El polimorfismo

- El polimorfismo se refiere al hecho de que una misma función adopte múltiples formas.
- Esto se consigue por medio de la sobrecarga:
  - Sobrecarga de funciones:
    - un mismo nombre de función para distintas funciones.

```
a = Sumar(c,d);
a = Sumar(c,d,5);
```

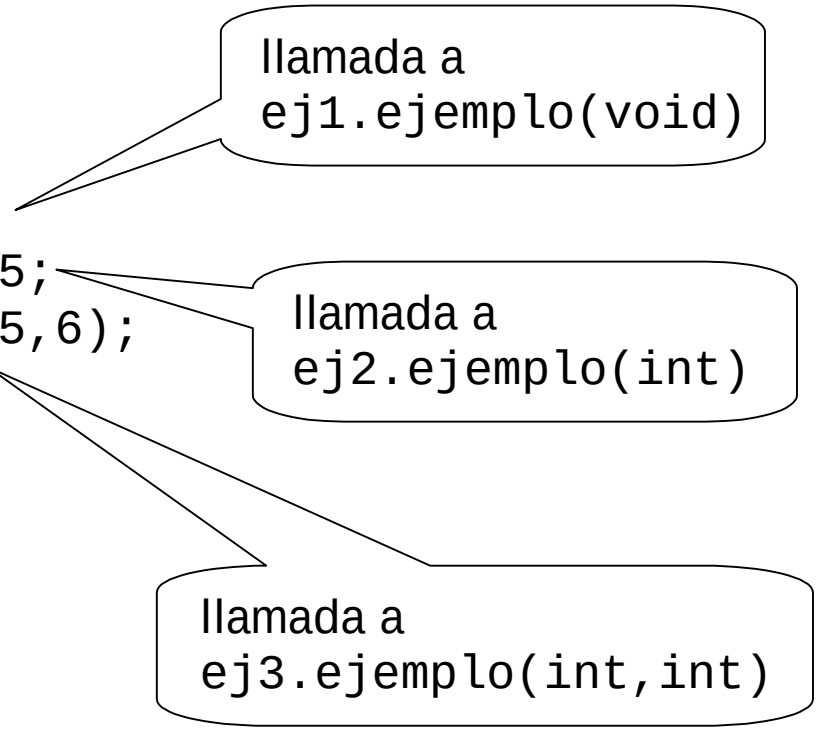
  - Sobrecarga de operadores:
    - un mismo operador con distintas funcionalidades.

```
entero1 = entero2 + 5;
cadena1 = cadena2 + cadena3;
```

    - Podemos sobrecargar el operador + sobre la clase cadena para que permita concatenar dos de estos objetos.

Sobrecarga de constructores

```
class ejemplo
{ int x;

public:
  ejemplo(void) { x = 10; }
  ejemplo(int i) { x = i; }
  ejemplo(int i,int j) { x = i+j; }
};


void f(void)
{ ejemplo ej1;
  ejemplo ej2=5;
  ejemplo ej3(5,6);

...
}
```



- Recordatorio:
  - Solo en caso de que no definamos ningún constructor tenemos el constructor por defecto.

Sobrecarga de funciones

```
class string
{ char *v;
  int tam;

public:
  string(void);
  string(string & s);
  friend int strcmp(string s1,string s2);
  friend int strcmp(string s1, char *c);
  friend int strcmp(char *c,string s1);
  string strcpy(string s);
  string strcpy(char *c);
};
```

- Tenemos 2 funciones costructor `string()`.

- Tenemos 4 posibles llamadas a 4 funciones diferentes `strcmp()`:
  - 3 en la clase.
  - 1 en la librería `string.h` de C.

