Programación en C++ (8) : polimorfismo

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Septiembre 2009

Outline

Previously en la clase...

Hemos visto que en C++, podemos formar nuevas clases o estructuras a partir de clases o estructuras existentes

- Dos mecanismos : herencia y composición
- Herencia : para objetos que forman sub-conyunto de un conjunto de objetos (versión especializada).
- Composición : para objetos que naturalmente vienen compuestos de otros objetos.
- Mecanismos para controlar el acceso en la herencia.
- Mecanismos automáticos de generación de constructores default/copia y asignación.
- Polimorfismo a través de funciones virtuales.

Outline

Late binding

El uso de la palabra llave virtual hace que el compilador crea pedazos de código dedicados al procesamiento de late binding al lugar de poner, clásicamente, los argumentos de la función y su dirección memoria.

Ese procedimiento esta basado en la creación de tablas virtuales : el objeto viene con un apuntador (secreto), llamado *vpointer* que apunta hacia esa tabla virtual, que contiene las direcciones de las funciones propias a ese objeto.

Late binding

Miremos con un debugeador el contenido de los objetos siguientes

```
class Image {
  int width;
  int height;
public:
  Image() : width(0),height(0) {};
  const char *getTypeName() {return "Image";};
};
class GreyImage : public Image {
  int someData;
public:
  GreyImage() : Image(),someData(0) {};
  const char *getTypeName() {return "GreyImage";}
};
```

Late binding

Miremos con un debugeador el contenido de los objetos siguientes

```
class BinaryImage : public GreyImage {
  int moreData;
public:
  BinaryImage() : GreyImage(),moreData(0) {};
};

int main() {
  BinaryImage bim;
  BinaryImage bim2;
  GreyImage gim;
  Image im;
}
```

Late binding

Sin función virtual :

```
(gdb) p bim
$1 = {
  <GreyImage> = {
    <Image> = {
      width = 0,
      height = 0
    },
    members of GreyImage:
    someData = 0
  },
  members of BinaryImage:
  moreData = 0
}
```

Late binding

Con funciones virtuales :

```
(gdb) p bim
$1 = {
  <GreyImage> = {
    <Image> = {
      _vptr$Image = 0x3028,
      width = 0,
      height = 0
    },
    members of GreyImage:
    someData = 0
  },
  members of BinaryImage:
  moreData = 0
}
```

Late binding

Con funciones virtuales :

```
(gdb) p bim2
$3 = {
  <GreyImage> = {
    <Image> = {
      _vptr$Image = 0x3028,
      width = 0,
      height = 0
    },
    members of GreyImage:
    someData = 0
  },
  members of BinaryImage:
  moreData = 0
}
```

Late binding

Con funciones virtuales :

```
(gdb) p gim
$4 = {
  <Image> = {
    _vptr$Image = 0x3068,
    width = 0,
    height = 0
  },
  members of GreyImage:
  someData = 0
}
```

Late binding

Con funciones virtuales :

```
(gdb) p im
$5 = {
  _vptr$Image = 0x3078,
  width = 0,
  height = 0
}
```

O sea :

- Los objetos son "más gordos" que lo que esperado, con este apuntador vptr.
- Cada clase lleva su propia tabla virtual, compartida por todos los objetos que la instancian. Notar que no cambia si pongo mas de una función virtual (pero cambiarán las direcciones en memoria).

Late binding

Que hay exactamente en estas tablas ? Pues let's see :

```
(gdb) p *bim._vptr$Image
$4 = (__vtbl_ptr_type) 0x2d88 <GreyImage::getTypeName()>
(gdb) p *gim._vptr$Image
$5 = (__vtbl_ptr_type) 0x2d88 <GreyImage::getTypeName()>
(gdb) p *im._vptr$Image
$6 = (__vtbl_ptr_type) 0x2d04 <Image::getTypeName()>
```

O sea, tenemos exactamente lo dicho : un arreglo de direcciones hacia métodos !

