Programación en C++ (9) : herencia multiple; patrones

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

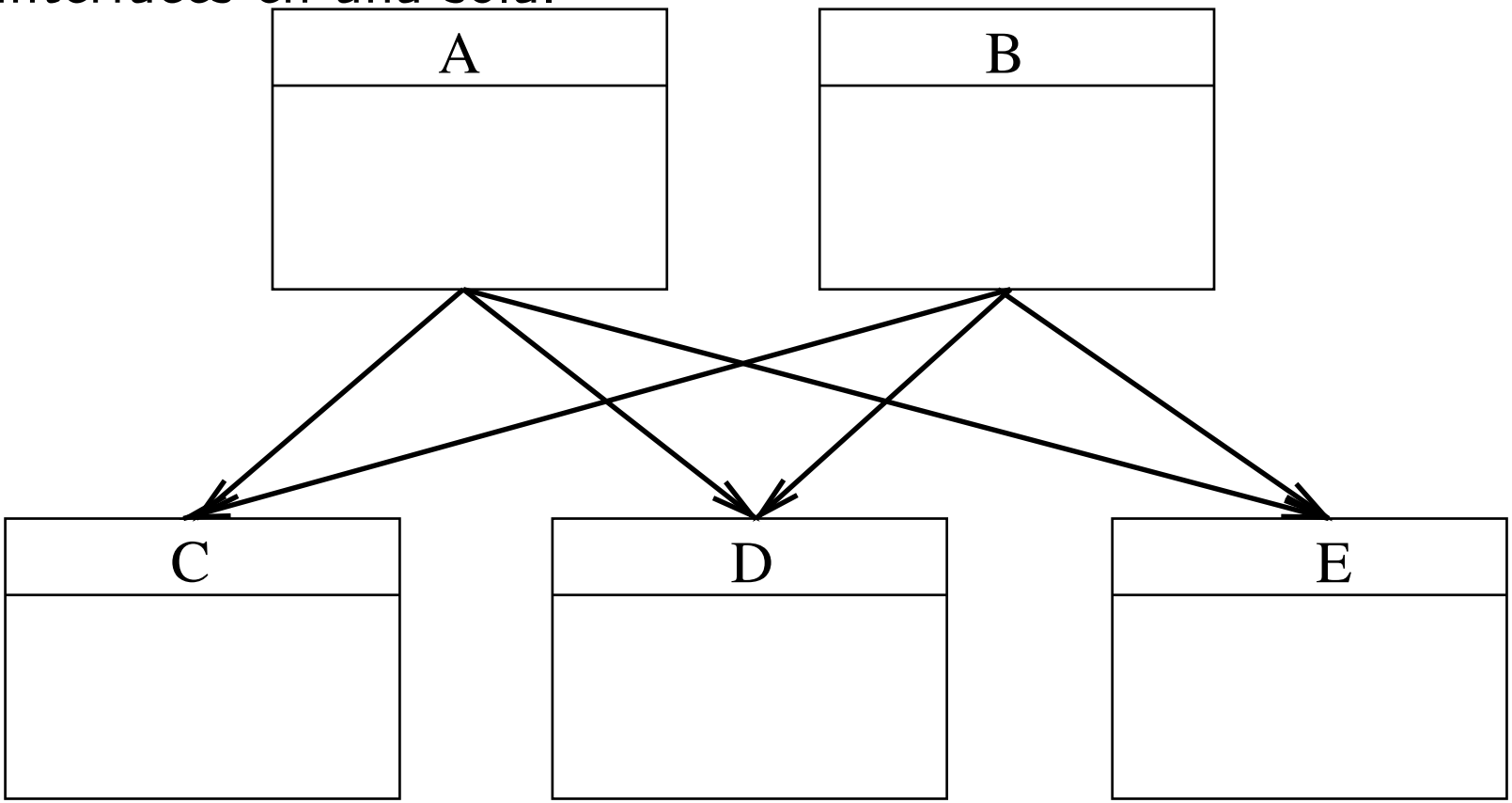
Septiembre 2009

Outline

Outline

Herencia múltiple

En principio al menos, la herencia múltiple es muy simple : decimos que una clase C dada herede de A Y de otra clase B, reuniendo las dos interfaces en una sola.