- Tenemos 2 funciones `strcpy()` en la clase.

```
void main(void)
{ string s1,s2;
  char c1[30],c2[30];

...

strcmp(s1,s2); // Compara las cadenas de
               // dos objetos string.

strcmp(s1,c1); // Compara un objeto
               // string y una cadena de
               // caracteres.
strcmp(c1,s1); // Viceversa que anterior

strcmp(c1,c2); // Compara dos cadenas de
               // caracteres

s1.strcpy(s2); // Copia un objeto string
               // s2 a un objeto string
               // s1

s1.strcpy(c1); // Copia la cadena c1 al
               // objeto string s1.
}
```

Sobrecarga de funciones

- En la sobrecarga de funciones se desarrollan distintas funciones con un mismo nombre pero distinto código.
- Las funciones que comparten un mismo nombre deben tener una relación en cuanto a su funcionalidad.
- Aunque comparten el mismo nombre, deben tener distintos parámetros. Éstos pueden diferir en :
  - El número
  - El tipo
  - El orden

  de manera que el compilador pueda distinguir entre las distintas funciones cuando encuentra una llamada.
- El tipo del valor de retorno de una función no es válido como distinción.
  - Esto es debido a que ese valor en C++ no es necesario que sea recogido por otro objeto.

Ejemplo de sobrecarga de funciones

```
void concatenar(char *, char *);
void concatenar(char *,char *,char *);
void concatenar(char *,int);

void main(void)
{ char v1[50]="1";
  char v2[30]="2" ;
  int val=3;
  char v3[10]="4";

   concatenar(v1,v2);
   concatenar(v1,v2,v3);
   concatenar(v1,val);
}

void concatenar(char *s1,char *s2)
{ strcat(s1,s2); }

void concatenar(char *s1,char *s2,char *s3)
{ strcat(s1,s2);
  strcat(s1,s3);
}

void concatenar (char *s1,int v)
{ char cadena[50];

itoa(v,10,cadena);
strcat(s1,cadena);
}
```

Sobrecarga de operadores

- Consiste en definir nuevas funcionalidades para los operadores definidos en el lenguaje.
  - Por ejemplo:

```
string s1,s2;
string s3=s1+s2;
```

concatenación de dos objetos string mediante el operador +

- Se pueden sobrecargar prácticamente todos los operadores del lenguaje:
  - No podemos inventar nuevos operadores.

```
int a; X x;
x**a ;      // NO es posible
```

- Los operadores son de distinta aridad:

```
unarios ++, --, -, ...
binarios +, -, *, %
ternarios ?:
```

- Tampoco podemos cambiar su aridad:

```
%a;         // NO es posible
```

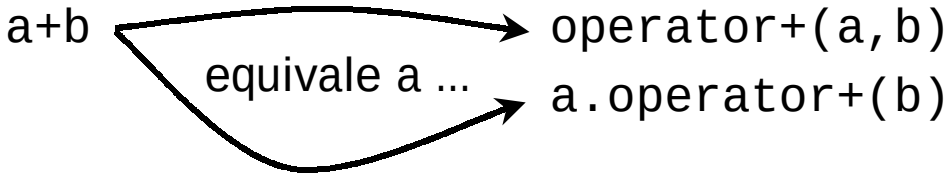
- No podemos re-definir funcionalidades de los operadores que ya están definidas:

```
int a=0;
a++;
```

No puedo cambiarlo para que haga otra cosa distinta a incrementar el valor de un entero.

- Al menos uno de los operandos del operador sobrecargado debe ser una clase definida por nosotros:
  - Ejemplo:
    - No puedo re-definir la suma de dos enteros, o la concatenación con el operador + de dos `char *`.
    - Si que puedo definir la concatenación de un objeto `string` con un `char *`, porque la clase `string` la definimos nosotros.

- La sobrecarga de operadores es un caso particular de la sobrecarga de funciones:



  - El operando de la izquierda puede pasar a ser el propietario de la función, o bien
  - el primer argumento de la función.

- Función `operator` miembro:

```
class A
{ public:
    A operator+(A b);
};

A A::operator+(A b)
{ ... }
```

- Función `operator` no miembro de una clase:

```
A operator+(A a,A b)
{ ... }
```

```
class A
{ int x;

public:
  A(void) { x=10; }
  A operator+(A a);
  A operator+(int i);
  friend A operator(int i,A a);
};

A A::operator+(A a)
{ A tmp;
tmp.x = x+a.x;
return tmp;
}

A A::operator+(int i)
{ A tmp;
tmp.x = x+i;
return(tmp);
}

A operator+(int i,A a)
{ A tmp;
tmp.x= a.x+i;
return(tmp);
}
```

