

Programación orientada a objetos: C++

Paradigmas de la Programación

FaMAF – UNC 2021

capítulo 12

basado en filminas de Vitaly Shmatikov

historia

- C++ es la extensión orientada a objetos de C
- diseñado por Bjarne Stroustrup en Bell Labs
 - motivado por su interés en simulación
 - algunas extensiones previas basadas en Simula
 - se añadieron características incrementalmente: clases, templates, excepciones, herencia múltiple, tests de tipado...

objetivos de diseño

- proveer características de orientación a objetos en un lenguaje tipo C, sin renunciar a la eficiencia
 - retrocompatible con C
 - mejorando el chequeo de tipos estático
 - con abstracción de datos, objetos, clases
 - priorizando código compilado
- principio importante: si no se usa una característica orientada a objetos, el código compilado debería ser igual de eficiente que C sin orientación a objetos

qué tan bien les salió?

- muy popular
- dadas las restricciones y los objetivos, muy buen diseño
- pero un diseño muy complicado:
 - muchas características con interacciones complejas, difíciles de predecir a partir de los principios básicos
 - la mayoría de usuarios serios usan sólo un subconjunto del lenguaje, porque el lenguaje completo es complejo e impredecible
 - muchas propiedades dependientes de implementación

restricciones importantes

- C tiene un modelo de máquina específico (no abstracto), porque tiene acceso al bajo nivel (por herencia de BCPL)
- no hay recolección de basura, por eficiencia, así que hay que manejar la memoria de objetos explícitamente
- las variables locales se guardan en los activation records
 - los objetos se tratan como generalizaciones de **structs**
 - se los puede alojar en el stack y tratarlos como **l-valores**
 - el programador puede acceder a la diferencia entre stack y heap

añadidos no orientados a objetos

- **templates de función** (programación genérica), en la [STL](#)
- **pasaje por referencia**
- **sobrecarga** definida por el usuario
- tipo **booleano**

sistema de objetos de C++

- clases
- objetos
 - con consulta dinámica (*dynamic lookup*) de funciones virtuales
- herencia
 - simple y múltiple
 - clases base públicas y privadas
- subtipado
 - ligado al mecanismo de herencia
- encapsulación

buenas decisiones

- niveles de visibilidad
 - Public: visible en todos lados
 - Protected: en las declaraciones de clase y sus subclases
 - Private: visible solamente en la clase donde se declara
- se permite herencia sin subtipado
 - clases base privadas y protegidas

áreas problemáticas

- Casts
 - irregular: a veces se fuerzan y a veces no
- sin garbage collection
- los objetos se alojan en el stack
 - mejor eficiencia, interacción con las excepciones
 - pero la asignación funciona mal, posiblemente con punteros colgantes
- sobrecarga
 - demasiados mecanismos de selección de código?
- herencia múltiple
 - como se busca eficiencia, el comportamiento es complicado

clase ejemplo: punto

```
class Pt {
```

```
    public:
```

```
        Pt(int xv);
```

```
        Pt(Pt* pv);
```

```
        int getX();
```

```
        virtual void move(int dx);
```

```
    protected:
```

```
        void setX(int xv);
```

```
    private:
```

```
        int x;
```

```
};
```

constructor sobrecargado

acceso público de lectura a datos
privados

función virtual

acceso de escritura protegido

datos privados

SubClases de punto
van a poder reescribir
este metodo si lo
quieren

funciones virtuales

- funciones virtuales
 - se acceden a través de un puntero en el objeto
 - se pueden redefinir en subclases derivadas
 - la función exacta que se llama se determina dinámicamente
- las funciones no virtuales son funciones comunes: no se pueden redefinir pero se pueden sobrecargar
- las funciones son virtuales si se declaran explícitamente o se heredan como virtuales, si no, son no-virtuales
- se paga overhead sólo si se usan funciones virtuales

```
class Animal {
    void /*non-virtual*/ move(void) {
        std::cout << "Este animal se mueve de
alguna forma" << std::endl;
    }
    virtual void eat(void) {}
};

class Llama : public Animal {
    // la función move() se hereda pero no se
puede modificar
    void eat(void) {
        std::cout << "Las llamas comen pasto!"
<< std::endl;
    }
};
```

ejemplo de clase derivada: punto coloreado

```
class ColorPt: public Pt {  
    public:  
        ColorPt(int xv,int cv);  
        ColorPt(Pt* pv,int cv);  
        ColorPt(ColorPt* cp);  
        int getColor();  
        virtual void move(int dx);  
        virtual void darken(int tint);  
    protected:  
        void setColor(int cv);  
    private:  
        int color; };
```