Late binding

Si añado otra función virtual someProcessing :

```
(gdb) p bim._vptr$Image[0]
$21 = (__vtbl_ptr_type) 0x2d0c <GreyImage::getTypeName()>
(gdb) p bim._vptr$Image[1]
$22 = (__vtbl_ptr_type) 0x2d60 <BinaryImage::someProcessi
```

El tipo no es explícitamente presente pero implícitamente sí, ya que las correspondencias están establecidas de manera no ambigua entre los objetos y los métodos.

Late binding

Entonces, cuando se llama una función desde un apuntador/referencia Image :

- Si el método es normal (no virtual) pues se pone una llamada clásica al método, pero será necesariamente el método de Image.
- Si el método es virtual, se pone código para llamar al buen método en la tabla virtual :

```
CALL Image::someProcessing()
```

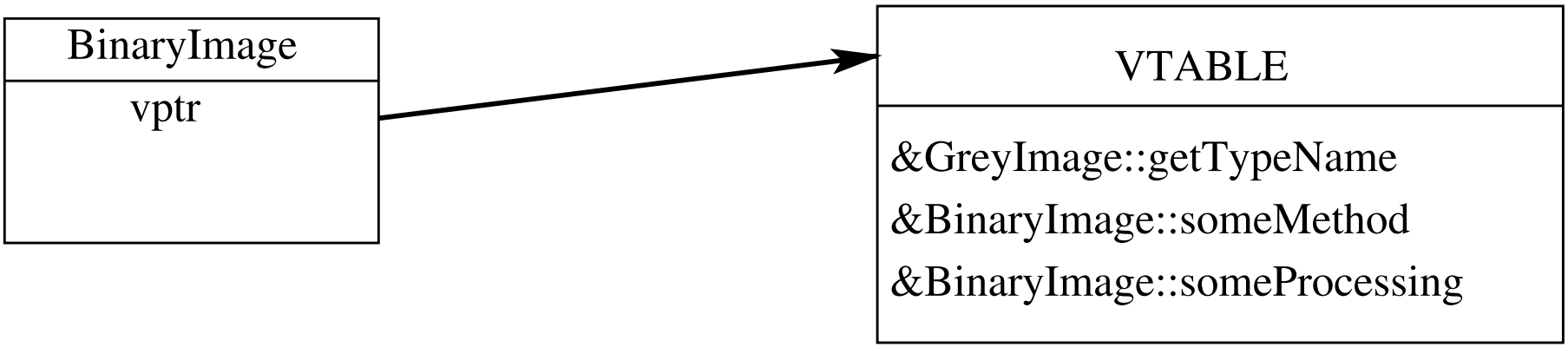
es reemplazado por :

```
CALL vptr[1]
```

Late binding

- El indice esta determinado fácilmente porque el orden de las funciones dentro de la tabla es el mismo para todos objetos de cualquier clase derivando de Image.
- La inicialización de la tabla virtual es fundamental y necesaria : se hace en el constructor !