Se define como “friend”

```
void main(void)
{ int k=3;
  A c;
  A d;

c=c+d;
c=d+k;
c=k+d;
}
```

```
void main(void)
{ int k=3;
  A c;
  A d;

c=c.operator+(d);
c=d.operator+(k);
c=operator+(k,d);
}
```

El operador asignación

- Para cualquier clase que se defina, el compilador define el operador asignación "=".

```
class A
{  int a;
 public: ...
};

A c,d;
c=d;
```

Se puede hacer sin definir operator= El resultado es la copia del valor d.a en c.a

- Cuando no me sirve, debo re-definirlo:

```
class B
{  int tam;
   int *buffer;
public:
  B(B &);
  ~B(void);
  B operator=(B);
}

B B::operator=(B b)
{ if (tam>0) delete buffer;
  tam=b.tam;
  buffer=new int[tam];
  for(int i=0; i<tam; i++)
    buffer[i]=b.buffer[i];
  return(*this);
}
```

Normalmente, tenemos que definirlo cuando necesitamos definir el constructor copia

Esto permite hacer:
a=b=c;

El operador incremento y el decremento

- Tiene dos posibles notaciones:
  - prefija: ++a
  - postfija: a++

```
class X
{ int i;

public:
  X operator++(void); // pre-incremento
  X operator++(int);  // post-incremento
}

X X::operator++(void)
{ i++;
  return(*this);
}

X X::operator++(int)
{ int aux=i;

i++;
return(aux);
}
```

Lo incrementamos ya pero se utiliza el valor antes de incrementar en la expresión donde se encuentre:

```
a=(b++);
// a=b.operator++();
c=b;  // c es a+1
```

Uso de referencias en sobrecarga de operadores

- Devolución de una referencia:
  - Se debe devolver una referencia cuando:
    - queremos que se pueda modificar el propio objeto que se devuelve.
  - Ejemplo: el operador subíndice.

```
class V
{ int vector[30];

public:
  int & operator[](int j);
};

int & V::operator[](int j)
{ return vector[j]; }

void main(void)
{ int k;
  V v;
...

k=v[5];
v[5]=k;
}
```

Para este caso NO hace falta la referencia.

Para este caso SI que es necesaria la referencia.

- Índices dobles:

```
class linea
{ int ln[10];

public:
  int & operator[](int j)
                  { return ln[i]; }
};

class matriz
{ linea mat[10];

public:
  linea & operator[](int j)
                    { return mat[j]; }
};


void main(void)
{ matriz m;
  int i;

i=m[5][5];
m[5][5]=i;
}
```

m[5] devuelve el objeto linea que está en la posición 5 de mat.

m[5][5] devuelve el entero que está en la posición 5 del vector ln que contiene el objeto linea que está en la posición 5 dela matriz mat.

Conversión de clases mediante constructores

```
class X
{ ...
}
```

```
class Y
{ ...

public :
  y ( X x )  {....}
  ...
}
```

- Un constructor Y::Y(X x) permite al compilador convertir objetos de la clase X en objetos de la clase Y.
- Este constructor permite :

```
X x;
Y y=x;
```

```
Void f(Y y);
...

X x;
f(x);
```

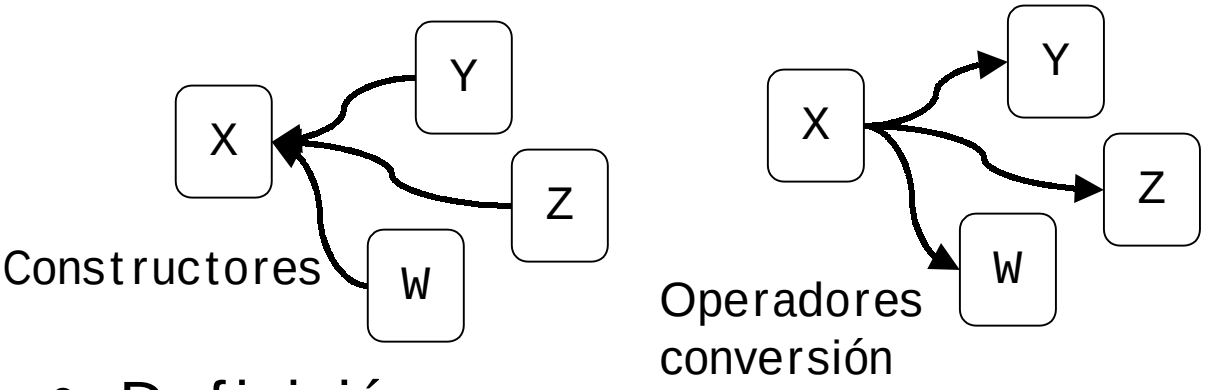
1. Construye un objeto Y a partir de un objeto X.
2. Donde se requiere un objeto de la clase Y se puede suministrar un objeto de la clase X, pues el objeto de la clase X es convertido en un objeto de la clase Y.