Herencia múltiple : ¿ útil ?

Recordar que esa propiedad que ofrece el lenguaje C++, no le ofrecen otros lenguajes diseñados para OO, porque lleva a "problemas" : Java por ejemplo, no lo permite.

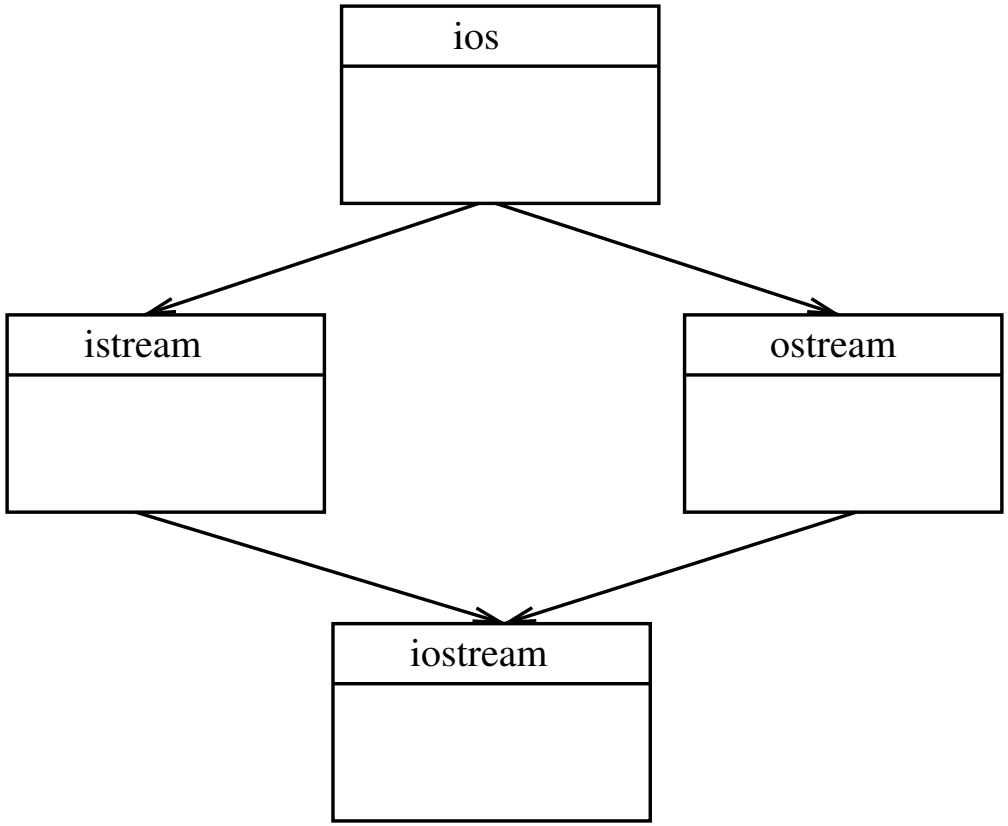
Herencia múltiple : uso típico

Imaginas que te dan dos librerías distintas : una que te provee *containers* (como listas, vectores...) que almacenan objetos de tipo Object, la otra que te provee Image. No tienes control sobre esas clases (las hacen otras personas) : para poder usar Image dentro de los *containers*, necesitas crear una nueva clase ObjImage que hereda de Object y Image.

Remarca importante : con los templates, ese problema se puede solvar de otra manera, mas simple.

Herencia múltiple : uso típico

Otro ejemplo típico es cuando dos clases separadas son útiles, y que también tiene sentido considerar una "unión de las interfaces". Es el caso por ejemplo de iostream :



Herencia múltiple

Se escribe muy simplemente :

```
class A {
public:
        void methodA();
};
class B {
public:
        void methodB();
};
class C : public A, public B {
public:
        void methodC();
};
```

Herencia múltiple

Y se hereda de todos los métodos de las interfaces :

```
C objC;
objC.methodA();
objC.methodB();
objC.methodC();
```

Dos remarcas importantes :

- Los controles de acceso vistos precedentemente siguen aquí.
- Como con herencia simple, no hay posibilidad de heredar parte de una clase (aunque la herencia puede ser $\mathrm{private}\ldots$).