Funciones virtuales : situación

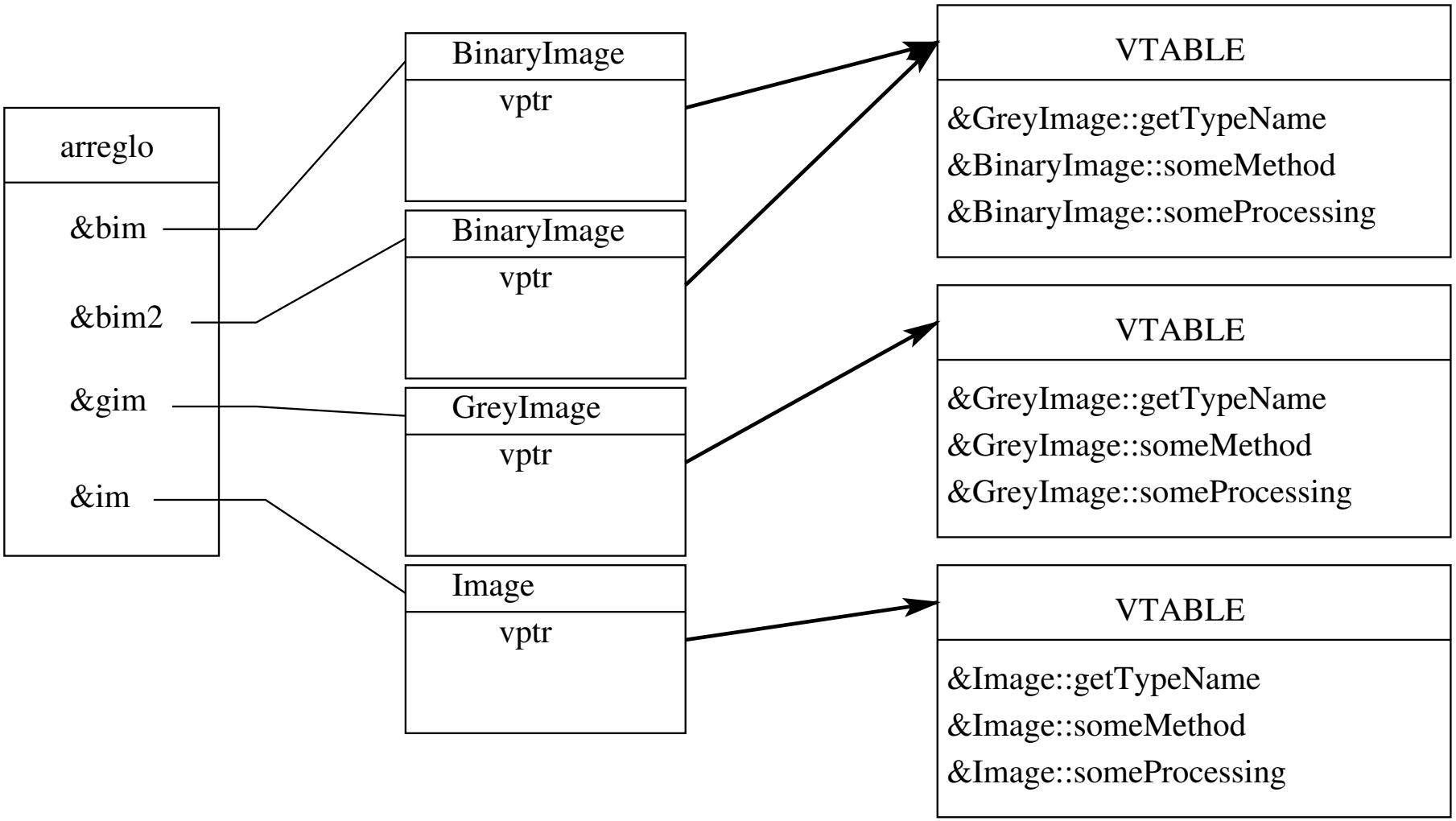


Funciones virtuales : situación

Imagina ahora un arreglo de apuntadores hacia Image :

```
BinaryImage bim;
BinaryImage bim2;
GreyImage gim;
Image im;
Image *arreglo[4];
arreglo[0] = &bim;
arreglo[1] = &bim2;
arreglo[2] = &gim;
arreglo[3] = &im;
```

Funciones virtuales : situación



Y con valores ?

Con un objeto, pues no se necesita nada de *late binding* ya que no hay ninguna ambigüedad (no hay upcasting) :

```
BinaryImage bim;
bim.someProcessing();
GreyImage gim;
Image im2=gim;
cout << im2.getTypeName() << endl;
```

im2 es sin ambigüedad un objeto de tipo Image.

¿Por qué no es el default ?

No es el mecanismo por default, aunque debería de ser si se seguía ortodoxia OO, porque, simplemente, no es lo más eficiente (*early binding*) y que hay que dejar en primer lugar esta eficiencia (la que esta heredada del C). Las tablas virtuales y funciones virtuales son esencialmente introducidas para un mejor diseño de los programas; pueden representar un sobrecargo de procesamientos pero raramente representativo con respeto a lo demás de una implementación.

Clases abstractas : métodos virtuales puros

Existe a veces la necesidad de hacer una clase que servirá de base para otras sin que instanciaciones de esa clase base tenga realmente sentido, en particular en el caso de clases que dan abstracciones útiles de otras. Un ejemplo clásico :

```
class Shape {
public:
        virtual int getArea();
};
class Square : public Shape {
public:
        int getArea();
};
...
```

Clases abstractas : métodos virtuales puros

La clase Shape sí es útil para escribir funciones genéricas que se apliquen a apuntadores o referencias hacia Square, $\mathrm{Triangle}$ o Circle. Ahora, qué sentido tendría una implementación de $\mathrm{getArea}()$ para esa clase ? Se puede escribir, pero no serviría de todos modos.

