Programación en C++ (7) : herencia/composición; polimorfismo

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009

Outline

Outline

Reuso de código existente

Hay dos mecanismos para construir clases/estructuras incluyendo otras clases de una manera o otra :

- Pegar objetos de otras clases dentro de una nueva clase : Composición.
- Decir que una nueva clase es una especialización de una clase existente y que tiene "unas cositas" mas que la clase general : Herencia.

Son mecanismos esenciales para evitar tener que re-codar miles de cosas.

Composición

Una primera clase simple :

```
class Size {
        int width;
        int height;
public:
        Size(int w=0,int h=0) : width(w),height(h){};
        int getWidth() const {return width;};
        int getHeight() const {return height;};
};
```

Composición

```
class Image {
        int a;
        char *data;
public:
        Size s;
        ...
};
```

El aceso a la anchura :

```
Image img(100,200);
int w = img.s.getWidth();
```

O (mejor) : re-defino un método $\mathrm{getWidth}()$ al interior de la clase Image y pongo el objeto Size $\mathrm{private}$

Composición

```
class Image {
        int a;
        char *data;
        Size s;
public:
        int getWidth() const {
                return s.getWidth();
        }
        ...
};
```

Notar que dentro de los métodos de Image no tengo derechos particulares en cuanto a los elementos de Size (para que sea el caso, necesitaría una friendship). Notar también el uso de const.

Herencia

```
class SquareImage : public Image {
        int a;
public :
        SquareImage(int aval) { a=aval;}
        int getA() const {return a};
};
```

Imaginar que, en memoria, los objetos SquareImage vienen con, primero, todos los elementos de Image y luego los suyos.

Herencia

Remarcas importantes :

- Los elementos private de Image siguen private en la clase que herede : pensar que el que hace esta herencia puede ser simplemente un "usuario".

```
SquareImage::SquareImage(int aval) {
        a=aval;
        int w = Image::s.width;   // NO
        int w = getWidth(); // OK;
};
```

- La palabra public al momento de declarar la herencia se refiere a la visibilidad afuera de los elementos/métodos de Image : los elementos public de Image siguen public en SquareImage

```
SquareImage simg;
int w = simg.getWidth();
```

Herencia

Remarcas importantes :

- Se puede redefinir elementos, tanto datos como métodos, y en este caso se llama (en la clase que herede) por default la version redefinida :

```
class SquareImage {
        void someMethod() {
                getA(); // El de SquareImage
                Image::getA(); // El de Image
        }
};
```

Herencia y composición : inicialización

Las clases compuestas o las clases que hereden no tienen acceso directo a todos los elementos, entonces queda el problema de inicialización. Para eso, usar los constructores :

```
class SquareImage {
public:
        SquareImage(int w=0) : Image(w,w), a(0) {
        }
};
```

Se puede usar esta misma sintaxis para datos miembros (de cualquier tipo : objetos o tipos básicos) o en caso de herencia. La idea es que todo elemento debe de ser inicializado antes de la primera llave del constructor.

Herencia y composición : inicialización

Intentar usarlo lo más seguido posible, incluso para tipos basicos y aun combinando herencia y composición :

```
class SquareImage : public Image {
int a;
OtraClase truc;
public:
        SquareImage(int w=0) : Image(w,w),truc(),a(0)
        }
};
```

Herencia y composición : destrucción

El destructor de la clase compleja invoca por sí mismo todos los destructores necesarios, en el buen orden.

```
class Class1 {
public:
  Class1() {cout << "Cons. Class 1" << endl;}
  ~Class1() {cout << "Des. Class 1" << endl;}};
class Class2 {
public:
  Class2() {cout << "Cons. Class 2" << endl;}
  ~Class2() {cout << "Des. Class 2" << endl;}};
class Class3 : public Class2 {
public:
  Class3() : Class2() {cout << "Cons. Class 3" << end
  ~Class3() {cout << "Des. Class 3" << endl;}};
```

Herencia y composición : destrucción, orden

```
class Class4 : public Class1 {
  Class3 obj;
  Class2 obj2;
public:
  Class4() : Class1(),obj2(),obj() {cout << "Cons. Cl
  ~Class4() {cout << "Des. Class 4" << endl;}
};
int main() {
  Class4 obj4;
}
```

Herencia y composición : destrucción, orden

```
Cons. Class 1
Cons. Class 2
Cons. Class 3
Cons. Class 2
Cons. Class 4
Des. Class 4
Des. Class 2
Des. Class 3
Des. Class 2
Des. Class 1
```

Notar que el orden depende sólo del orden en la declaración, no de la lista de constructores precediendo el cuerpo del constructor.