Herencia múltiple : interfaces puras

Un uso de herencia múltiple que no está contestado : heredar interfaces puras y combinarlas en una nueva clase. Eso necesita en C++ el uso de clases abstractas, con métodos virtuales puros

```
class A {
        int datoA;
public:
        virtual void methodA() = 0;
};
class B {
        int datoB;
public:
        virtual void methodB() = 0;
};
```

Herencia múltiple : interfaces puras

```
class C : public A, public B {
        int datoC;
public:
        void methodA() {...};
        void methodB() {...};
};
```

Herencia múltiple : memoria

Sin el virtual :

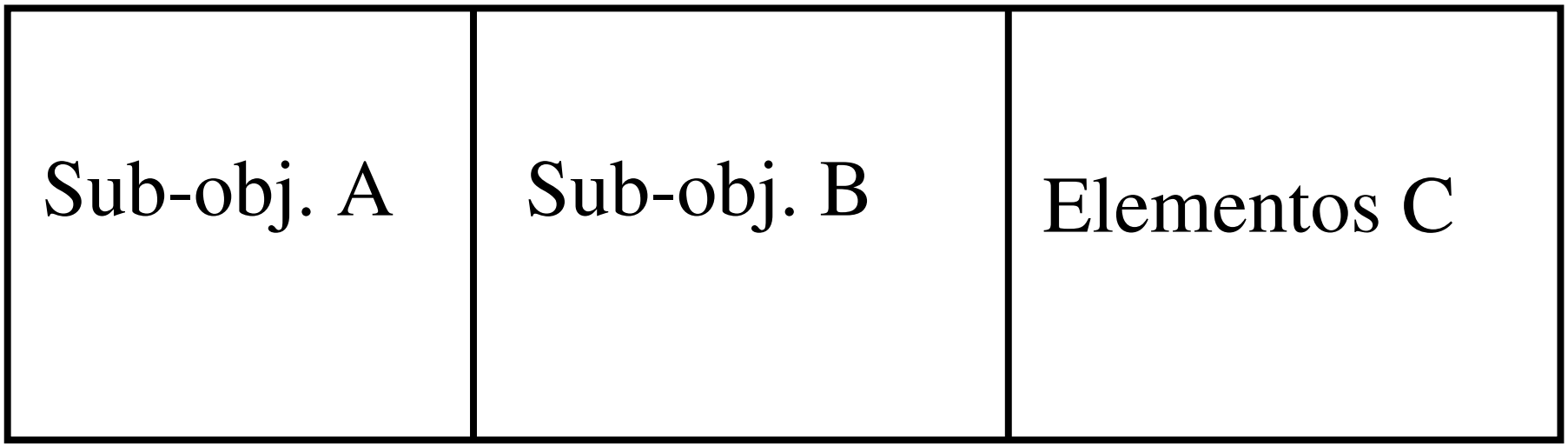
```
cout << sizeof(A) << endl;
cout << sizeof(B) << endl;
cout << sizeof(C) << endl;
```

Me da :

```
4
4
12
```

O sea, aparentemente, y lógicamente, se concatenan los datos.

Herencia múltiple : memoria



Herencia múltiple : upcasting

El *upcasting* es automático y posiciona los apuntadores en el buen lugar en la clase :

```
C oc;
cout <<                          &oc  << endl;
cout << static_cast<A *>(&oc) << endl;
cout << static_cast<B *>(&oc) << endl;
```

Nos da lo correcto :

```
0xbfffea14
0xbfffea14
0xbfffea18
```

Pero cuidado que un reinterpret_cast no hubiera funcionado !

