Tema 3: C++

Declaración de una clase básica:

Class nombre[: <lista de clases>]{

[<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>]*
};

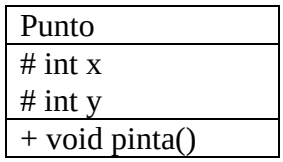
<lista de clases>:: [virtual]<privilegio acceso><nombre clase>[,[virtual]<privilegio acceso><nombre clase>]*

Punto
int x # int y
+ void pinta()

EjemploPunto.cpp

```
#include <iostream>

using namespace std;

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

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

Diseño modular:
El operador de resolución de ámbito **::**
.hpp y .cpp



Punto.hpp

```
#ifndef PUNTOHPP
#define PUNTOHPP

class Punto{
protected:
        int x;
        int y;
public:
        void pinta();
};

#endif
```

Punto.cpp

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

using namespace std;

void Punto::pinta(){
        cout << "Punto\n";
}
```

Ejemplo.cpp

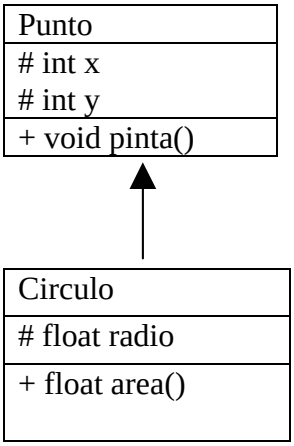
```
#include "Punto.hpp"

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

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

Archivos de código objeto y bibliotecas .o y .a

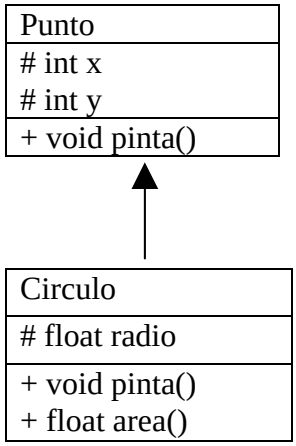
Herencia simple:

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



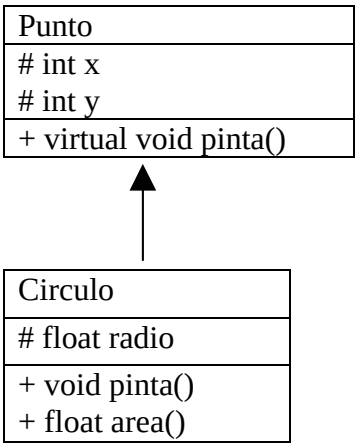

```
class Punto{
protected:
        int x;
        int y;
public:
        void pinta(){
                cout << “Punto\n”;
        }
};

class circulo: public Punto{
protected:
        float radio;
public:
        float area(){
                return 2*3.14*radio;
        }
};

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

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 . o ->:




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

class circulo: public Punto{
protected:
        float radio;
public:
        void pinta(){
                cout << "Punto\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;
}
```

Métodos virtuales: ejemplo.
¿Cómo se llama al de *Punto* con un virtual?




```
class Punto{
protected:
        int x;
        int y;
public:
        virtual void pinta(){
                cout << "Punto\n";
        }
};

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

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

La referencia this

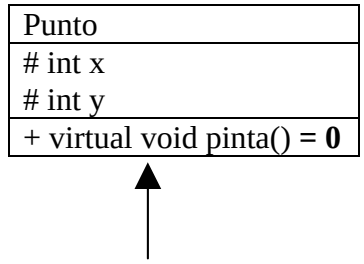
Problemas de ambigüedad por enmascaramiento con locales o argumentos
Estructuras enlazadas

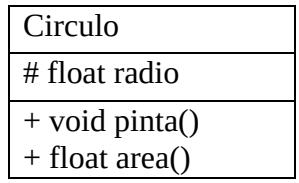
```
class Punto{
private:
        int x;
        int y;
public:
        void cambia( int x, int y ){
          this->x = x;
          this->y = y;
        }
};
```

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

Clase abstracta:
virtual

Método virtual puro.