Clases abstractas : métodos virtuales puros

Existe en C++ la posibilidad de hacer de esa clase una clase abstracta, o sea que no pueda ser implementada o instanciada. Para eso, es suficiente declarar una de sus funciones virtuales como funcion virtual pura, añadiendo a su declaración un "=0" :

```
class Shape {
public:
        virtual int getArea() = 0;
};
...
```

Clases abstractas : métodos virtuales puros

Los métodos virtuales puros obligan a implementar versiones de esos métodos en las clases derivadas

- Manera de separar aún mas interfaz de implementación, esta vez a través de varias clases (clases abstractas/clases "normales").
- Manera de obligar los que van a heredar de tus clases a implementar unas funciones

Clases abstractas : métodos virtuales puros

Notar que :

- Para implementar este mecanismo, deja vacía la entrada correspondiente de la tabla virtual.
- En consecuencia, una clase heredera después de varias "generaciones" puede no tener implementada la función, pero uno de sus "papas" sí.
- Se puede declarar sólo referencias o apuntadores a una clase abstracta :

```
Rectangle r;
Shape &shapeRef = r;
```

Igualmente, no se puede llamar una función con objetos de una clase abstracta por argumentos.

Clases abstractas : métodos virtuales puros

Notar que :

- Se puede definir métodos virtuales no-puros dentro de una clase abstracta.
- Se puede aún dar una definición a una función virtual pura, con la condición de que no sea inline. Esta función no será puesta en las tablas virtuales pero se podrá invocar desde clases herederas.
- Si una clase derivada no implementa funciones virtuales puras, será también una clase abstracta.

