **entre <>** después del nombre de la plantilla.

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

class fracc {
  int x,y;
public:
  fracc(int a=0, int b=0) { x=a; y=b; }
};

template <class T1, class T2> void func(T1 a, T2 b, int c); //prototipos
template <class T> T calcular(int i);
template <class T> T calcular(T a, int i); //sobrecarga de plantilla
template <class T1, class T2> T1 proceso(T2 a[], T2& b);
template <class T1, class T2> T1 proceso(T2 a, T1 b); //sobrecarga de plantilla

int main(void) {
  int a, b, c[10];
  float *x[10], y, *z;
  fracc *f1[10], f2(1,5), *f3; //fracc es una clase para manejar fracciones
  //...
  func(a, c[0], 2);            //void func<int,int>(int a, int b, int c)
  func(f3, *z, a);             //void func<fracc*,float>(fracc *a, float b, int c)

  a = calcular<int>(4);        //int calcular<int>(int i);
  f3 = calcular<fracc *>(a);   //fracc *calcular<fracc *>(int i);
  y = calcular(y, a);          //float calcular<float>(float a, int i);

  a = proceso<int>(x, z);      //int proceso<int, float *>(float* a[], float*& b);
  f3 = proceso<fracc *>(z, y); //fracc *proceso<fracc *,float>(float a[], float& b);
  f2 = proceso(f3, f2);        //fracc proceso<fracc, fracc *> (fracc *a, fracc b);

  system("PAUSE");
  return EXIT_SUCCESS;
}
```

En **calcular** el parámetro genérico T no es un parámetro de la función, sino que es el tipo devuelto. En la llamada, el compilador no puede deducir cual es el tipo T, por lo que hay que indicarlo explícitamente.

Lo mismo ocurre en **proceso**, donde el parámetro genérico T1 no forma parte de los argumentos de la función proceso, por lo que en la llamada hay que indicar explícitamente el tipo de T1 entre < >

Función genérica (Plantilla de funciones)
Especialización de plantillas de función:
Una versión de una plantilla para un parámetro de plantilla concreto se denomina especialización.
Cuando el código de una plantilla no sirve para todos los tipos posibles, se puede programar una versión para un tipo concreto. La especialización se puede hacer:

- Programando una versión para un tipo concreto
- Mediante una especialización explícita de la plantilla: **template<>**

Ejemplo: Definir una función mínimo y una función permutar que sirvan para cualquier tipo de dato

En vez de hacer una versión para int, otra para float, otra para char, etc, creamos una plantilla, de forma que en cada llamada el tipo de los parámetros particularizan la plantilla.

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

class fracc {
  int n,d;
public:
  fracc(int a=0, int b=1) { n=a; d=b; }
  void ver() { cout << n << "/" << d << "\n"; }
  bool operator<(fracc f) { return (n*f.d)<(d*f.n); }
};
//prototipos de plantillas y funciones
template <class T> void permutar(T&, T&);
template <class T> T minimo(T a, T b);
template <> char *minimo<char *>(char *a, char *b);
char *minimo(char *a, char *b);

template <class T> void permutar(T& a, T& b){
  T temp=a;  a=b;  b=temp;
}

template <class T> T minimo(T a, T b) {
  cout << "version template <class T>\n";
  if (a < b) return a;
  else return b;
}

template <> char *minimo<char *>(char *a, char *b) {
  cout << "version template <>\n";
  return (strcmp(a,b) < 0) ? a : b;
}

char *minimo(char *a, char *b) {
  cout << "version tipo \n";
  return (strcmp(a,b) < 0) ? a : b;
}

int main(void) {
  int ia=1,ib=5, ic;
  char *cad1="hola", *cad2="adios";
  fracc f1(5,2), f2(1,5), f3;      //objeto tipo fracc
  cout << minimo(ia, ib) << endl; //T es int
  f3 = minimo(f1, (fracc)2);      //T es fracc
  cout << minimo(cad1, cad2) << endl;
  cout << minimo<char *>(cad1, cad2) << endl;
  permutar(f1,f2);                //usa una version de permutar donde T es fracc
  permutar(ia,ib);                //usa una version de permutar donde T es int
  permutar(cad1, cad2);
  f1.ver(); f2.ver(); f3.ver();
  cout << ia << ", " << ib << endl;
  cout << cad1 << ", " << cad2 << endl;
  system("PAUSE");  return EXIT_SUCCESS;
}
```