la clase pública de base es el supertipo

} constructor sobrecargado

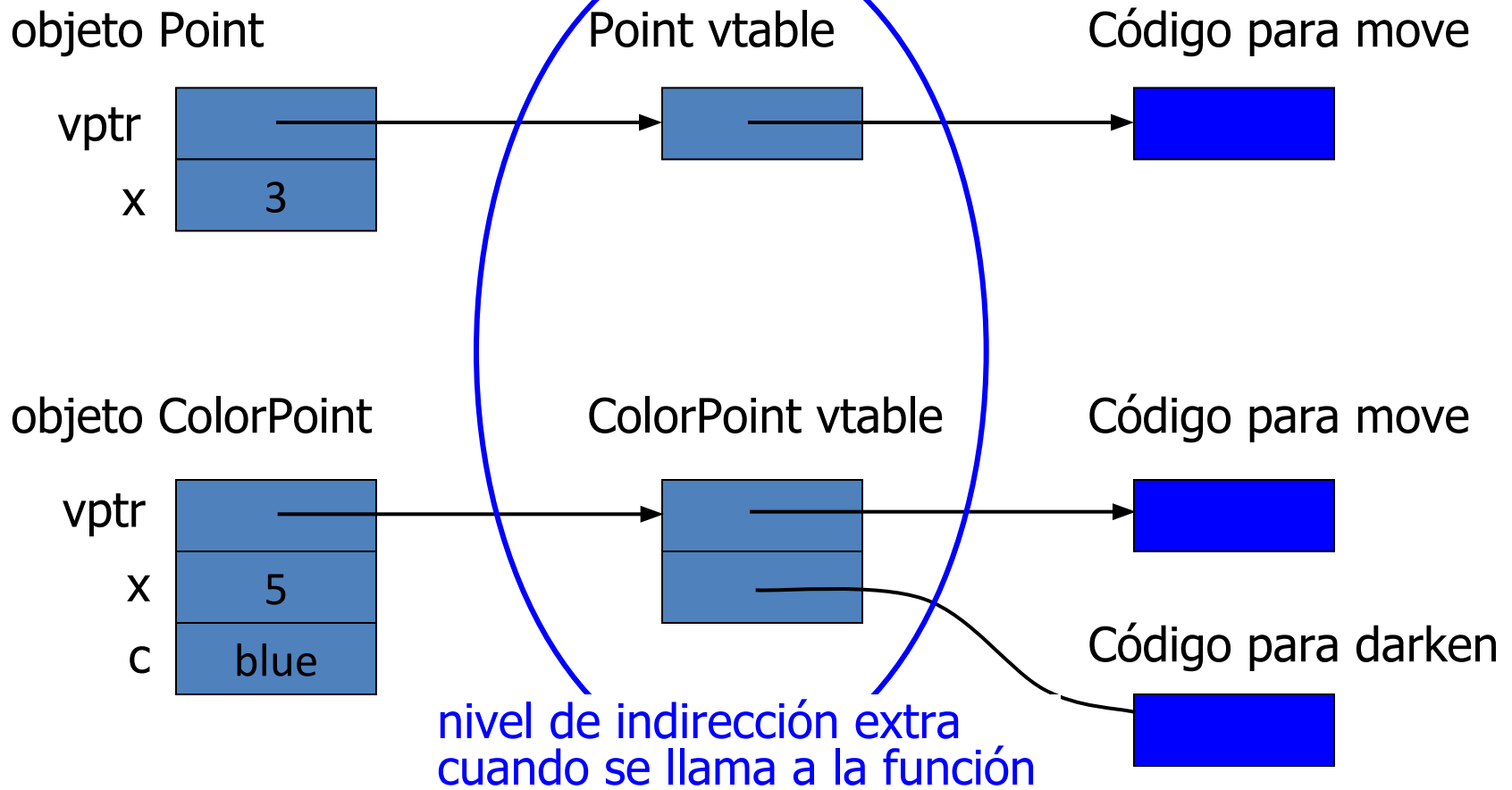
función no-virtual

} funciones virtuales

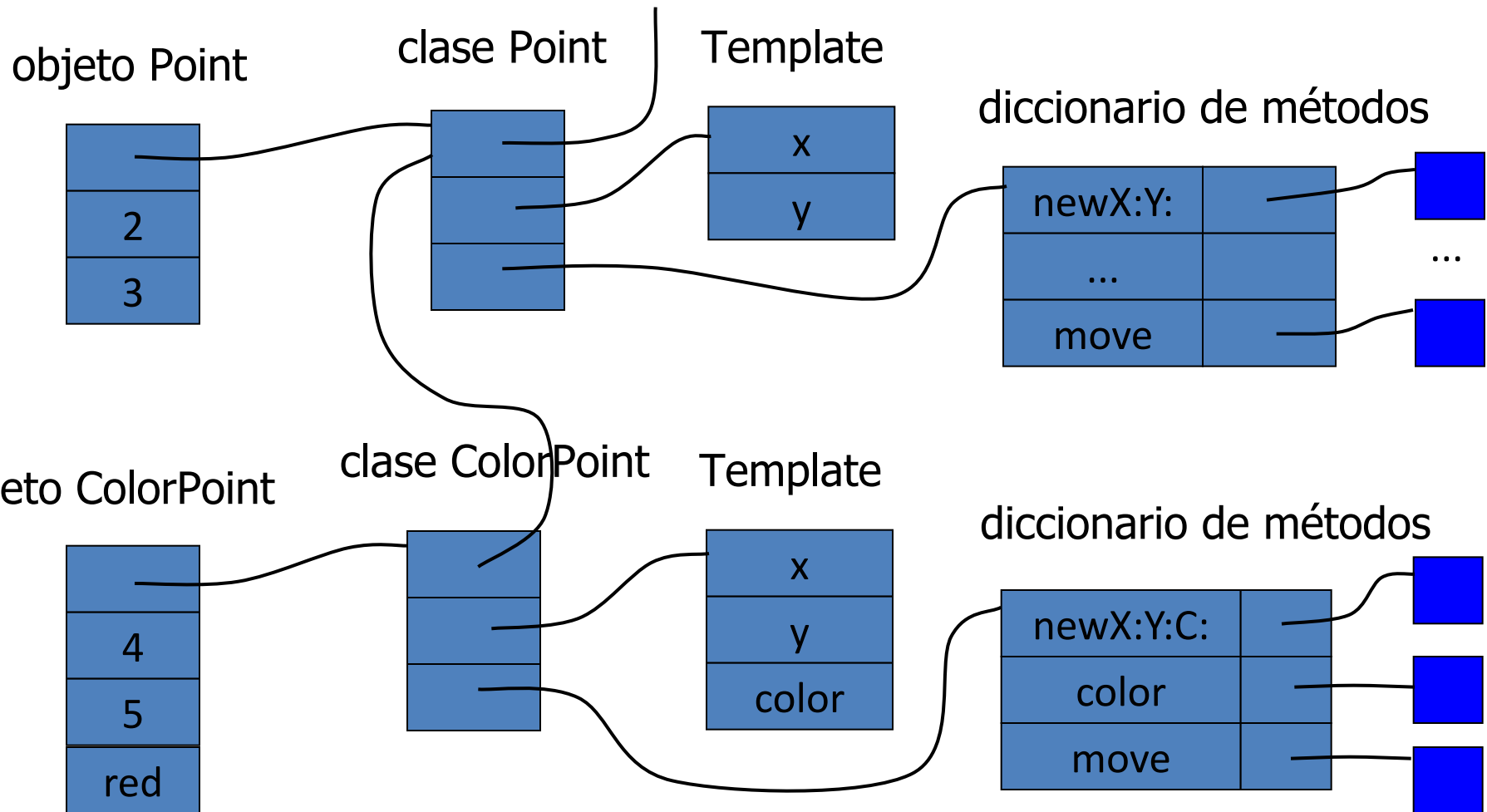
acceso de escritura Protected

datos Private

representación en tiempo de ejecución



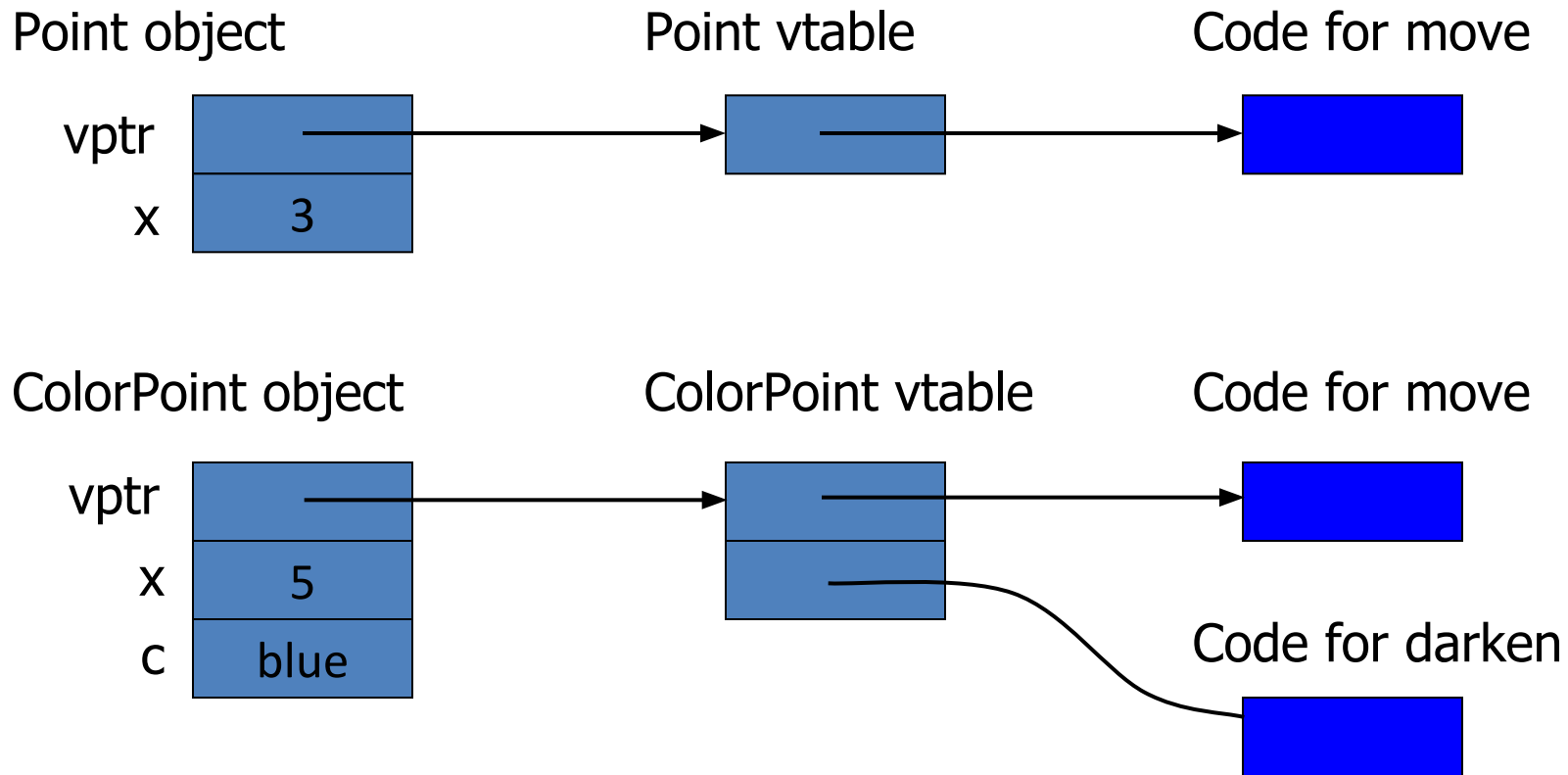
comparación con Smalltalk



por qué el lookup en C++ es más simple?

- Smalltalk no tiene sistema de tipos estático
 - el código `p message:params` puede referirse a cualquier objeto
 - necesitamos encontrar un método que use el puntero del objeto
 - diferentes clases ponen los métodos en diferentes lugares en el diccionario de métodos
- C++ le dá al compilador una superclase
 - el offset de los datos y los punteros a funciones son los mismos en la subclase y la superclase, se conocen en tiempo de compilación
 - el código `p->move(x)` compila al equivalente de `*(p->vptr[0])(p,x)` si `move` es la primera función en la vtable

métodos de consulta (1)

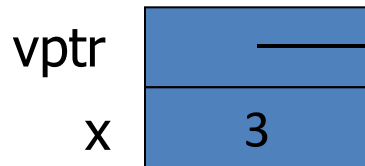


```
Point p = new Pt(3);
```

```
p->move(2);           // Compiles to equivalent of (*(p->vptr[0]))(p,2)
```

métodos de consulta

Point object



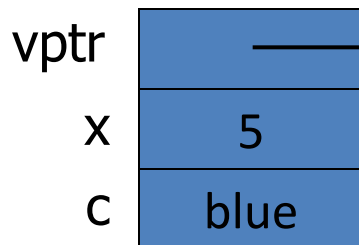
Point vtable



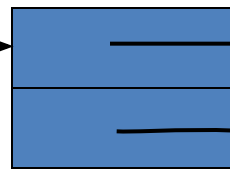
Code for move



ColorPoint object



ColorPoint vtable



Code for move



Code for darken



llamadas a funciones virtuales

- una función puede llamar a otra

```
class A {  
    public:  
        virtual int f (int x);  
        virtual int g (int y);  
};  
int A::f(int x) { ... g(i) ...; }  