Métodos homónimos de la clase madre

¿Qué pasa con un método homónimo a uno de la clase madre

```
class Class1 {
public:
  int metodo() {}
  int metodo(Class1 &a) {}
};
class Class2 : Class1 {
public:
  int metodo(int a) {}
};
int main() {
  Class2 obj2; Class1 obj1;
  obj2.metodo();      // NO
  obj2.metodo(obj1); // NO
  obj2.metodo(1);     // OK
}
```

Métodos homónimos de la clase madre

Cada vez que se define un método, en la clase herediendo, de mismo nombre que uno de los métodos de la clase madre, se esconden todas las funciones sobrecargadas de la clase madre. Pero. . . en POO, ese cambio de prototipo no es la manera normal de hacer. Veremos mas tarde otra forma de manejar funciones sobrecargada (polimorfismo).

Métodos heredados. . . o no

Todos los métodos no estan heredados por este mecanismo de herencia :

- Obviamente los constructores tienen que estar definidos en cada clase.
- El operador de asignación $=$, similarmente.
- Notar que el operador de conversión automática es heredado (¿por qué ?)
- Aunque no estén heredadas, el compilador puede crear una nueva forma sintetizada a partir de las que existen en las clases para constructores, operadores de asignación.

Lo sintetizado

- El operador = sólo está sintetizado cuando se trata de objetos de mismo tipo.
- La asignación y el constructor copia no son sintetizados si tu defines unos : cuidado a llamar los de los elementos/clase madre.

Lo sintetizado

```
class Class1 {
public:
  Class1() {cout << "Cons. Class 1" << endl;}
  Class1(const Class1 &otro) {cout << "Consc. Class 1"
class Class4 : public Class1 {
public:
  Class4() {cout << "Cons. Class 4" << endl;}};
int main() {
  Class4 obj1;
  Class4 obj2 = obj1;
}
```

```
Cons. Class 1
Cons. Class 4
Consc. Class 1
```

Lo sintetizado

Para el constructor por default, que defina o no el constructor por default de $\mathrm{Class4}$ no importe : el constructor por default de $\mathrm{Class1}$ estará siempre llamado (en cualquier constructor de $\mathrm{Class4}$, mientras no hay llamada explícita a un constructor de $\mathrm{Class1}$).

Lo sintetizado

Ahora para los constructores por copia y la asignación las cosas son un poco diferentes si tu los defines

```
class Class4 : public Class1 {
public:
  Class4() {cout << "Cons. Class 4" << endl;}
  Class4(const Class4 &otro) {cout << "Consc. Class 4"
};
```

```
Cons. Class 1
Cons. Class 4
Cons. Class 1
Consc. Class 4
```

Y eso no es lo que queremos !

¿Composición o herencia ?

Composición : cuando necesitamos usar esos objetos en la implementación pero que no necesitamos toda la interfaz.

```
class Class5 {
Class1 element1;
Class2 element2;
...
};
```

Como en general la presencia de esos objetos es ligada a la implementación, mejor dejarlos private.

¿ Composición o herencia ?

Sin embargo, puede ser útil a veces dejarlos públicos, cuando construimos una clase con la idea de varias partes constituyendo un todo, y que estas partes pueden tener una interfaz importante en el funcionamiento : la clase Robot, por ejemplo, es constituida por varias partes importantes desde el punto de vista del usuario.
Es poco practico tener que usar métodos de acceso en este caso :

```
class Robot : public Computer {
        Motorization motor;
        Odometry odo;
public:
        Motorization &getMotorization() {
                return motor;
        };
        ...
}
```

¿ Composición o herencia ?

```
class Robot : public Computer {
public:
        Motorization motor;
        Odometry odo;
}
```

Lo mas coherente :

- Si hay una relación de tipo “subconjunto” entonces usar herencia (un Robot es un Computer).
- Si hay relación de parte a todo, usar composición (un Robot tiene Motorization).

¿ Composición o herencia ?