En este ejemplo la plantilla **permutar** sirve para cualquier tipo de dato.

minimo no sirve para cadenas (char *) porque el operador < no se aplica a cadenas, por lo que hay que hacer una especialización para ese tipo.

Pantalla:

```
version template <class T>      1
version template <class T>
version tipo                    adios
version template <>             adios
1/5  5/2  2/1
5, 1
adios, hola
```

En el ejemplo se ha hecho de las 2 formas posibles. Solo hace falta hacerla de una forma (elimina una de ellas y vera que funciona).

La 2ª se utiliza cuando en la llamada a la plantilla hay que poner uno de los tipos entre <> por no estar dicho tipo en los parámetros (vea calcular y proceso pág. anterior) ya que **cuando se usa <> en la llamada** tiene prioridad la plantilla sobre la especialización tipo concreto.

Si elimina amarillo la versión que se ejecuta en las llamadas a minimo con cadena es la gris

Si elimina gris la versión que se ejecuta en minimo<char *> es la plantilla, no la amarilla

La clase **fracc** debe tener sobrecargado el operador < para que **minimo** no de error. Si no lo tuviera habría que especializar también **minimo** para el tipo fracc y declararla **friend** en la clase fracc para poder acceder a parte privada

```
fracc minimo(fracc a, fracc b) {
 if ((a.n*b.d)<(a.d*b.n)) return a;
 else return b;
}
```

Genéricos: Clase genérica (Plantilla de clase)
Generalmente se utilizan para construir clases contenedoras (clases que contienen otros objetos) de propósito general como pilas, colas, listas enlazadas, arboles, conjuntos, etc, en las que las operaciones a realizar (insertar, eliminar, buscar,etc.) son las mismas independientemente del tipo de objeto que aloje.

Sintaxis: template <class T1 [,class T2]> declaración_de_la_clase
Un objeto plantilla de clase se declara indicando entre <> los tipos correspondientes a los tipos genéricos.

Especialización de plantillas de clase:
Cuando el código de una plantilla de clase no sirve para todos los tipos posibles, se puede programar una versión para un tipo concreto. La especialización de plantilla se hace indicando **template<>**

También se puede hacer una **Especialización parcial de plantillas de clase.**

Especialización de un método de la plantilla de clase:
En vez de especializar una plantilla completa, también se puede especializar métodos concretos.
En el ej. Especializamos ver() para fracc,fracc porque no tenemos sobrecargado << en fracc

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

class fracc {
  int n,d;
public:
  fracc(int a=0, int b=1) { n=a; d=b; }
  void ver() const { cout << n << "/" << d; }
};

template <class T1, class T2>
class Generica {
  T1 x;
  T2 y;
  int n;
public:
  Generica(T1 x, T2 y, int i=0) {
    cout << "template <class T1, class T2>\n";
    this->x=x; Generica::y=y; n=i;
  }
  void ver() const;
};

template <class T1, class T2>
void Generica<T1, T2>::ver() const
 { cout << x << ", " << y << ", " << n << endl; }

template <>
void Generica<fracc, fracc>::ver() const
 { x.ver(); cout << ", "; y.ver(); cout << ", " << n << endl; }

template <class T>
class Generica<T, float> {
  T x;
  float y;
  int n;
public:
  Generica(T x, float y, int i=0) {
    cout << "template <class T>\n";
    this->x=x; Generica::y=y; n=i;
  }
  void ver() const {
    cout << x << ", " << y << ", " << n << endl;
  }
};
```

Si un método se define fuera de la clase hay que poner lo que está en **negrita**

Para especializar un método se pone:
template <>
tipo Clase<tipos_concreto>>::