int A::g(int y) { ... f(j) ...; }
```

- cómo sabemos que `f` llama a la `g` adecuada? Si `g` se redefine en la clase derivada `B`, entonces la `f` que se hereda tiene que llamar a `B::g`

el puntero "this"

- el código se compila de forma que la función miembro toma al objeto mismo como primer argumento

código `int A::f(int x) { ... g(i) ...; }`

compilado `int A::f(A *this, int x) { ...
this->g(i) ...; }`

- el puntero "this" se puede usar en la función miembro, para devolver el puntero del objeto, pasar el puntero del objeto a otra función, etc.
- igual al "self" de Smalltalk

funciones no virtuales

el código para funciones no virtuales se encuentra igual que para las funciones comunes

- el compilador genera el código de la función y le asigna una dirección
- la dirección se ubica en la tabla de símbolos
- en el lugar de llamada, se obtiene la dirección de la tabla y se ubica en el código compilado
- pero en el caso de clases aplican algunas reglas especiales sobre alcance
- la sobrecarga se resuelve en tiempo de compilación, a diferencia de la consulta de una función virtual en tiempo de ejecución

reglas de alcance en C++

calificadores de alcance: `::`, `->`, y `.`

`class::member`, `ptr->member`, `object.member`

- global (objeto, función, enumerador, tipo):
nombre fuera de una función o clase no prefijado por `::` unario y no calificado
- alcance de clase: nombre después de `X::`, `ptr->` o `obj.`, se refiere a un miembro de la clase `X` o a la clase base de `X`, asumiendo `ptr` es un puntero a la clase `X` y `obj` es un objeto de la clase `X`