Un caso muy importante en que herencia es apropiado : si queremos usar la interfaz de una clase, sin tener que re-implementar cada cosa :

```
class Robot {
        Computer c;
public:
        void method1() {c.method1();};
        void method2() {c.method2();};
        ...
}
```

Puede ser fastidioso, mejor hacer que Robot herede de Computer y de su interfaz.

Herencia private

Hasta ahora solo vimos casos de herencia public (la palabra que precede la herencia). No obstante, es posible cambiar eso y declarar la herencia private, y eso esconderá (para el exterior) todos los elementos/métodos en la clase que herede. El efecto es más o menos igual a incluir como miembro private una instancia de la clase.

Herencia private

En este caso, se puede hacer uso de unos métodos/elementos públicos en particular especificándoles con la palabra using :

```
class Computer {
public:
        int nProcesador;
        int switchOn();
        int switchOn(int);
...
class Robot : private Computer{
public:
        using Computer::nProcesador;
        using Computer::switchOn; // Hecho publico
...
void someFunction(Robot &r) {
        switch (r.nProcesador) {
```

Elementos protected

Existe un estado intermediario entre private y public, que permite proteger el acceso para afuera excepto para las clases que hereden de esta : protected

```
class Computer {
protected:
        int nProcesador;
...
class Robot : public Computer{
public:
        void someMethod() {
        switch (nProcesador) {
        ...
```

Elementos protected

En general, preferir elementos private, en particular para los datos y métodos utilitarios. Ahora, puede ser practico usar la palabra protected en unos casos, si unos de esos datos pueden ocurrir frecuentemente en la implementación para clases derivadas.

Herencia protected

Muy raro, pero existe : para las clases derivadas y amigas, todos los elementos públicos quedan públicos pero para otras clases u otro tipo de código, están private.

En conclusión...

Los dos mecanismos de creación de nuevas clases a partir de clases existentes realizan plenamente la voluntad de poder re-usar código existente y de hacer evolucionar su código incrementalmente :

- Búsqueda de los *bugs* facilitada (las nuevas partes son fácilmente identificables).
- Pedazos bien separados.
- Desarrollo incremental del código bien controlado, ladrillos de construcción.

Upcasting

Una característica muy importante de la herencia es la relación particular entre la clase madre y la clase hija que hace que una referencia a una instancia de la hija puede "estar vista" sin problema como una referencia a una instancia de la madre :

```
class SquareImage : public Image {
...
};
void fonc(const Image &img) {...};

int main() {
        SquareImage img;
        fonc(img);
        ...
```

Upcasting

Este mecanismo es lógico : todas los métodos (públicos) que están eventualmente llamados dentro de `fonc` desde un objeto de tipo `Image` están disponibles, por definición, para objetos de tipo `SquareImage`.

Por el momento, eso se aplica sólo a referencias y apuntadores.

Upcasting

Ahora veamos lo que pasa con el constructor por copia :

```
class SquareImage : public Image {
        int a;
};
```

Si no hay constructor por copia dentro de la clase derivada, entonces se llamará un constructor por copia sintetizado a partir (1) del constructor por copia de la clase de base y (2) de copia normal para tipos básicos.

Upcasting

Ahora si se define el constructor por copia por sí mismo se necesita llamar al constructor por copia de la clase de base :

```
class SquareImage : public Image {
int a;
public:
SquareImage(const SquareImage &img) : Image(img),
                                      a(other.a) {
...
}
};
```

Y en este caso otra vez llamamos a una función (el constructor) esperando referencia a Image una referencia a SquareImage.

Upcasting

Poder hacer upcasting puede ser la razón que nos hace elegir herencia sobre composición : en unos casos podemos querer manipular referencias a objetos de una clase derivada como si fueran referencias a objetos de la clase "genérica".

Notar que eso no solo vale para funciones sino también para asignaciones de referencias o apuntadores :

```
SquareImage img;
Image *imagePtr = &img;
Image &imageRef = img;
```

Upcasting

¿Qué pasa cuando métodos están definidos en ambas clases (la de base y la derivada) ?

```
SquareImage img;
Image &imageRef = img;
imageRef.someMethod();
```

Pues cual de los dos va a llamar ? A priori, llamará al de Image, porque la única cosa que sabe el compilador (al momento de compilar) es que tiene acá una referencia/apuntador hacia Image !