```
template<>
class Generica<char *, char *> {
  char* x;
  char* y;
  int n;
public:
  Generica(char* a, char* b, int i=0) {
    cout << "template <>\n";
    x=new char[strlen(a)+1]; strcpy(x, a);
    y=new char[strlen(b)+1]; strcpy(y, b); n=i;
  }
  void ver() const {cout << x <<", "<< y <<", "<< n << endl;}
  int getn() { return n; } //solo lo tiene esta plantilla!!!
};

int main(void) {
  char *cad1="hola", *cad2="adios";
  fracc f1(5,2), f2(1,5), f3;     //objetos tipo fracc
  Generica<int, int> g1(2,3);
  Generica<fracc, fracc> g2(f1, f2, 3);
  Generica<fracc, float> g3(f1, 2.3, 3);
  Generica<char *, float> g4(cad1, 1.0);
  Generica<char*, char*>g5(cad1, cad2);
  g1.ver();
  g2.ver();
  //g3.ver(); //ERROR porque fracc no tiene sobrecargado <<
  g4.ver();
  g5.ver();
  //g1.getn(); g2.getn(); g3.getn(); g4.getn(); //ERROR
  cout << g5.getn() << endl;
  system("PAUSE");  return EXIT_SUCCESS;
}
```

Pantalla:

```
template <class T1, class T2>
template <class T1, class T2>
template <class T>
template <class T>
template <>
2, 3, 0
5/2, 1/5, 3
hola, 1, 0
hola, adios, 0
0
```

Genéricos

Clase genérica:

Punto< T >
- T x - T y
+ Punto(T x, T y) + void pinta()

Complejo
- double real - double imag
+ Complejo (double r=0, double i=0) + friend operator<<

Pantalla:

```
(10, 5)
(10, 5)
(a, b)
(10.23, -0.77)
(10.23, -0.77)
(1+0.45i, 3.2+4i)
pi y copia son iguales
```

Lo amarillo no es necesario ya que al no haber memoria dinámica, el constructor de copia y el operador de asignación de oficio que genera el compilador en caso que nosotros no lo hagamos, hace una copia binaria de los datos, que es lo que nosotros hemos codificado.
Observe lo marcado en negrita en ambos métodos: hay que poner <T> porque es una plantilla.

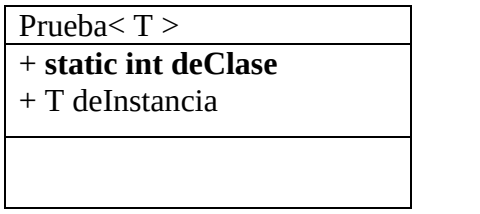
Pruebe a comentar ambos métodos y verá que el resultado obtenido es el mismo

Lo verde es un ejemplo de función no miembro amiga. Observe que hay que usar una letra diferente (H) distinta a la usada para la clase (T) ya que no es un método de la clase sino una función no miembro.

Lo gris no se debe usar para crear un objeto temporal porque lo que se crea con new solo se libera con delete

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <string.h>

using namespace std;

template <class T>
class Punto{
private:
   T x,y;
public:
   Punto( T nx, T ny );
   Punto(const Punto<T>& p)  { x=p.x ; y=p.y ; } //constructor de copia
   Punto<T>& operator=(const Punto<T> &p) ; //operador asignacion
   void pinta();
   template <class H> friend bool operator==(const Punto<H> &a, const Punto<H> &b);
};

template <class T>
Punto<T>::Punto( T nx, T ny ){    x=nx;    y=ny;    }

template <class T>
void Punto<T>::pinta(){   cout << "(" << x << ", " << y << ")\n";  }

template <class T>
Punto<T>& Punto<T>::operator=(const Punto<T> &p) {
  if (this!= &p) { x=p.x ; y=p.y ; }
}

template <class H> bool operator==(const Punto<H> &a, const Punto<H> &b) {
 return (a.x==b.x && a.y==b.y);
}

class Complejo {
   double real, imag;
public:
   Complejo( double r=0, double i=0 ){ real=r; imag=i; }
   friend ostream& operator<<( ostream &s, const Complejo &c ) {
      s << c.real << "+" << c.imag << "i";
      return s;
   }
};

int main(int argc, char *argv[])
{
   Punto<int> pi(10, 5), copia(pi);
   Punto<char> pc('a', 'b');
   Punto<double> pd(10.23, -0.77),  cpd(0,0);
   Punto<Complejo> pcom( *(new Complejo(1, 0.45)),
                         Complejo(3.2, 4) );
   cpd=pd;
   pi.pinta(); copia.pinta(); pc.pinta();
   pd.pinta(); cpd.pinta();pcom.pinta();
   if (pi==copia) cout << "pi y copia son iguales\n";
   system("PAUSE");    return EXIT_SUCCESS;
}
```