Ejemplo. Conversión de clases

```
class complejo
{ float real;
  float imaginaria;
public:
  complejo(float a,float b)
      { real=a; imaginaria=b; }
  complejo(float a=0.0)
      { real=a; imaginaria=0.0; }
  friend complejo
           operator+(complejo,complejo);
  operator=(complejo);
};

complejo operator+(complejo a,complejo b)
{ complejo c;

c.real=a.real+b.real;
c.imaginaria=a.imaginaria +b.imaginaria;
return c;
}

void main(void)
{ complejo c1;
  complejo c2(1.5,2.3);
  float f ;
c1= c2 + f; //operator+(c2,complejo(f));
c2= f + c1; //operator+(complejo(f),(1);
c1= f + 5;  //c1.operator=(complejo(f+5));
}
```

Operadores de conversión

- Los constructores de una clase X:
  - permiten definir qué otras clases pueden ser convertidas en la clase X.
- Los operadores de conversión que se definen en una clase X:
  - permiten definir en qué otras clases se pueden convertir la clase x.



- Definición:

```
operator clase() {......};
```

Aunque se devuelve un objeto de la clase "clase", no se pone tipo de retorno.

No se definen parámetros

El nombre de la clase destino, en la que queremos que se puedan convertir los objetos.

Operadores de conversión

```
class racional
{ int numerador;
  int denominador;

public:
  racional(int num,int den)
    { numerador=num; denominador=den; }

  operator complejo()
    { complejo c=(float)numerador /
                 (float)denominador;
      return c;
    }
};


void main(void)
{ complejo c1, c2;
  racional r1;

c1=r1+c2;
}
```

r1 es convertido en un complejo, pudiendo así llamar a operator+(complejo,complejo);

Operadores de conversión

```
void main(void)
{ complejo c1, c2;
  racional r1;

c1=r1+c2;
}
```