funciones virtuales vs. sobrecargadas

```
class parent { public:
    void printclass() {printf("p ");};
    virtual void printvirtual() {printf("p ");};    };
class child : public parent { public:
    void printclass() {printf("c ");};
    virtual void printvirtual() {printf("c ");};    };
main() {
    parent p;  child c;  parent *q;
    p.printclass(); p.printvirtual();
    c.printclass(); c.printvirtual();
    q = &p;  q->printclass(); q->printvirtual();
    q = &c;  q->printclass(); q->printvirtual();
}
```

Output: p p c c p p p c

subtipado

- subtipado en principio: A es un subtipo de B (**A** **<:B**) si todo objeto A se puede usar en un contexto en el que se necesitaba B sin errores de tipo
ejemplo:

```
Point:      int getX();  
            void move(int);  
  
ColorPoint: int getX();  
            int getColor();  
            void move(int);  
            void darken(int tint);
```

- en C++: A es un subtipo de B si la clase A tiene como clase base pública a B
 - esto es más débil de lo que sería necesario

no hay tipado sin herencia

```
class Point {  
    public:  
        int getX();  
        void move(int);  
    protected:    ...  
    private:      ...  
};
```

```
class ColorPoint {  
    public:  
        int getX();  
        void move(int);  
        int getColor();  
        void darken(int);  
    protected:    ...  
    private:      ...  
};
```

C++ no trata este ColorPoint como subtipo de Point, pero Smalltalk sí lo haría

por qué esta decisión de diseño?