Genéricos
Clase genérica: variables y métodos de clase (static)



Ojo: static se pone en el .hpp pero no en el .cpp

Las static const se inicializan dentro de la propia clase!!!:

```
#include <cstdlib>
#include <iostream>

using namespace std;

template< class T >
class Prueba{
public:
   static int deClase;
   static const float pi = 3.14;
   T deInstancia;
};
template< class T >
int Prueba<T>::deClase=0;

int main(int argc, char *argv[])
{
   Prueba<int> p1; //ambas comparten deClase y pi
   Prueba<int> p2;
   Prueba<float> p3;

   p1.deClase=1;    //Prueba<int>::deClase=1;
   p2.deClase=2;    //Prueba<int>::deClase=2;
   p3.deClase=3;    //Prueba<float>::deClase=3;

   cout << p1.deClase << "," << p1.pi << " -- "
        << p2.deClase << "," << p2.pi << " -- "
        << p3.deClase << "," << p3.pi << "\n";

   cout << Prueba<int>::deClase << ","
        << Prueba<int>::deClase << ","
        << Prueba<float>::deClase << "\n";


   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Declaración previa de una clase genérica:
Una clase genérica, al igual que una clase normal puede tener una declaración adelantada

Declaración previa necesaria

```
template <class T> class Punto; //declaracion previa de
                                //la plantilla Punto
template <class T> //definicion de la plantilla Otra
class Otra {
  T x;
public:
  friend class Punto<T>;  //cuando llega aquí sabe
 …                        //que Punto es una plantilla
};

template <class  T> //definicion de la plantilla Punto
class Punto {
  T x, y;
  …
};
```

Declaracion previa no necesaria

```
template <class  T> //definicion de la plantilla Punto
class Punto {
  T x, y;
  …
};

template <class T> //definicion de la plantilla Otra
class Otra {
  T x;
public:
  friend class Punto<T>;  //sabe que Punto es una plantilla
 …
};
```

Genéricos
Clase genéricas: Herencia

Al igual que una clase **(clase derivada)** se puede crear a partir de otra clase (**clase base**) introduciendo sus propios atributos y métodos, una plantilla **(plantilla derivada)** se puede crear a partir de otra plantilla (**plantilla base**) o de una clase normal (clase base).

Ejemplo de herencia a partir de una PLANTILLA base

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

class fracc {
  int n,d;
public:
  fracc(int a=0, int b=1) { n=a; d=b; }
  friend ostream& operator<<( ostream &s, const fracc &f ) {
    s << f.n << "/" << f.d;
    return s;
  }
};

template <class T>
class PBase {
  T x;
public:
  PBase(T x) {
    cout << "Plantilla Base\n";
    this->x=x;
  }
  void ver() { cout << x; }
  T getx();
};

template <class T>
T PBase<T>::getx() { return x; }

template <class T1, class T2>
class PHija1: public PBase<T1> {
  T2 y;
public:
  PHija1(T1 x, T2 y):PBase<T1>(x) {
    cout << "Plantilla Hija1\n";
    this->y=y;
  }
  void ver() {  PBase<T1>::ver(); //llamo a ver() del padre
                cout << " " << y << endl; }
  void ver2() {
    cout << PHija1<T1, T2>::getx() << "," << y << endl;
  }
  void ver3();
};

template <class T1, class T2>
void PHija1<T1, T2>::ver3() {
  cout << PBase<T1>::getx() << "-" << y << endl;
}
```

```
template <class T>
class PHijo: public PBase<T> {
  int y;
public:
  PHijo(T x, int y):PBase<T>(x) {
    cout << "Plantilla Hijo\n";
    this->y=y;
  }
  int gety() { return y; }
};

int main(void) {
  fracc f1(5,2);     //objetos tipo fracc
  PBase<int> pb1(2);
  PBase<fracc> pb2(f1);
  PHija1<fracc, fracc> ph1(f1, fracc(2,2));
  PHija1<int, fracc> ph2(6, f1);
  PHijo<int> ph(500,6);
  pb1.ver(); cout << endl;
  pb2.ver(); cout << endl;
  ph.ver(); cout << " " << ph.gety() << endl;
  ph1.ver();  ph1.ver2(); ph1.ver3();
  ph2.ver();  ph2.ver2(); ph2.ver3();
  ph2.PBase<int>::ver(); cout << endl;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Pantalla:

```
Plantilla Base                          //pb1
Plantilla Base                          //pb2
Plantilla Base Plantilla Hija1          //ph1
Plantilla Base Plantilla Hija1          //ph2
Plantilla Base Plantilla Hijo           //ph
2                                 // pb1.ver(); cout << endl;
5/2                               // pb2.ver(); cout << endl;
500 6                             // ph.ver(); ph.gety();
5/2 2/2                           // ph1.ver();
5/2,2/2                           // ph1.ver2();
5/2-2/2                           // ph1.ver3();
6 5/2                             // ph2.ver();
6,5/2                             // ph2.ver2();
6-5/2                             // ph2.ver3();
6                                 //ph2.PBase<int>::ver();
```

getx() se hereda del padre y hay que indicarlo asi (diciendo que es un método de clase hija) o indicando que es el método de la clase padre (es lo mismo)

Genéricos

Clase genéricas: Herencia

Al igual que una clase **(clase derivada)** se puede crear a partir de otra clase (**clase base**) introduciendo sus propios atributos y métodos, una plantilla **(plantilla derivada)** se puede crear a partir de otra plantilla (plantilla base) o de una clase normal (**clase base**).

Ejemplo de herencia a partir de una CLASE base

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

class fracc {
  int n,d;
public:
  fracc(int a=0, int b=1) { cout << "Fraccion\n"; n=a; d=b; }
  friend ostream& operator<<( ostream &s, const fracc &f ) {
    s << f.n << "/" << f.d;
    return s;
  }
  fracc get() { return *this; }
};

template <class T>
class PHija: public fracc {
  T x;
public:
  PHija(T x, int n=0, int d=1): fracc(n, d) {
    cout << "Plantilla\n";
    this->x=x;
  }
  void ver() { cout << x << " " << fracc::get(); }
  T get(); //oculta el get() de fracc
};

template <class T>
T PHija<T>::get() { return x; }
```

```
int main(void) {
  fracc f1(5,2);     //objetos tipo fracc
  PHija<int> ph1(2);
  PHija<char *> ph2("hola", 2);
  PHija<fracc> ph3(f1, 2, 2);
  ph1.ver(); cout << endl;
  ph2.ver(); cout << endl;
  ph3.ver(); cout << endl;
  cout << ph2.get(); cout << endl;
  cout << ph2.fracc::get(); cout << endl;
  system("PAUSE");
  return EXIT_SUCCESS;
}
```

Pantalla:

```
Fraccion                    //f1
Fraccion Plantilla          //ph1
Fraccion Plantilla          //ph2
Fraccion Fraccion Plantilla //ph3
2 0/1                       //ph1.ver( )
hola 2/1                    //ph2.ver( )
5/2 2/2                     //ph3.ver( )
hola                        //cout << ph2.get();
2/1                         //cout << ph2.fracc::get();
```

getx() de plantilla oculta el getx() heredado del padre

Organización del código de las Plantillas: IMPORTANTE!!!