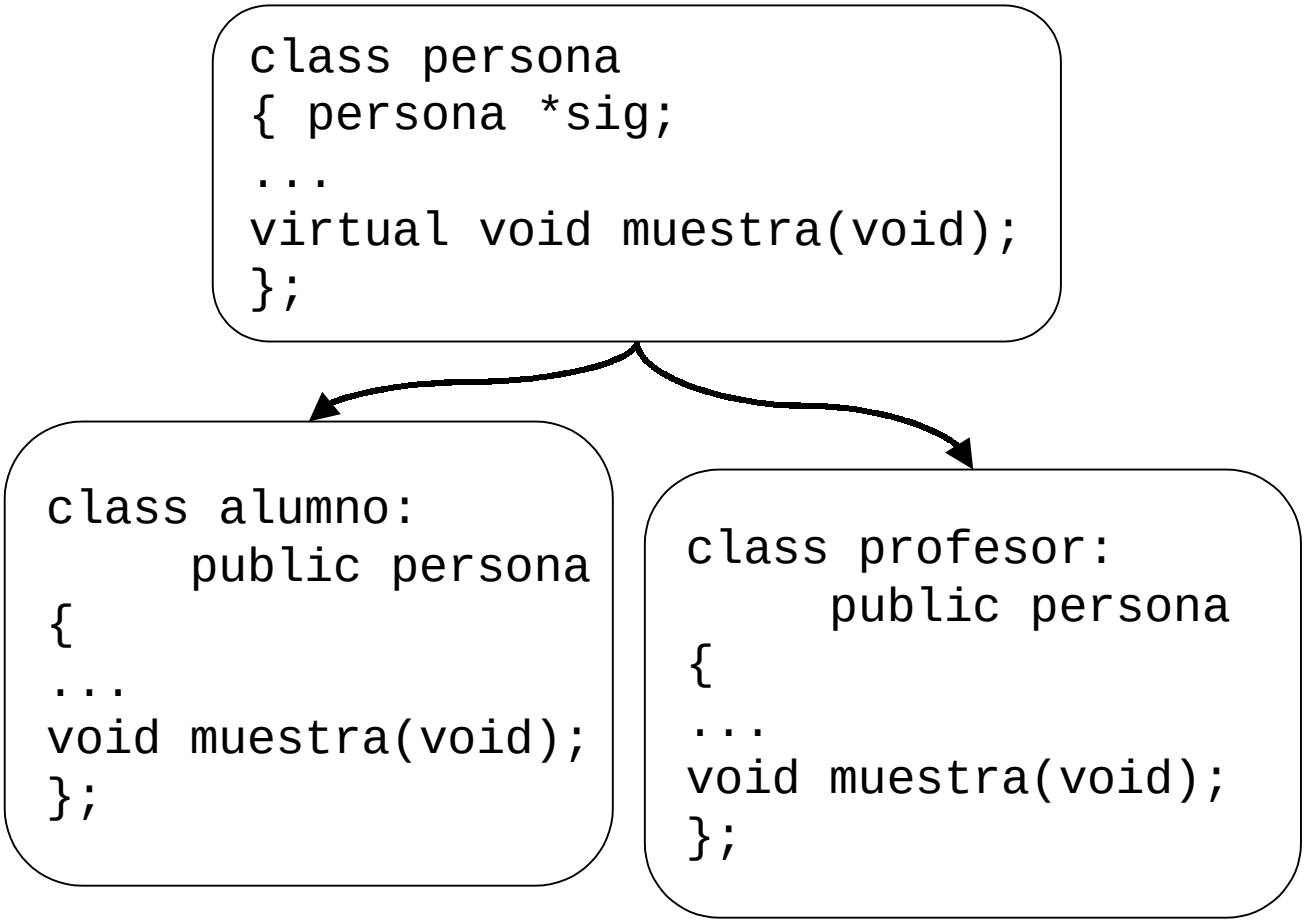
El compilador no sabría a quién llamar.

- Tendriamos una ambigüedad si tenemos definido a la vez:

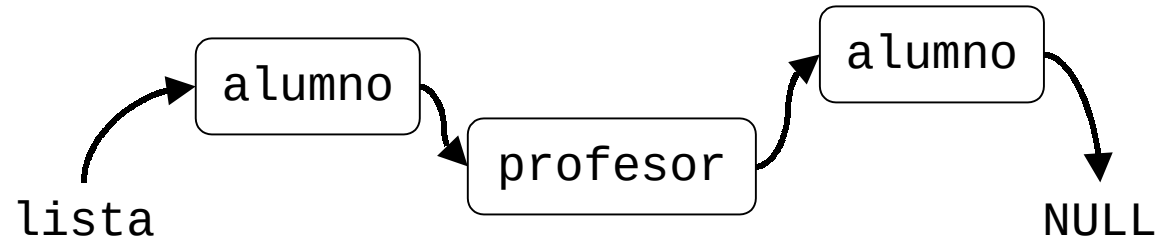
```
operator+(complejo,complejo);
operator+(racional,complejo);

operator complejo();
complejo::complejo(racional);
```

El compilador no sabría si:
- convertir el racional en complejo con el constructor de la clase complejo o
- con el operador de conversión de la clase racional.

- Funciones virtuales:



  - Son una clase de sobrecarga de funciones, pero con la diferencia de:
    - Sobrecarga de funciones: se decide que función se va a ejecutar en tiempo de compilación.
    - Funciones virtuales: se decide en tiempo de ejecución (poliformismo en tiempo de ejecución).




```
persona *lista,*p;
...
for(p=lista; p!=NULL; p=lista->sig)
  p->muestra();
```

- Se ejecutará la función `muestra()` de `alumno` o de `profesor` dependiendo de la clase de objeto a la que apunte `p` (ligadura dinámica, en tiempo de ejecución).
- La función `persona::muestra()` no se ejecuta porque es `virtual` y ha sido substituida en cada clase derivada.
- Si no se hubiera definido como `virtual` la función `muestra()` en la clase `persona`:
  - se ejecutaría siempre `persona::muestra()` en el ejemplo anterior.