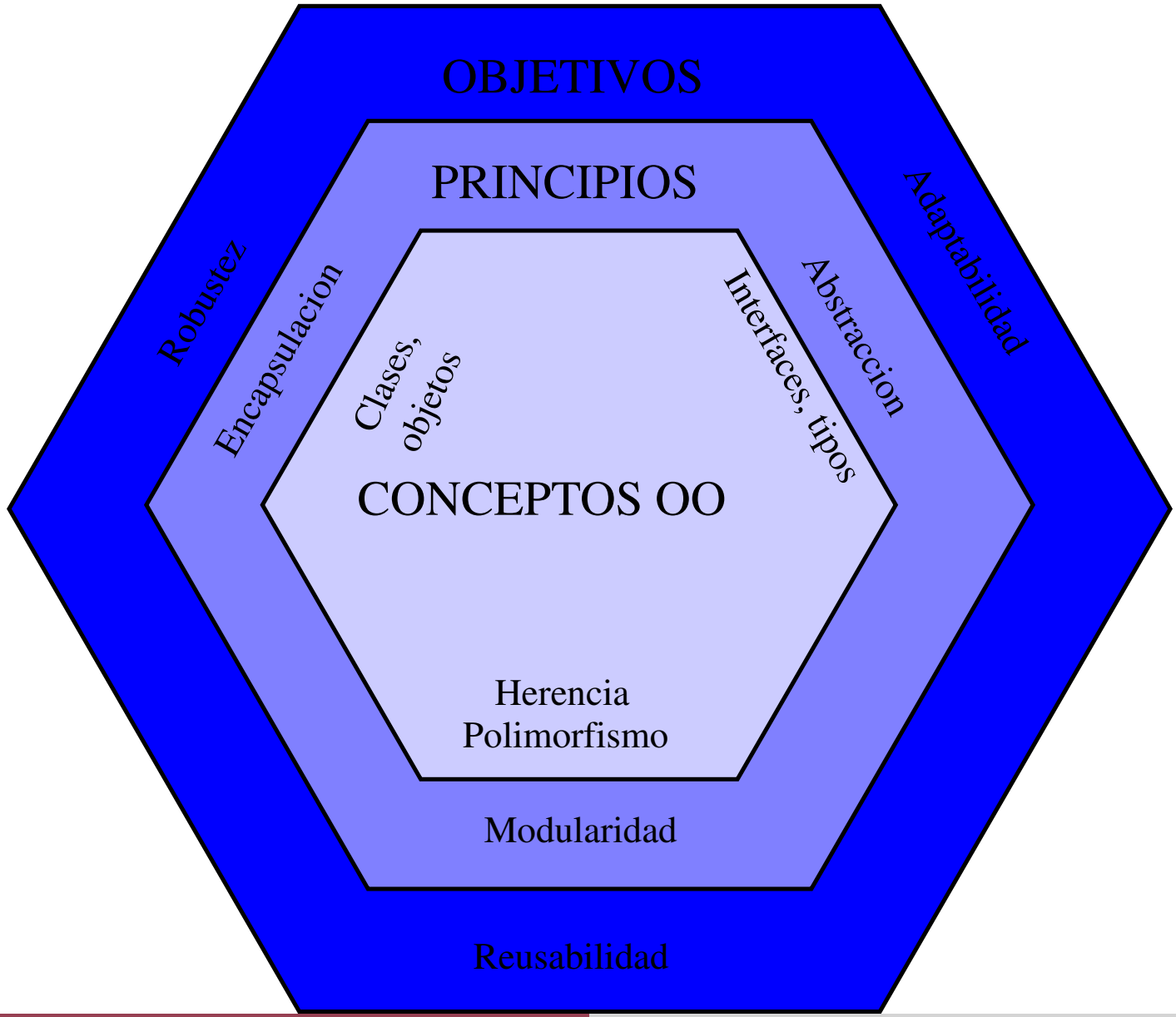
Upcasting

Nos gustaría hacer *upcasting* así pero conservando de una manera u otra la "memoria" de que venimos de una clase derivada particular, y entonces llamar los "buenos" métodos.

En C++ hay un mecanismo que nos permite eso : las funciones virtuales.