Herencia múltiple : upcasting

Y al regresar al apuntador/referencia del tipo original (C *), no hay problemas, desde cualquier de los dos apuntadores derivados, si efectivamente estamos en el caso de un apuntador hacia un sub-objeto de tipo C :

```
C oc;
A* aptr = static_cast<A *>(&oc);
B* bptr = static_cast<B *>(&oc);
cout << &oc << endl;
cout << static_cast<C *>(aptr) << endl;
cout << static_cast<C *>(bptr) << endl;
```

Nos da :

```
0xbffff03c
0xbffff03c
0xbffff03c
```

Herencia múltiple : upcasting

Los *offsets* están manejados automáticamente entre todos los *casts* de apuntadores, pero cuidado al manipularlos mientras se puede:

```
B bb;
cout << &bb << endl;
cout << static_cast<C *>(&bb) << endl;
```

Me da :

```
0xbffff038
0xbffff034
```

pero no lo puedo usar como apuntador a C !

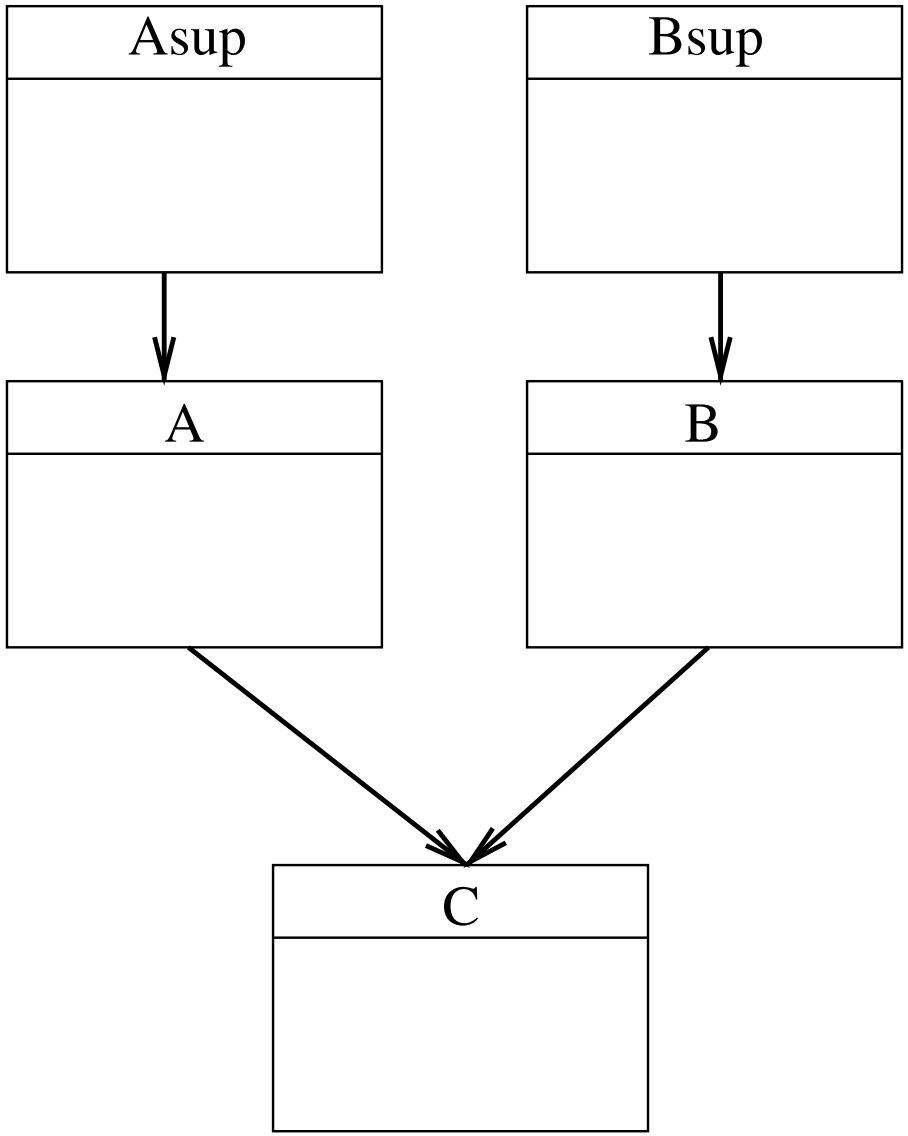
Herencia múltiple : inicialización

La inicialización en los constructores se hace igual que con herencia/composición, a través de lista de inicializadores :

```
C::C(const C& objc) : A(objc), B(objc) {
        ...
}
```

Notar que el orden de llamada de los constructores en caso *default*, o constructores por copia, es el dado en la definición de la herencia múltiple; los destructores están llamados en el orden reverso.