Funciones virtuales

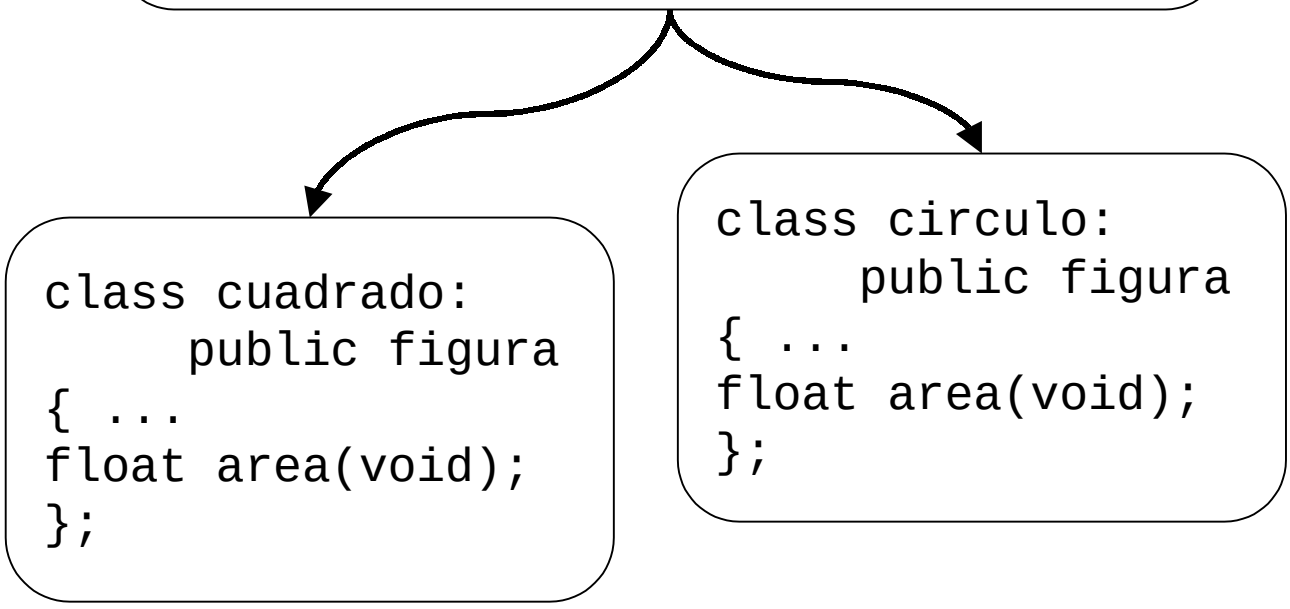
- Cuando se define una función virtual:
  - El prototipo debe ser el mismo en las clases base y derivadas.
- Una función definida como virtual en una clase base:
  - es virtual en todas sus clases derivadas, sea cual sea el nivel de derivación.
  - Puede ser re-definida o no en las clases derivada.

```
class base
{ virtual void f(void);
};

class d1: public base
{ void f(void);
};

class d11: public d1
{ void f(void);
};

base *b;
d11 d;
b=&d ; b->f();
```

No es necesario volver a escribir la palabra virtual.

Se llama a d11:f();

```
class figura
{ ...
float area(void) { cout << “Nada”; }
};
```



```
class cuadrado:
     public figura
{ ...
float area(void);
};
```