- el código depende sólo de la interfaz pública
 - en principio, si la interfaz de ColorPoint contiene a la interfaz de Point los clientes podrían usar ColorPoint en lugar de Point (como en Smalltalk)
 - pero el offset en la tabla de funciones virtuales puede ser distinta, y de esta forma perder eficiencia (como en Smalltalk)
- si no funciona ligada a la herencia, el subtipado lleva a pérdida de eficiencia
- también por encapsulación: el subtipado basado en herencia se preserva si hacemos modificaciones en la clase base

subtipado de funciones

- subtipado en principio: A es un subtipo de B ($A <: B$) si una expresión A se puede usar en todo contexto en el que se requiere una expresión B
- subtipado para resultados de función
 - si A es un subtipo de B , entonces $C \rightarrow A$ es un subtipo de $C \rightarrow B$
 - **covariante**: $A <: B$ implica $F(A) <: F(B)$
- subtipado para argumentos de función
 - si $A <: B$, entonces $B \rightarrow C <: A \rightarrow C$
 - **contravariante**: $A <: B$ implica $F(B) <: F(A)$

para saber más...

... sobre covariación y contravariación:

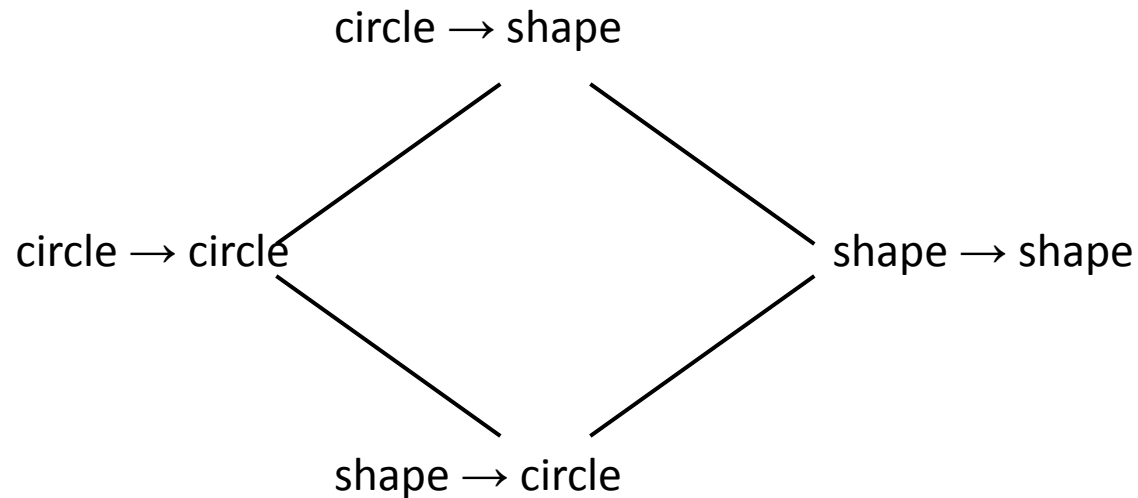
[http://en.wikipedia.org/wiki/Covariance_and_contravariance_\(computer_science\)](http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science))

es seguro sustituir una función `f` en el lugar de una función `g` si `f` acepta un tipo más general de argumentos y devuelve un tipo más específico que `g` (el constructor de tipo `->` es contravariante en el tipo del input y covariante en el tipo del output).

Por ejemplo, si tenemos una función `gato -> animal` la podemos sustituir por `gato -> gato` o por `animal -> animal`, pero no podríamos si en lugar de `gato -> animal` tuviéramos `animal -> gato`.

ejemplos

- si `circle <: shape`, entonces



los compiladores de C++ reconocen sólo algunas formas de subtipado de función

subtipado con funciones

```
class Point {  
    public:  
        int getX();  
        virtual Point *move(int);  
    protected: ...  
    private: ...  
};
```

```
class ColorPoint: public Point {  
    public:  
        int getX();  
        int getColor();  
        ColorPoint *move(int);  
        void darken(int);  
    protected: ...  
    private: ...  
};
```

heredado, pero lo repetimos acá por claridad

- en principio, podríamos tener `ColorPoint <: Point`
- en la práctica, algunos compiladores lo permiten y otros no

clases abstractas

- una clase abstracta es una clase sin implementación completa
- se declara con `=0` (what a great syntax! 😊)
- útil porque puede tener clases derivadas
 - como el subtipado se sigue de la herencia en C++, se pueden usar las clases abstractas para construir jerarquías de subtipos
- establece la disposición de la vtable (tabla de funciones virtuales)

```
class Vehicle {
public:
    explicit
    Vehicle( int topSpeed ) : m_topSpeed( topSpeed )
    {}
    int TopSpeed() const {
        return m_topSpeed;
    }
    virtual void Save( std::ostream& ) const = 0;
private:
    int m_topSpeed;
};

class WheeledLandVehicle : public Vehicle {
public:
    WheeledLandVehicle( int topSpeed, int numberOfWheels )
    : Vehicle( topSpeed ), m_numberOfWheels( numberOfWheels )
    {}
    int NumberOfWheels() const {
        return m_numberOfWheels;
    }
    void Save( std::ostream& ) const;
private:
    int m_numberOfWheels;
};
```