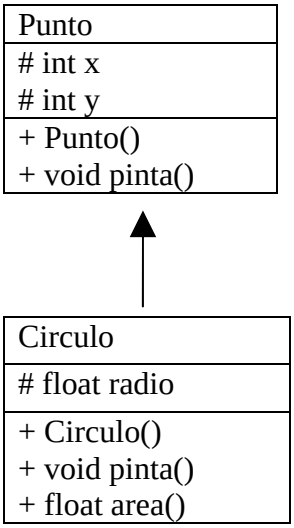
Ojo: virtual se pone en el .hpp pero no en el .cpp

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

class Circulo: public Punto{
protected:
        float radio;
public:
        void pinta(){
                cout << "Círculo\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;
}
```

Acceso a la superclase.

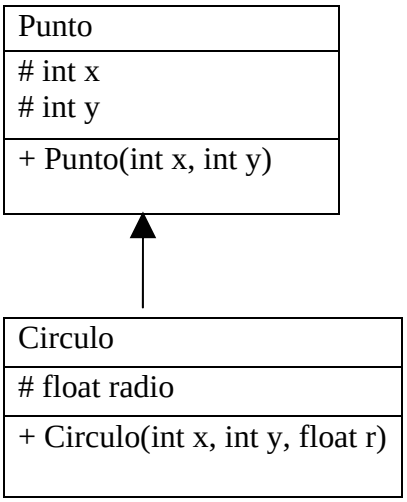
Para acceder a las superclases, utilizaremos el operador de resolución de ámbito **::**

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:




```
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();
        }
        float area(){
                return 2*3.14*radio;
   }
};

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

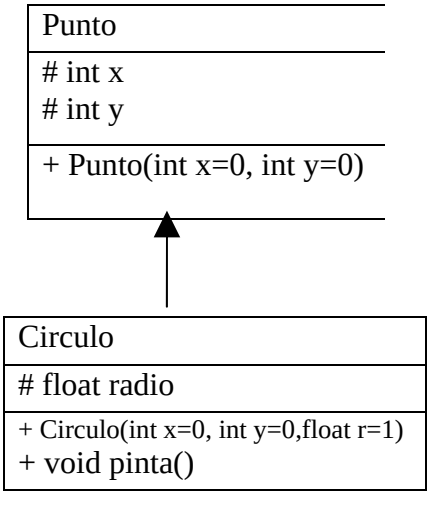
Instanciación: Constructores/Destructores
Constructores no por defecto



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:

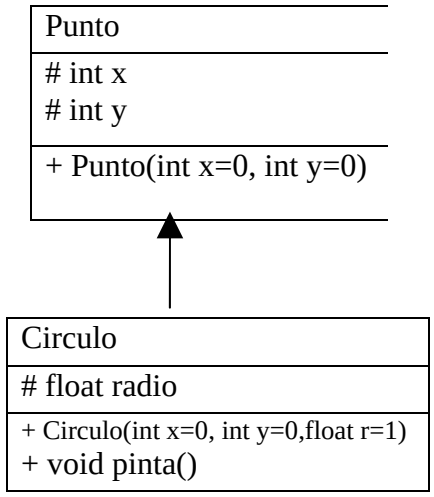
Circulo *c = new Circulo();
Circulo *c = new Circulo(10,20,1.5);

Después de new se especifica un constructor:

con arrays:

char *cadena = new char[20];
Circulo *array = new Circulo[10];

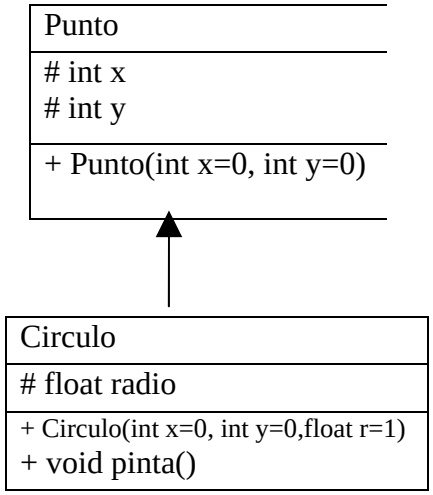
Nota: Para utilizar new con arrays es necesario que exista un constructor por defecto
Recuerde: si se crea un array estático, se llama a los constructores de cada elemento. Si es un array de referencias, esto será responsabilidad del programador.




```
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();   // 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

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