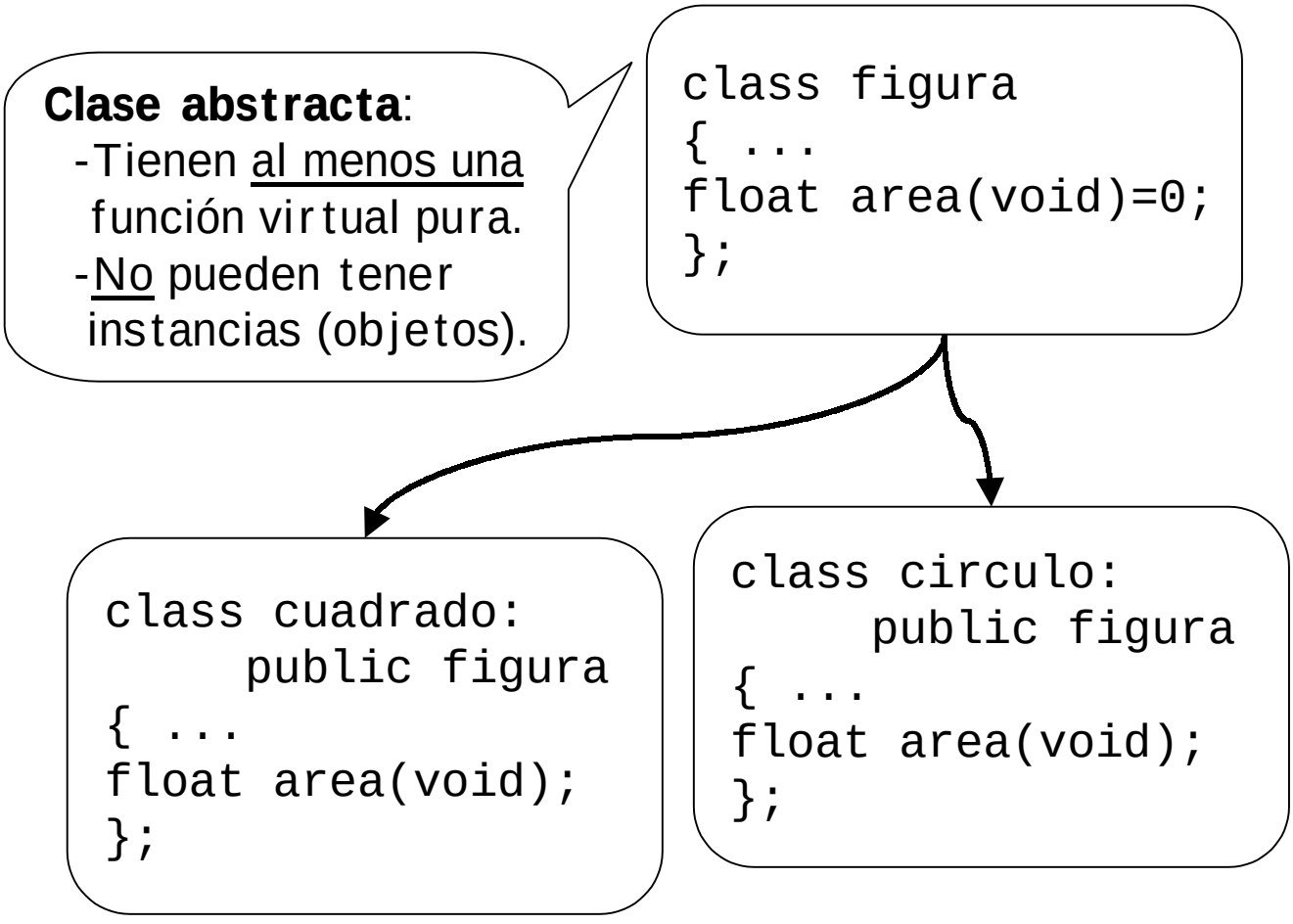
```
class circulo:
     public figura
{ ...
float area(void);
};
```

- No tiene mucho sentido:
  - definir código a la función `figura:area()`.
  - existan objetos de la clase `figura`.
- Podemos definir la clase `figura` como una clase abstracta.

- Funciones virtuales puras:

```
virtual tipo
   nombre_funcion(lista_de_parametros) = 0;
```

  - todas las clases derivadas están obligadas a re-definirla.



Destructor virtual

```
class color  //clase base
{ ...
  virtual ~color();
};

class rojo: public color
{ ...
  ~rojo();
}

class rojo_oscuro: public rojo
{ ...
  ~rojo_oscuro();
}

void main(void)
{ color *paleta[3];

paleta[0]=new rojo;
paleta[1]=new rojo_oscuro;
paleta[2]=new color;

for(int i=0; i<3; i++)
  delete paleta[i];

}
```

Punteros a clase base “color”.

Punteros de clase base apuntando a objetos de clases derivadas.

Si el destructor no se define como virtual, solo se ejecutará `color::~color()` para los 3 objetos.