Outline

Los tres grandes elementos de C++



Los tres grandes elementos de C++

Lo que hace la diferencia entre C y C++, y que hace de C++ un lenguaje OO :

- Mecanismos de abstracción : separación interfaz/implementación mas forzada que en C.
- Mecanismos de encapsulación : clases, estructuras, funciones propias a clases.
- Mecanismos de modularidad : permite el re-uso del código existente a través de la herencia y del polimorfismo, o sea el uso del mismo código que manipula objetos "genéricos", pero llamando a "versiones" de métodos propias a la especialización del objeto en particular.

Llamadas a funciones

En un código C normal :

```
void fonc(Image *img) {
}
...
fonc(&img1);
```

A la compilación, la llamada a la función se traduce por código objeto con la dirección memoria de la función que llamar, que hará el programa irá a ejectutar las instrucciones de esa función en particular (con los argumentos dados) (*early binding*) : no hay elección posible, es una y una sola función que estará llamada.

Llamadas a funciones

Eso explica el comportamiento del programa al llamar un método después de un *upcasting* :

```
SquareImage img;
Image &imageRef = img;
imageRef.someMethod(); // Llama al de Image, a priori
```

Llamadas a funciones

Ahora, conceptualmente, la única manera de poder llamar al buen someMethod sería poder decidir que cual llamada hacer no al nivel del compilador sino al nivel de la ejecución misma (late binding)

```
SquareImage img;
Image &imageRef = img;
// Podría decidir cual llamar en la ejecucion ?
imageRef.someMethod();
```

Eso implica que el programa pueda verificar de que tipo se esta tratando realmente (un verdadero Image o una de sus derivadas). Hasta ahora no tenemos lo suficiente, pero el C++ provee lo necesario para hacerlo.

Funciones virtuales

Para poder usar esta funcionalidad, se necesita usar, en la clase base, la palabra llave virtual con la declaración de la función sobre la que queremos hacer *late binding*.

```
class Image {
...
public :
        virtual void someMethod();
}
```

Eso hará que toda llamada a someMethod(), dentro de una clase derivada que lo define, a partir de una referencia a Image obtenida por upcasting, llamará la buena versión, la de la derivada !

Funciones virtuales

```
void Image::someMethod() {
        cout << "IMAGE" << endl;
};
void SquareImage::someMethod() {
        cout << "SQIMAGE" << endl;
};
int main() {
    SquareImage img;
    Image &imageRef = img;
    Image *imagePtr = &img;

    imageRef.someMethod();
    imagePtr->someMethod();
}
```

Funciones virtuales

Por magia, obtenemos :

```
SQIMAGE
SQIMAGE
```

mientras manipulamos algo que no es directamente del tipo SquareImage !

Funciones virtuales : extensibilidad

Aquí aparecen todos los beneficios del polimorfismo : si tengo una parte del programa escrita en términos de una versión genérica de mis objetos (por ejemplo, en términos de Image), puedo sin problema escribir versiones especializadas de Image, como $\mathrm{SquareImage}$, $\mathrm{BinaryImage}$, y usarlas luego a través de esa parte del programa.

En esta parte del código, no necesito saber cuales son las derivadas, y se puede añadir otras, no me importa ! (mi código seguirá funcionando)

Funciones virtuales : ejemplo

Un ejemplo clásico de función virtual es una función que regrese el nombre de la clase :

```
const char *Image::getTypeName() {
        return "Image";
};
const char *SquareImage::getTypeName() {
        return "SquareImage";
};
```

Funciones virtuales : ejemplo

O en el caso de procesamiento de imágenes :

```
int Image::someProcessing() {
        // Version generica
};
int SquareImage::someProcessing() {
        // Version que puedo optimizar
        // para este tipo particular de imagenes
};
```

Mi código :

```
int fonc(Image &img) {
        img.someProcessing();
}
```

Funcionará adaptándose al tipo de objeto que le paso por referencia.

Funciones virtuales : ejemplo

```
class Image {
public:
        virtual const char *getTypeName() {
                return "Image";
        };
};
class GreyImage : public Image {
public:
        const char *getTypeName() {
                return "GreyImage";
        }
};
class BinaryImage : public GreyImage {
};
```

No defino $\mathrm{getTypeName}$ para mi clase $\mathrm{BinaryImage}$.

Funciones virtuales : ejemplo

```
BinaryImage bim;
Image &imgRef = bim;
cout << bim.getTypeName() << endl;
```

Me da :

```
GreyImage
```

o sea, en el momento de la ejecución, se elige la versión del método la mas “cercana” en la jerarquía de las clases.