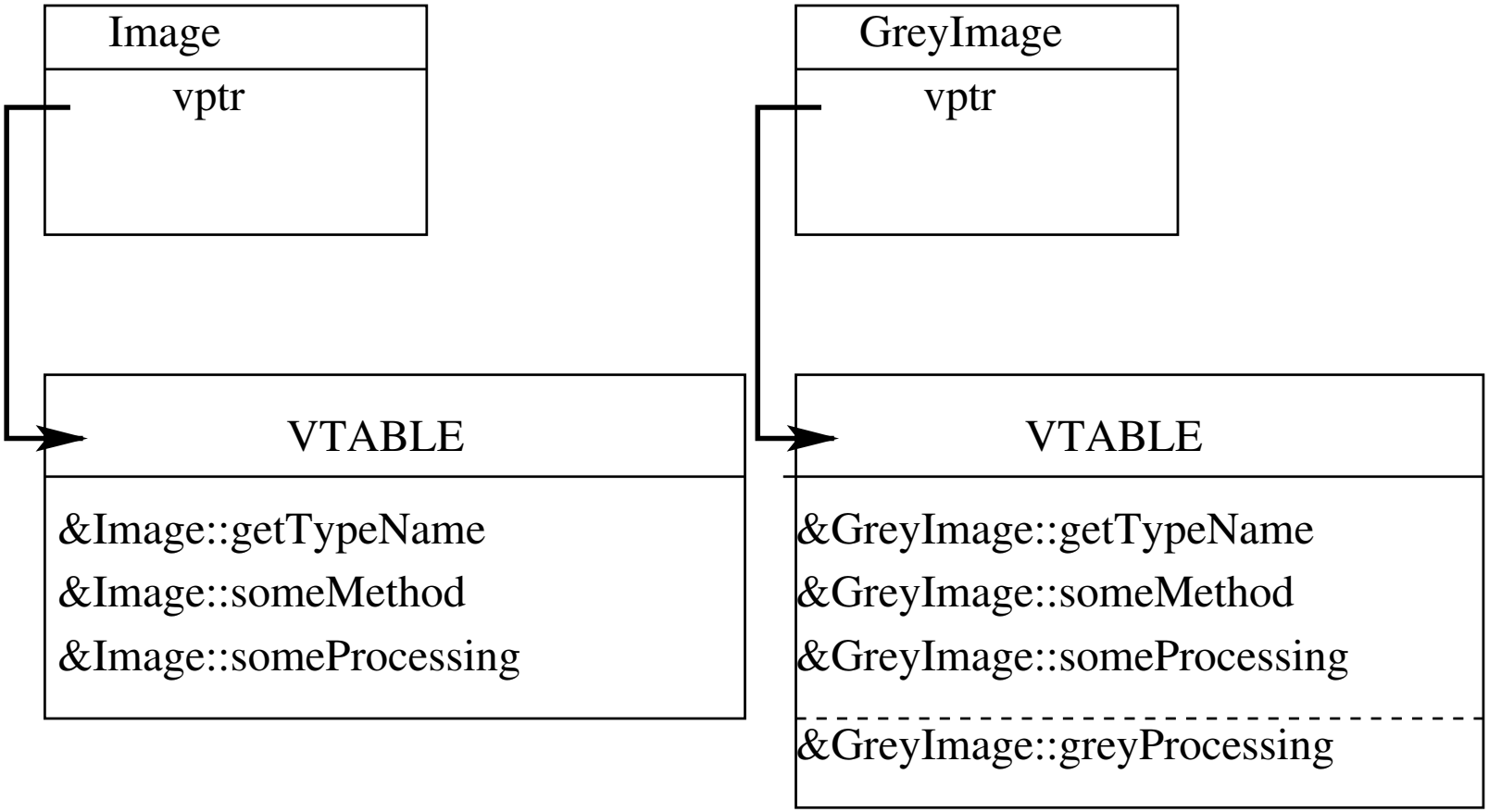
Herencia y la vtable

Imaginemos que tenemos una función virtual que esta definida solo al nivel de GreyImage y no al nivel de Image :

```
virtual void greyProcessing() {};
```

¿Qué pasa ahora con las tablas virtuales ?

Herencia y la vtable



Los espacios para las funciones virtuales adicionales son simplemente añadidos en la tabla virtual ; el upcasting hacia un tipo dado hace que el compilador toma en cuenta tablas virtuales de tamaño y estructura bien especificados.

Herencia y la vtable

Ahora ¿qué pasa si quiero llamar a greyProcessing() desde un apuntador Image

```
Image *img;
...
((GreyImage *)img)->greyProcessing()
```

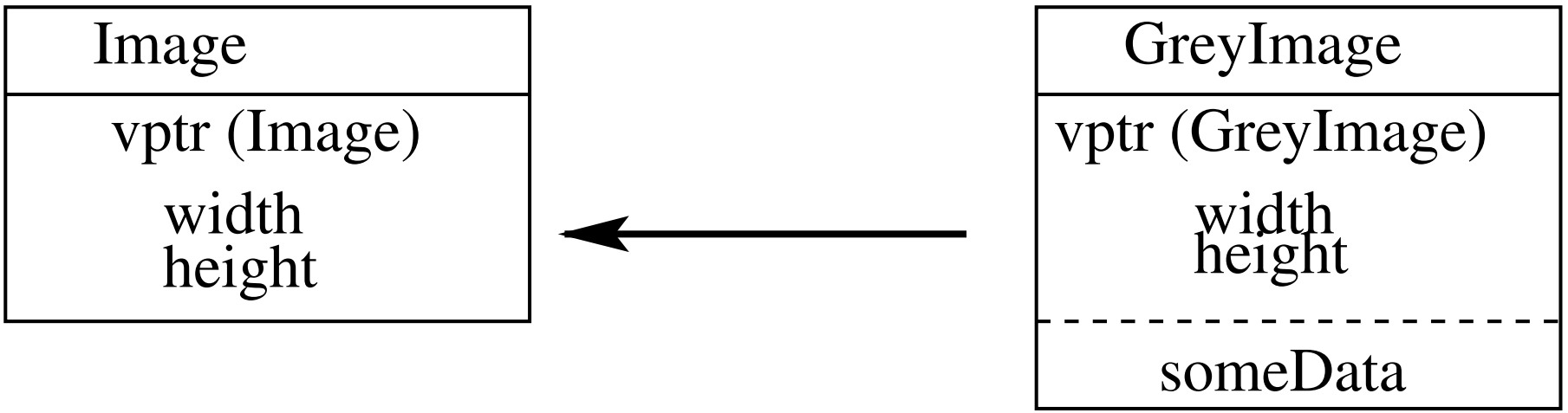
Si sí de manera segura que mi img es de tipo GreyImage, entonces puedo hacer un cast y llamar métodos de GreyImage. Pero en el caso general, hay que verificarlo y eso implica poder determinar el tipo (Run Time Type Identification).