Herencia múltiple : clases polimorficas



Herencia múltiple : clases polimorficas

Al heredar de varias clases con funciones virtuales, notar que los apuntadores hacia las tablas virtuales de cada jerarquía están copiadas también, eso siendo necesario para poder manipular el objeto como apuntador a cada una de las clases base :

```
C oc;
B *bptr = &oc;   Bsup *bsptr = bptr;
A *aptr = &oc;   Asup *asptr = aptr;
```

Herencia múltiple : duplicados

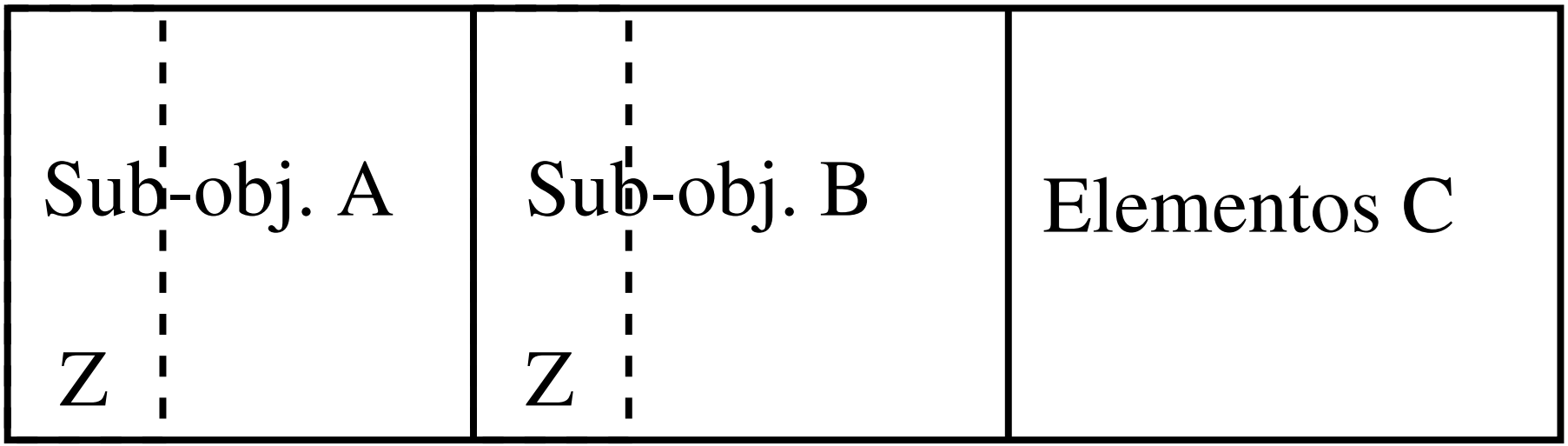
Imaginemos ahora que A y B hereden de la misma clase Z. Si C hereda de los dos, tenemos un problema importante : al imprimir los tamaños de los objetos, nos da :

```
C oc;
cout << sizeof(A) << endl;
cout << sizeof(B) << endl;
cout << sizeof(C) << endl;
```

```
8
8
20
```

O sea, el objeto Z esta presente dos veces en el objeto C !