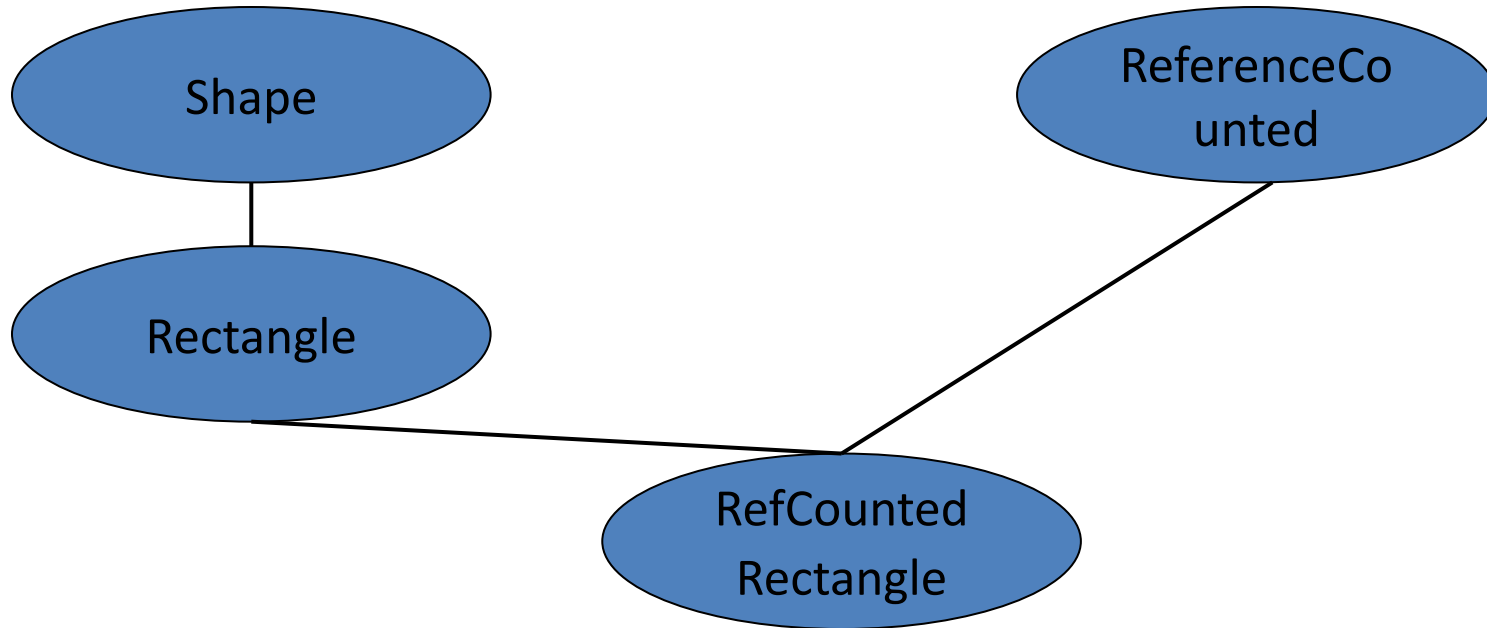
```
class DrawableObject
{
    public:
        virtual void Draw(GraphicalDrawingBoard&) const = 0;
//draw to GraphicalDrawingBoard };

class Triangle : public DrawableObject
{
    public:
        void Draw(GraphicalDrawingBoard&) const; //
};

class Rectangle : public DrawableObject ....

typedef std::list<DrawableObject*> DrawableList;
```

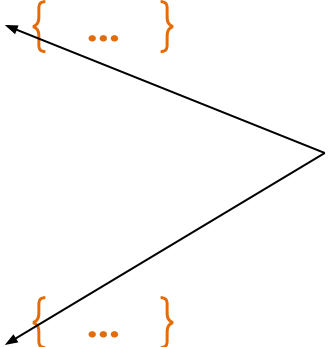
herencia múltiple



se heredan funcionalidades independientes de clases independientes

problema: choques de nombre (*name clashes*)

```
class A {  
    public:  
        void virtual f() { ... }  
};  
class B {  
    public:  
        void virtual f() { ... }  
};  
class C : public A, public B { ... };  
...  
C* p;  
p->f();    // error
```



el mismo
nombre en
dos clases
base!

cómo resolver choques de nombre

- resolución implícita: con reglas arbitrarias
- resolución explícita: el programador debe resolver los conflictos explícitamente
 - ← la que usa C++
- no permitida

resolución explícita de choques de nombre

- reescribir la clase C para llamar a A::f explícitamente

```
class C : public A, public B {  
    public:  
        void virtual f( ) {  
            A::f( ) ; // llama A::f(), no B::f();  
        }  
}
```