Upcasting de objetos en valor

Pasar referencias o apuntadores permite hacer upcasting de manera transparente y sin efectos de bordo porque todas direcciones de todos tipos tienen el mismo tamaño ! Ahora con valores, tenemos problemas porque los tamaños sí son diferentes.

Lo que hace entonces es simple : para poner el argumento sobre la pila, tiene de todos modos que crear un objeto de tipo la clase de base. Le da una tabla virtual de los objetos de esta clase y recopia la parte “superior” del objeto derivado (lo común).

Upcasting de objetos en valor



Ya no tenemos el mismo tipo de upcasting : pasamos aquí por la creación de un objeto diferente, instanciando la clase base, y no es polimorfismo.

Funciones virtuales y overloading

Unas reglas :

- No se puede redefinir un método virtual en una clase derivada con mismos argumentos y un tipo de regreso diferente.
- Se puede sobrecargar (argumentos diferentes).
- El hecho de redefinir una versión de un método (virtual o no !) de la clase de base esconde las eventuales formas sobrecargadas de ese método.

Funciones virtuales y overloading

Un caso en que sí se puede redefinir una funcion virtual cambiando el valor de regreso : cuando el regreso es un apuntador o una referencia hacia la clase base; en este caso, se puede regresar un apuntador o una referencia hacia una instancia de la clase derivada

```
class Image {
  int width,height;
public:
  Image() : width(0),height(0) {};
  virtual Image &modify() {return *this;};
};
class GreyImage : public Image {
  int someData;
public:
  GreyImage() : Image(),someData(0) {};
  GreyImage &modify() {return *this;};
};
```

Funciones virtuales y overloading

Esa ultima funcionalidad no es exactamente lo que se espera de polimorfismo, y no se usará cuando se necesita procesamientos genéricos, pero puede ser útil y mas eficiente en otras ocasiones en que versiones especializadas son necesarias.

Funciones virtuales y constructores

Nunca olvidar que, cuando se trata de funciones virtuales, el compilador tiene que añadir código de inicialización de las tablas virtuales en cada constructor. Eso se puede sumar a otras cosas (llamadas a constructores de elementos incluidos) que hace que ya no es en nada benéfico usar constructores `inline`

Funciones virtuales y constructores

¿Qué pasa si estamos dentro del constructor y que llamamos a una función virtual ? A priori se podría pensar que finalmente un constructor es sólo un tipo especial de métodos.

Pero :

- El método llamado puede ser llamado a un nivel "más abajo" en la jerarquía y puede requerir que los objetos de "arriba" hayan sido inicializados !
- Cada llamada a un constructor crea también el apuntador a la tabla virtual : en un constructor dado, sólo tenemos acceso a la tabla virtual que ha sido apuntada en este momento, entonces el mecanismo no se puede usar "todavía" !

Entonces una función virtual llamada en un constructor se comporta como función normal (la local es llamada).

Destructores/ destructores virtuales

Se recuerda que los destructores están llamados en el orden reverso de los constructores, empezando por "limpiar" los niveles mas derivados, y llamando a los destructores de niveles mas genéricos. Se podría pensar que manejar constructores y destructores es igual por esa simetría.

Pero no !

Destructores virtuales

Porque :

```
Image *img = new GreyImage(); // Upcasting
delete img;
```

Se distingue entre los constructores/alocaciones (explícitamente, por los nombres), pero el delete es "ambiguo" : se necesita en este ejemplo particular llamar al delete (destructor+liberación) adecuado, o sea el de GreyImage. Si no añado mecanismo de destructor virtual, estoy en líos : llamará al de Image sólo. Aquí no tengo problema al declarar el método virtual, ya que todo esta bien inicializado.