Herencia múltiple : duplicados



Herencia múltiple : duplicados

No es falso, y a priori se podría usar tal cual, pero se rompe el mecanismo de herencia aquí :

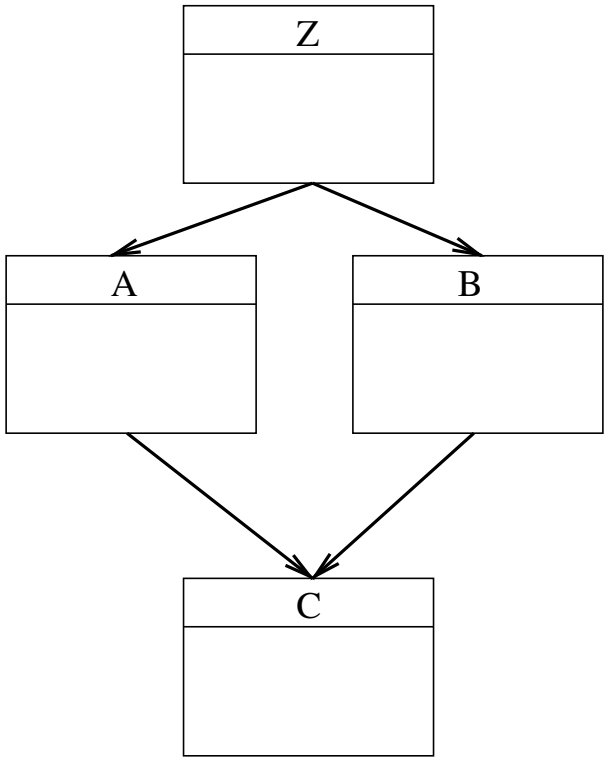
- No tiene mucho sentido tener que manipular dos instancias del objeto $\mathbb{Z}$: no habría coherencia.
- el *upcasting* hacia $\mathbb{Z}$ no puede funcionar : cual de los dos objetos se elegiría ?

  ```
  test3.cpp:35: error: 'Z' is an ambiguous base of 'C'
  ```

- De la misma manera, no se puede invocar a métodos de la base $\mathbb{Z}$, por la ambigüedad, incluso dentro de los métodos de la clase.

Herencia múltiple : duplicados

Lo que queremos realmente es que en el caso de que heredamos de dos clases con base común, esta instancia sea compartida por las dos clases en el objeto heredando de esas dos clases !



Herencia múltiple : base virtual

Este mecanismo existe y consiste en hacer de la clase base (Z) una clase de base virtual :

```
class A : virtual public Z {
...
};
class B : virtual public Z {
...
};
class C : public A, public B {
...
};
```

Herencia múltiple : base virtual

El uso de la palabra llave $\mathrm{virtual}$ hace que el objeto creado (A o B) en una jerarquía de herencia no instancia él mismo el objeto Z

```
A* aptr = static_cast<A *>(&oc);
B* bptr = static_cast<B *>(&oc);
Z *zptr = static_cast<Z *>(&oc);
cout << &oc  << endl;
cout << aptr << endl;
cout << bptr << endl;
cout << &oc.datoC << endl;
cout << zptr << endl;
```

Herencia múltiple : base virtual

El resultado :

```
0xbffff02c
0xbffff02c
0xbffff034
0xbffff03c
0xbffff040
```

Interpretación ? Notar que eso necesita que ambas clases hereden virtualmente de Z ! Tenemos 8 octetos para A y B : el dato Y un apuntador hacia el objeto Z común !

Herencia múltiple : base virtual

La cosa importante en este asunto es que el constructor de Z ya no lo tienen que manejar A y B, sino la clase mas derivada ! En caso de creacion de objetos A o B independientes, sí se invoca al constructor de Z pero en el caso de derivación de un objeto cruzado (con A o algunas de sus derivaciones Y B o una de sus derivaciones), es a este cruzamiento de hacerlo !

Herencia múltiple : base virtual

Un problema sigue : los métodos "normales" definidos en las clases A y B tienen que seguir efectivas a la vez para objetos de estas dos clases "sólos" o en una clase combinada.

```
class A : virtual public Z {
  int datoA;
public:
  A(int z, int a) : Z(z) {datoA=a;}
  friend ostream& operator<<(ostream& os, const A& a)
    return os << static_cast<const Z&>(a)
              << ',' << a.datoA;
  }
};
```

Y al imprimir un objeto C ?