Al trabajar con plantillas todo el código lo debemos tener en un único fichero. Si queremos tener un fichero .cpp y otro .hpp debemos incluir al final del fichero .hpp el fichero .cpp, es decir, debemos hacer lo siguiente (el .cpp no se incluye en el proyecto, sino que se incrusta con #include en el .hpp):

PHija.hpp (si se añade al proyecto)

```
#ifndef PHIJA_HPP
#define PHIJA_HPP
#include "fracc.h"
using namespace std;

template <class T>
class PHija: public fracc {
  T x;
public:
  PHija(T x, int n=0, int d=1);
  void ver() ;
  T get(); //oculta el get() de fracc
};

#include "PHija.cpp" //incrusto el .cpp aqui

#endif
```

PHija.cpp (no se añade al proyecto OJO!!!)

```
#include <iostream>
#include "fracc.h"
#include "PHija.hpp"
using namespace std;

template <class T>
PHija<T>::PHija(T x, int n, int d): fracc(n, d) {
   cout << "Plantilla\n";
   this->x=x;
}

template <class T>
void PHija<T>::ver() { cout << x << " " << fracc::get(); }

template <class T>
T PHija<T>::get() { return x; }
```

El ejemplo anterior es equivalente a tenerlo todo en un único fichero, tal como está arriba (si el main lo pongo en otro fichero aparte simplemente debemos añadir #include "PHija.hpp" en el fichero main)

ANEXO:

Ayuda para identificar las necesidades de especialización de una Plantilla

Al trabajar con plantillas y definir una clase o función en la que los tipos de todos o algunos de los parámetros o miembros es un parámetro más, el usuario debe indicar el tipo en el momento de usar la función genérica o la clase genérica.

El compilador, en función del tipo o tipos instanciados en la plantilla, genera para cada tipo el código del programa correspondiente, de manera que si, por ejemplo, tenemos una plantilla de una función no miembro instanciada para el tipo int y float, el compilador genera una función no miembro para el tipo int y otra idéntica para el tipo float (el código es el mismo, lo único que cambia es el tipo).

En función del tipo a que se instancie dicha plantilla, el código de la plantilla puede que sirva o no, de ahí que, si para un tipo de dato concreto el código de la plantilla no sirve, debemos especializarlo.

Una versión de una plantilla para un parámetro de plantilla concreto se denomina especialización. Cuando el código de una plantilla no sirve para todos los tipos posibles, se puede programar una versión para un tipo concreto.

El programador, por tanto, a la hora de implementar las plantillas debe saber de antemano los tipos de datos para los que queremos que funcione dicha plantilla, para así poder anticipar si hace falta o no especializar la plantilla para un tipo o tipos de datos concretos.

Como a veces puede que el programador no sea capaz de anticipar esa necesidad de especializar la plantilla, podemos usar el compilador para ayudarle a identificar esa necesidad.

Para ello, simplemente debemos poner al final de cada fichero .h de la clase o clases que hacen uso de la plantillas una instanciación a los tipos de datos para los que queremos que funcione.

Ej: Si tenemos por ejemplo una plantilla de una clase denominada Pclase y queremos que dicha plantilla sirva para los tipos de datos **int**, **float** y **string** debemos instanciar dichos tipos en la plantilla así:

Pclase.hpp (si se añade al proyecto)

```
#ifndef PCLASE_HPP
#define PCLASE _HPP
using namespace std;

template <class T>
class Pclase {
  T x; …
public:
  Pclase( );
  void ver( ) ;
  void decrementar( );
};

#include "Pclase.cpp" //incrusto el .cpp aqui

template class Pclase<int>;
template class Pclase<float>;
template class Pclase<string>;

#endif
```

Pclase.cpp (no se añade al proyecto OJO!!!)

```
#include <iostream>
#include "Pclase.hpp"
using namespace std;

template <class T>
Pclase<T>::Pclase( ) {  …  }

template <class T>
void Pclase<T>::ver( ) { cout << x ; }

template <class T>
void Pclase<T>:: decrementar ( ) { x--; }
```

Si no instanciamos los tipos de datos, el compilador no puede determinar si el código de la plantilla es válido para esos tipos de datos, a menos que creemos un main( ) y en él definamos objetos de esos tipos concretos.