```
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();
   }
   for (int i=0; i<10;i++){
      c[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();
delete c;
```

para arrays:

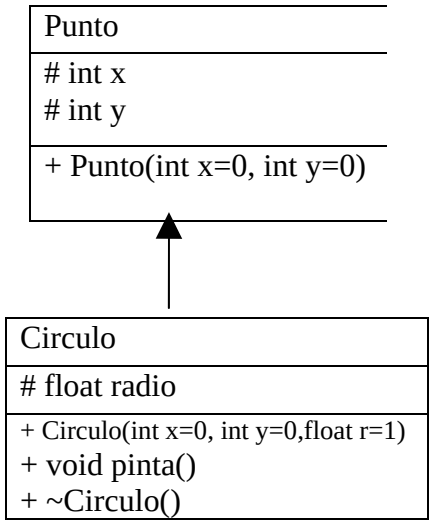
```
Circulo *c = new Circulo[10];
delete [] c;  ¡¡ Atención, esto llama al destructor de cada uno de los 10 círculos !!
                 delete c no falla pero no llama a los destructores correctamente
                 (llama al destructor del primero) !!
```

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 ~

Instanciación: Constructores/Destructores
new y delete

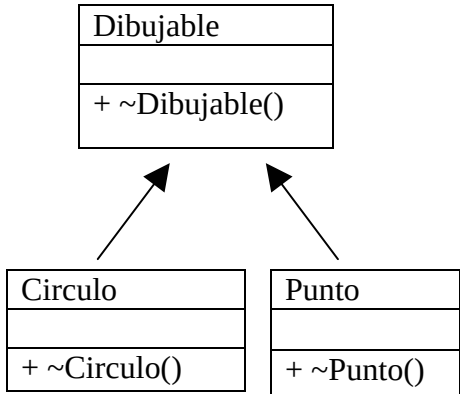
Para realizar correctamente el ejemplo anterior:




```
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();
   }
   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.

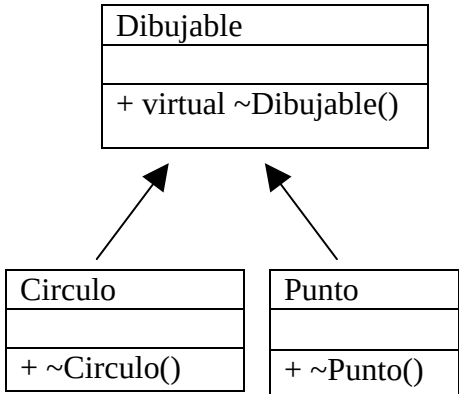


Comportamiento estático

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

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

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

int main(int argc, char *argv[])
{
   Dibujable *d;
   Punto *p = new Punto();
   Circulo *c = new Circulo();
   d = p;
   delete d;
   d = c;
   delete d;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Instanciación: Constructores/Destructores
Destructores polimórficos

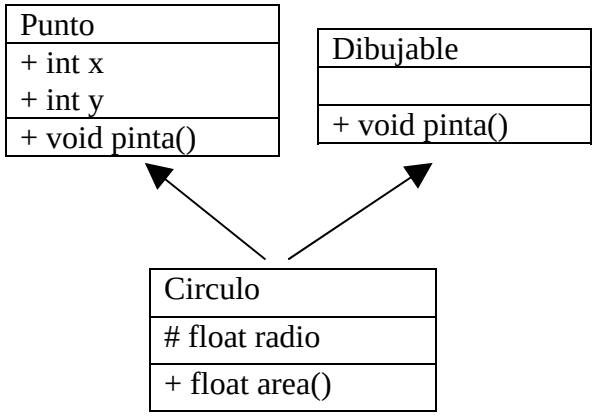
Los destructores polimórficos tienen sentido. Los constructores, no.



Comportamiento dinámico

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

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

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

int main(int argc, char *argv[])
{
   Dibujable *d;
   Punto *p = new Punto();
   Circulo *c = new Circulo();
   d = p;
   delete d;
   d = c;
   delete d;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

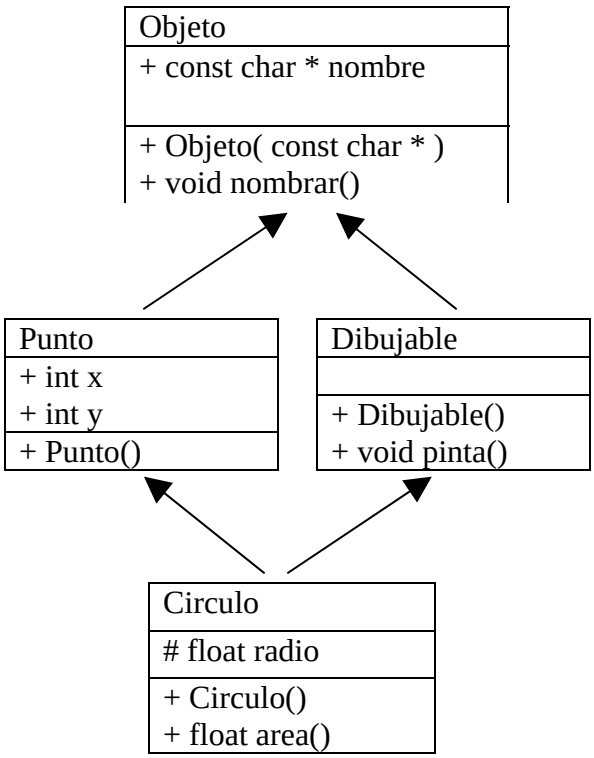
Herencia múltiple: llamada a métodos con el mismo nombre en los antecesores:




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

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

class Circulo: public Punto, public Dibujable{
protected:
        float radio;
public:
   void pinta(){
      Dibujable::pinta();
   };
   float area(){
      return 2*3.14*radio;
   }
};

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

Herencia múltiple: llamadas a constructores: caso del antecesor común

Al declarar los constructores:

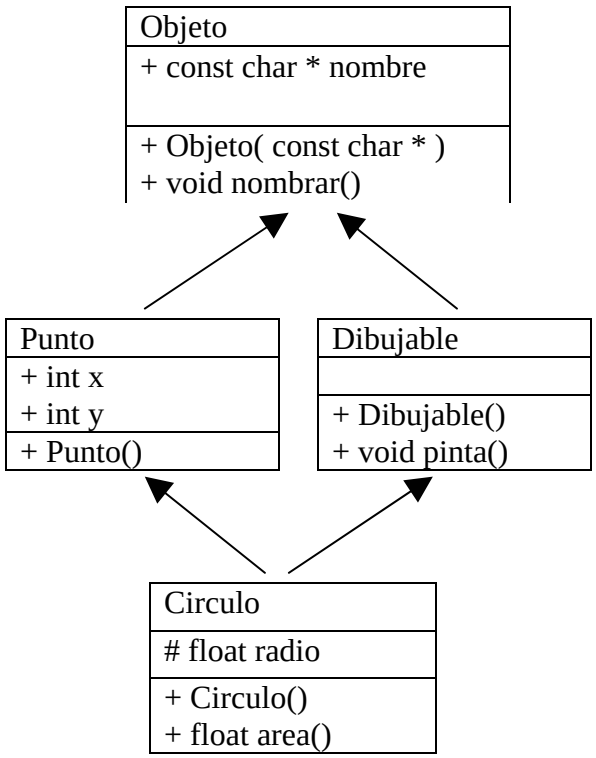


Se pueden llamar a varios constructores:
Circulo(): Dibujable(), Punto(10,0){}
(En caso de que existiera un constructor Punto(int,int))

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

class Punto: public Objeto{
public:
        int x;
        int y;
        Punto(): Objeto( "Punto\n" ){
   }
};

class Dibujable: public Objeto{
public:
   Dibujable(): Objeto( "Dibujable" ){
   }
        void pinta(){
      cout << "Dibujable\n";
   }
};

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

int main(int argc, char *argv[])
{
   Circulo c;
   //Objeto *o; // no permitido
   //o = &c;
   //o->nombrar();
   Punto *p = &c;
   Dibujable *d = &c;
   p->nombrar();
   d->nombrar();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Herencia múltiple: llamadas a constructores: caso del antecesor común

Al declarar los constructores:

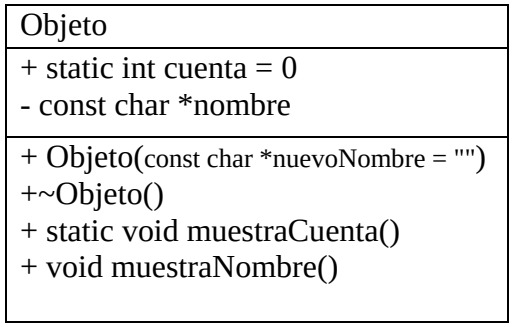



```
class Objeto{
private:
   const char * nombre;
public:
        Objeto( const char *nombre ){
      // Reserva de memoria!!
      this->nombre = strdup(nombre);
   }
   void nombrar(){
      cout << this->nombre;
   }
};

class Punto: virtual public Objeto{
public:
        int x;
        int y;
        Punto(): Objeto( "Punto\n" ){
   }
};

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

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

int main(int argc, char *argv[])
{
   Circulo c;
   Objeto *o;
   o = &c;
   o->nombrar();
   Punto *p = &c;
   Dibujable *d = &c;
   p->nombrar();
   d->nombrar();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

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



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

Inicialización de variables de clase static (no constantes). Las constantes se pueden inicializar dentro de la propia clase:

static const float pi = 3.14;

```
class Objeto{
private:
   const char * nombre;
public:
   static int cuenta;
  Objeto( const char *nuevoNombre = "" ){
      nombre = strdup(nuevoNombre);
      cuenta++;
   }
    virtual ~Objeto(){ cuenta--; }
    static void muestraCuenta(){
      //cout << nombre; No permitido
      cout << cuenta;
   }
    void muestraNombre(){
      cout << cuenta <<":"<< nombre<<"\n";
   }
};
int Objeto::cuenta=0;

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

Static: clase 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?