Herencia múltiple : base virtual

El mecanismo de base virtual requiere en varias ocasiones desdoblar las funciones en dos versiones : la versión para el caso de objeto instanciado sólo y la versión de una instanciacion de un objeto "cruzado".

```
class A : virtual public Z {
  int datoA;
protected:
  void localPrint(ostream& os) const {
    os << ','<< datoA; // Solo los datos de A !
  }
public:
  A(int z, int a) : Z(z) {datoA=a;}
  friend ostream& operator<<(ostream& os, const A& a)
    return os << static_cast<const Z&>(a)
              << ',' << a.datoA;
  }
```

Herencia múltiple : base virtual

Así la clase cruzada llama en su método de impresión : las versiones $\mathrm{localPrint}$ de cada clase madre y la impresión de los datos de C para evitar duplicar la impresión del objeto de base Z.

Herencia múltiple : base virtual

La inicialización de los objetos heredados de clases con base virtual se hace así :

- Inicialización de los objetos de bases virtuales, de arriba por abajo y de izquierda a derecha (según las definiciones de las clases).
- Inicialización de los objetos de base no virtuales.
- Inicialización de los elementos de la clase.
- Cuerpo del constructor.

Herencia múltiple : problemas de nombres

Puede haber mas problemas, con los nombres de las clases madres :

```
class A : virtual public Z {
public:
        void someMethod();
};
class B : virtual public Z {
public:
        void someMethod();
};
class C : public A, public B {
...
};
C obc;
objc.someMethod(); // Cual llamar ?
```

Herencia múltiple : problemas de nombres

En este caso, se puede hacer la llamada no ambigua especificando uno de los dos métodos que llamar, sistemáticamente :

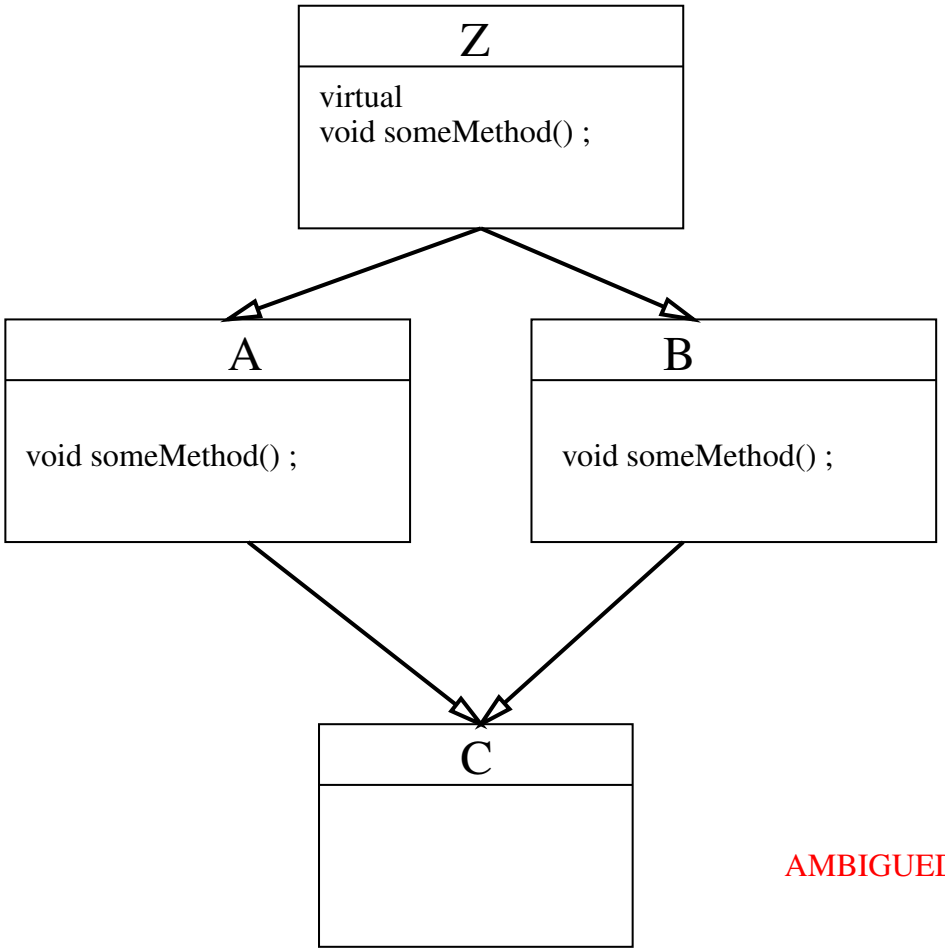
```
class C : public A, public B {
public:
        using A::someMethod(); // No hay ambigüedad
};
C obc;
objc.someMethod();
```

pero ya no podemos llamar al de B, sin hacer un cast o escribiendo un método dedicado dentro de C.