Destructores virtuales puros

Ya que manipulamos destructores, la noción de destructor virtual puro se puede definir legalmente, pero esta vez el método no puede quedarse sin definición : estará llamado en todos casos !

Otras diferencias es que (1) no es necesario redefinirlo en las clases derivadas (pero la clase sigue abstracta) y (2) una clase que no implementa el destructor no es necesariamente abstracta (porque el compilador crea uno !)

```
class Shape {
public:
    virtual ~Shape() = 0;
};
Shape::~Shape() {};

class Rectangle : public Shape {};
```

Destructores virtuales puros

La única ventaja fundamental es que impide la instanciación de objetos de esta clase

```
int main() {
  Shape s; // NO !
  Shape *r = new Rectangle();
  delete r;   // Llama al de rectangle !
}
```

Destructores y funciones virtuales

Igualmente a los constructores, no se llama en destructores a métodos virtuales por el mecanismo de tablas virtuales !

Por qué ?

Outline

Jerarquías de objetos

Hay unas ocasiones en que podemos querer manipular apuntadores heterogéneos : imaginar por ejemplo un caso en que queremos hacer una pila (Stack) de objetos. Por ejemplo :

```
int *a = new int;
GreyImage *glm = new GreyImage;
string *s = new string;
Stack pila;
pila.push_back(a);
pila.push_back(glm);
pila.push_back(s);
```

Implementar Stack : ¿cuál es el problema ?

Jerarquías de objetos

La solución : hacer que todos tus objetos heterogéneos no sean tan heterogéneos, y que hereden de una misma clase "madre" : por ejemplo, una clase "Object". Definirías un destructor virtual que permitiría llamar el buen destructor en cada nivel.

En Java eso es la solución por default : todos los tipos son clases que hereden de una clase madre Object.

Jerarquías de objetos

Qué pasa si el objeto que quieres usar para tu Stack, lo quieres heredar de otra clase, externa, sobre que no tienes control. Por ejemplo, una clase que herede de string :

```
class MyString : public string {
...
}
```

En este caso, la única manera de guardar el esquema precedente es de usar herencia múltiple : que MyString herede de string Y de Object.

Operadores virtuales

Operadores son funciones como otras, entonces el mecanismo de funciones virtuales también se aplica :

```
class Number {
public:
  virtual const Number &operator++(int) {};
};
class Integer : public Number {
  int i;
public:
  const Integer &operator++(int) {
    i++;
  }
};
```

Operadores virtuales

Que se usan como vimos para funciones virtuales :

```
int main() {
   Integer a;
   Number *aptr = &a;
   (*aptr)++;
}
```

Operadores virtuales

Ahora tenemos un problema para operadores binarios : queremos poder sumar, multiplicar. . . objetos diferentes manipulados por referencias o apuntadores a tipos genéricos :

```
int main() {
  Integer a;
  Float b;
  Number &n1 = a;
  Number &n2 = b;
  a *= b;
}
```

El mecanismo de tablas virtuales sólo permite recorrer una jerarquía a la vez. ¿Como hacer ?

Operadores virtuales

Hacer un doble recorrido a través de un conjunto de funciones virtuales intermediarias sobrecargadas :

```
class Integer : public Number {
  int i;
public:
  const Integer &operator*=(Number &otro) {
    otro.rightMultiply(this);
    return *this;
  }
  Number &rightMultiply(Integer *i) {...
  };
  Number &rightMultiply(Float *f) {...
  };
};
```

Downcasting

Upcasting es, al menos en el caso de herencia simple, muy seguro : a priori, no hay perdidas de información en el proceso. Ahora, el proceso inverso (*downcasting*) es mucho mas peligroso : iríamos de un tipo genérico hacia una versión especializada, que puede necesitar mas espacio memoria. Un error de tipo puede ser fatal : se necesita al hacer este *cast* verificar que podemos hacerlo !

```
Image *nimg    = new BinaryImage;
GreyImage *img = nimg;
```

No lo acepta el compilador.

Downcasting

Evitar :

- El cast a la C, muy peligroso :

```
Image *nimg     = new BinaryImage;
GreyImage *img = (GreyImage *)nimg;
```

- El static_cast, que esta hecho al momento de la compilación, y no hace ninguna verificación en la ejecución.