Haciéndolo así no tenemos por que crear un main( ) para saber si la plantilla va a funcionar o no.

Al compilar el programa, el compilador creará una clase para cada tipo de dato, generando el siguiente código (lo hace en binario en el fichero objeto .o)

```
class Pclase<int> {
  int  x; …
public:
  Pclase<int>( );
  void ver( ) ;
  void decrementar( );
};

Pclase<int>::Pclase<int>( )  { … }

void Pclase<int>::ver() { cout << x ; }

void Pclase<int>:: decrementar ( )  {
   x--;
}
```

```
class Pclase<float> {
  float  x; …
public:
  Pclase< float  >( );
  void ver( ) ;
  void decrementar( );
};

Pclase< float >::Pclase< float >( )  { … }

void Pclase< float >::ver()  { cout << x;}

void Pclase< float >:: decrementar ( ) {
   x--;
}
```

```
class Pclase<string> {
  string x; …
public:
  Pclase< string >( );
  void ver( ) ;
  void decrementar( );
};

Pclase< string >::Pclase< string >( ) { … }

void Pclase< string >::ver( )  { cout << x ;}

void Pclase< string >:: decrementar ( )  {
   x--;
}
```

Al hacerlo, el compilador detectará que para la clase string el método decrementar no es válido, ya que el operator- - no esta definido en la clase string por lo que en la línea **x--;** dará error.

Gracias a la ayuda del compilador, el programador sabrá (si no había caído en la cuenta), que el método decrementar habrá que especializarlo para la clase string, haciendo un código distinto en ella. Suponiendo que el decrementar de string quiero que elimine la última letra del string el código sería:

Pclase.hpp (si se añade al proyecto)

```
#ifndef PCLASE_HPP
#define PCLASE _HPP
using namespace std;

template <class T>
class Pclase {
  T x; …
public:
  Pclase( );
  void ver( ) ;
  void decrementar( );
};

#include "Pclase.cpp" //incrusto el .cpp aqui

template class Pclase<int>;
template class Pclase<float>;
template class Pclase<string>;

#endif
```

Pclase.cpp (no se añade al proyecto OJO!!!)

```
#include <iostream>
#include "Pclase.hpp"
using namespace std;

template <class T>
Pclase<T>::Pclase( ) {  …  }

template <class T>
void Pclase<T>::ver( ) { cout << x ; }

template <class T>
void Pclase<T>:: decrementar ( ) { x--; }

template <>
void Pclase <string>:: decrementar ( )  {
   x.resize(x.size()-1);
}
```

Al compilar el programa, el compilador creará una clase para cada tipo de dato, generando el siguiente código (lo hace en binario en el fichero objeto .o)

```
class Pclase<int> {
  int  x; …
public:
  Pclase<int>( );
  void ver( ) ;
  void decrementar( );
};

Pclase<int>::Pclase<int>( )  { … }

void Pclase<int>::ver() { cout << x ; }

void Pclase<int>:: decrementar ( )  {
   x--;
}
```

```
class Pclase<float> {
  float  x; …
public:
  Pclase< float  >( );
  void ver( ) ;
  void decrementar( );
};

Pclase< float >::Pclase< float >( )  { … }

void Pclase< float >::ver()  { cout << x; }

void Pclase< float >:: decrementar ( ) {
   x--;
}
```

```
class Pclase<string> {
  string x; …
public:
  Pclase< string >( );
  void ver( ) ;
  void decrementar( );
};

Pclase< string >::Pclase< string >( ) { … }

void Pclase< string >::ver( )  { cout << x ;}

void Pclase <string>:: decrementar ( )  {
   x.resize(x.size()-1);
}
```