- elimina ambigüedad
- preserva la dependencia de A
 - los cambios a A::f cambiarán C::f

vtable para herencia múltiple

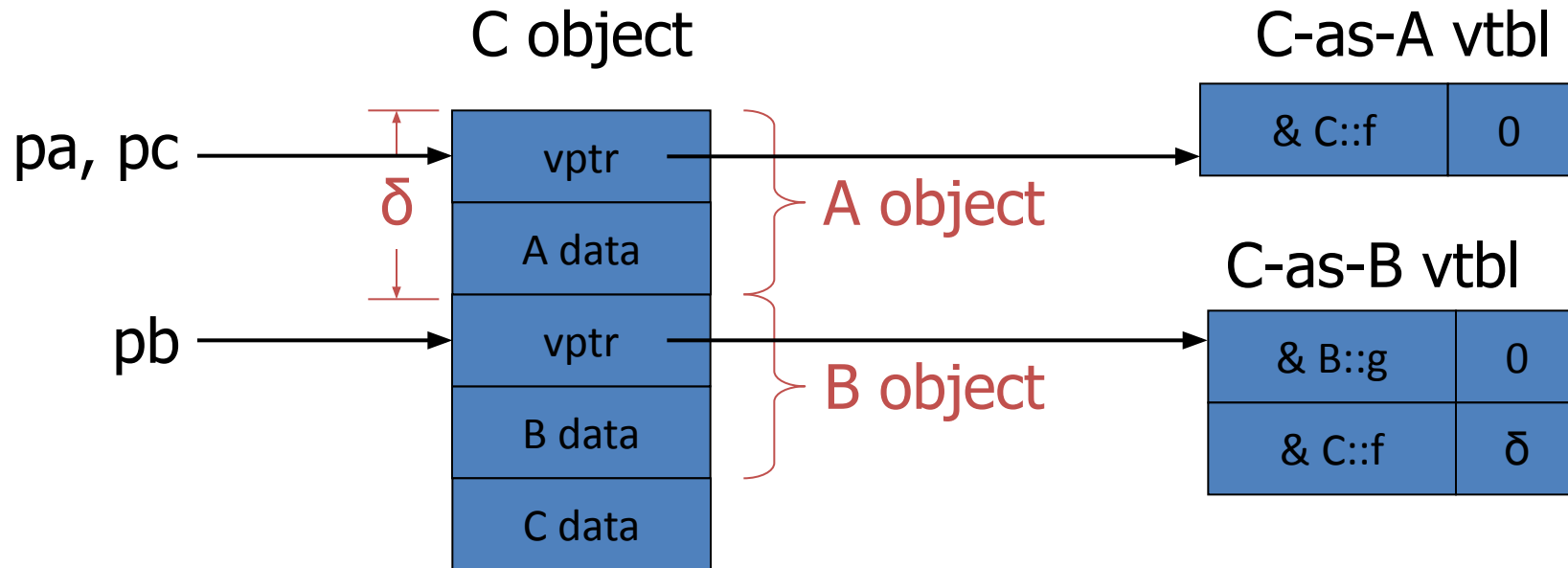
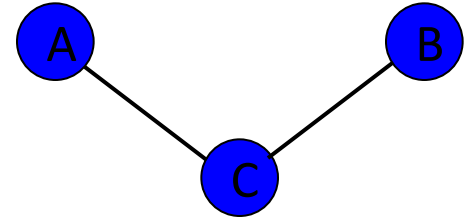
```
class A {  
    public:  
        int x;  
        virtual void f();  
};  
class B {  
    public:  
        int y;  
        virtual void g();  
        virtual void f();  
};
```

```
class C: public A, public B {  
    public:  
        int z;  
        virtual void f();  
};
```

```
C *pc = new C;  
B *pb = pc;  
A *pa = pc;
```