Downcasting

Para eso viene el *casting* visto en la clase anterior sobre *casting* : dynamic_cast, que permite hacer *downcast*, verificando el tipo al momento de la ejecución, cuando la clase es polimorfica :

```
Image *nimg     = new BinaryImage;
GreyImage *img = dynamic_cast<GreyImage *>(nimg);
```

El dynamic_cast

- El *downcasting* no funciona con dynamic_cast si no hay polimorfismo (con un método virtual). No aceptaría lo contrario en la compilación.
- En el caso que un dynamic_cast no funcione, regresa 0 al momento de la ejecución (o emite una excepción, en caso de referencias)

```
int main() {
   Integer a;
   Number *aptr = &a;
   Float *f     = dynamic_cast<Float *>(aptr);
   cout << f << endl;
}
```

Imprime:

0

El dynamic_cast

Pero reservar este mecanismo de dynamic_cast a situaciones en que esta necesario : es costoso checar dinámicamente el tipo; si estas seguro que recibes objetos de tal tipo, preferir un *cast* estático.

No olvidar checar el apuntador regresado por el dynamic_cast.

El dynamic_cast : Apuntadores void*

No se puede usar el dynamic_cast con apuntadores sin tipo, como los void*

```
int main() {
  void *aptr = new Integer; // OK
  Integer *f = dynamic_cast<Integer *>(aptr);// NO !
}
```

El dynamic_cast : otra formulación

Lo que hubiéramos podido hacer es un dynamic_cast "a la mano", haciendo exactamente lo que hace el dynamic_cast : primero, verificar el tipo y segundo hacer el *cast*

```
class Image {
        static const int id=0;
public:
         virtual bool isA(int otroid) {
                 return (id == otroid);
         }
};
```

El dynamic_cast : otra formulación

Una clase derivada :

```
class GreyImage : public Image {
  static const int id=1;
public:
  bool isA(int otroid) {
    return (id == otroid)||Image::isA(otroid);
  }
  static GreyImage *dynCast(Image *img) {
    return img->isA(id)?static_cast<GreyImage *>(img):0;
  }
};
```

Y otra :

```
class SquareImage : public Image {
  static const int id=2;
public:
  bool isA(int otroid) {
    return (id == otroid)||Image::isA(otroid);
  }
  static SquareImage *dynCast(Image *img) {
    return img->isA(id)?static_cast<SquareImage *>(img):0;
  }
};
```

El dynamic_cast : otra formulación

Luego, se puede usar como un dynamic_cast :

```
int main() {
  Image *g = new GreyImage;
  SquareImage *s = SquareImage::dynCast(g);
  GreyImage *gg = GreyImage::dynCast(g);
  cout << s << endl;
  cout << gg << endl;
}
```

Imprime

0

0x300300

Downcasting

En general, evitarlo y diseñar sus programas para no tener que usarlo ! En práctica, arreglarse para usar upcasting únicamente a través del mecanismo de funciones virtuales.

typeid

Aunque haya que evitar usarlo diseñando bien sus programas, puede ser que necesitemos hacer tests explícitos sobre el tipo del objeto.
Existe para eso un operador de C++ dedicado a eso : typeid

typeid

```
#include <typeinfo>
int main() {
   Image *g = new GreyImage;
   SquareImage s;
   Image &sref = s;
   cout << typeid(g).name() << endl;
   cout << typeid(s).name() << endl;
   cout << typeid(*g).name() << endl;
   cout << typeid(sref).name() << endl;
}
```

typeid

En el ejemplo precedente

P5Image
11SquareImage
9GreyImage
11SquareImage

typeid

Este operador puede ser sobre-usado (en estructuras switch) : en principio hay que evitarlo, ya que su uso hace perder todo el beneficio de polimorfismo, o sea la independencia relativa del código al nombre y la naturaleza de las clases que hereden de una clase base.

typeid

El funcionamiento de typeid es muy simple : se añade una entrada especial en la tabla virtual de las clases, esta entrada apunta a una estructura type_info que contiene, en particular, una cadena de caracteres con el nombre del tipo.