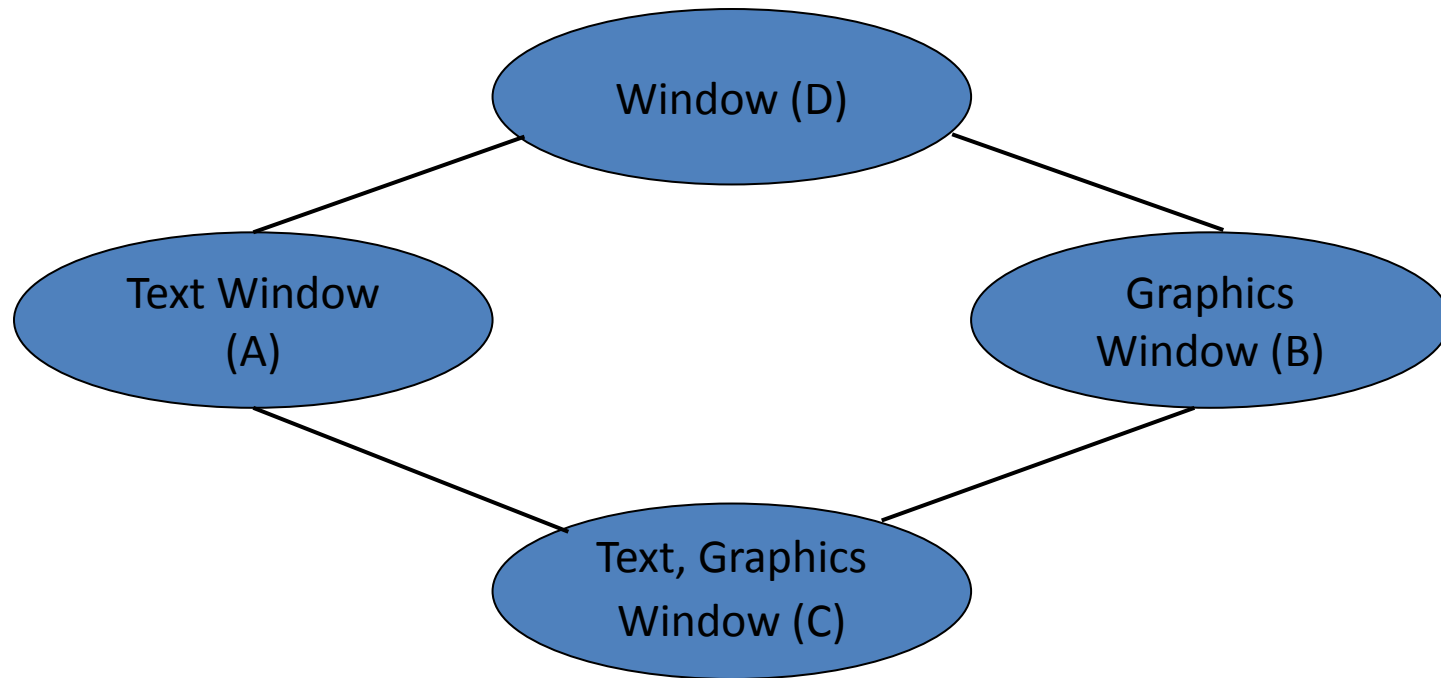
tres punteros al mismo objeto, pero
con diferentes tipos estáticos

esquema de objeto



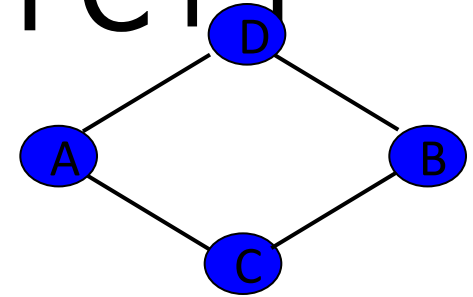
- el offset δ en la vtbl se usa en la llamada a pb->f, porque **C::f** se puede referir a datos de A que están arriba del puntero pb
- la llamada a pc->g puede proceder como C-as-B vtbl

herencia múltiple “diamante”



- se hereda la interfaz o la implementación dos veces?
- qué pasa si las definiciones tienen conflicto?

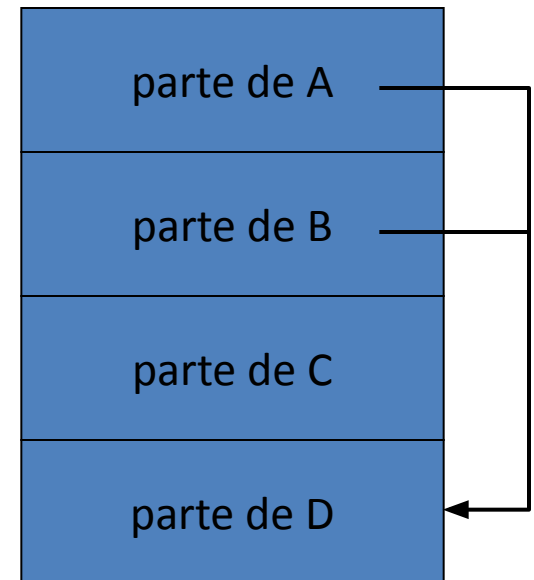
herencia diamante en C++



- problema: clases base estándares
 - los miembros de D ocurren dos veces en C
- solución: clases base virtuales

```
class A : public virtual D { ... }
```

 - se evita el duplicado de los miembros de la clase base
 - se requieren punteros adicionales para compartir la parte D de A y B



la herencia múltiple en C++ es complicada en parte porque quiere mantener lookup eficiente

```
class storable // clase base heredada por transmitter y receiver
{
    public:
        storable(const char*);
        virtual void read();
        virtual void write();
        virtual ~storable();
    private: ....}
```

```
class transmitter: public storable
{
    public:
        void write(); }
```

```
class receiver: public storable
{
    public:
        void read(); }
```

```
class radio: public transmitter, public receiver
{
    public:
        void read(); }
```

```
class transmitter: public virtual storable
{
    public:
    void read();
    ...
}
```

```
class receiver: public virtual storable
{
    public:
    void read();
    ...
}
```

Si usamos herencia virtual, garantizamos tener una sola instancia de la clase base común, no hay ambigüedad.

resumen de C++

- objetos
 - creados por clases
 - contienen datos del miembro y un puntero a la clase
- clases: tabla de funciones virtuales
- herencia
 - clases base públicas y privadas, herencia múltiple
- subtipado: sólo con clases base públicas
- encapsulación
 - un miembro se puede declarar público, privado o protegido
 - la inicialización de los objetos se puede forzar parcialmente