Herencia múltiple : polimorfismo

Para ambigúedades verticales, el mecanismo de polimorfismo maneja lo de llamar la función al buen nivel, o sea el más derivado. En general (el grafo de herencia es mas complicado), se usa el método dominante, o sea el que tiene el nivel de derivación mas elevado (que sea heredero directo o indirecto).

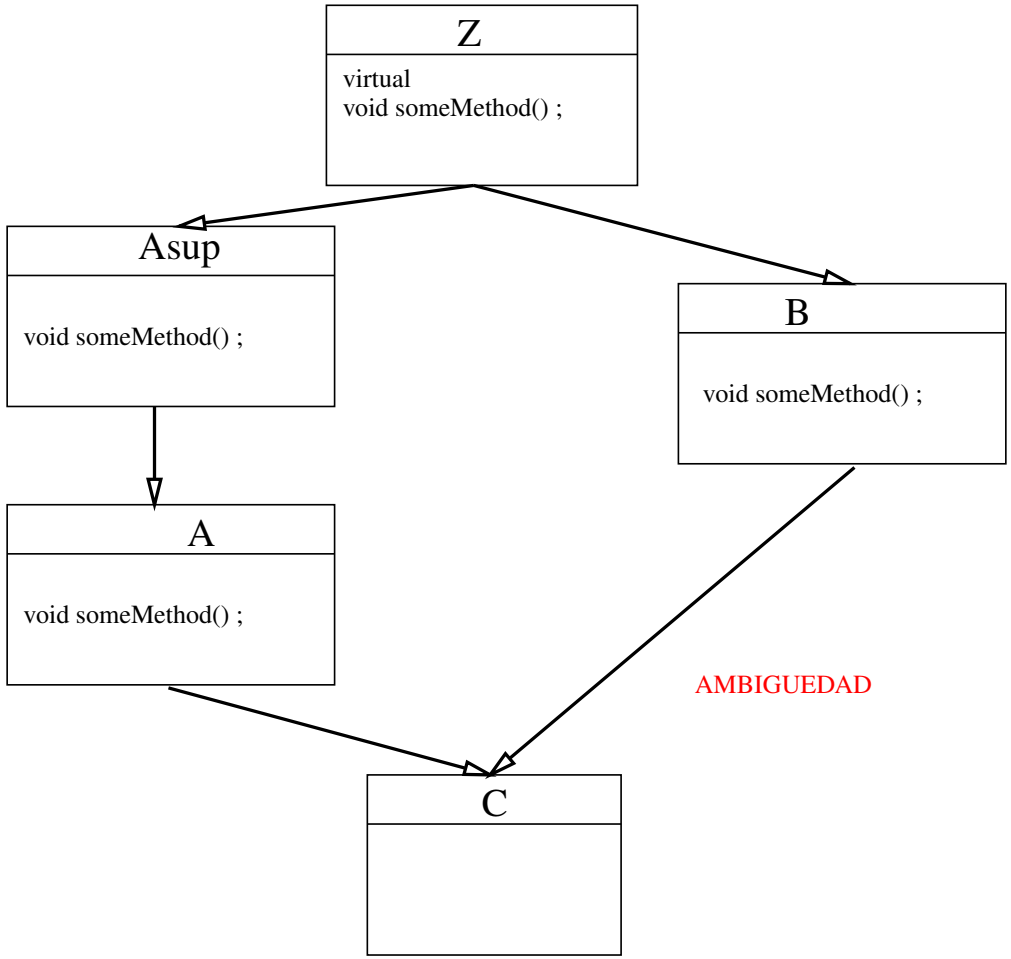
Herencia múltiple : polimorfismo



AMBIGUEDAD

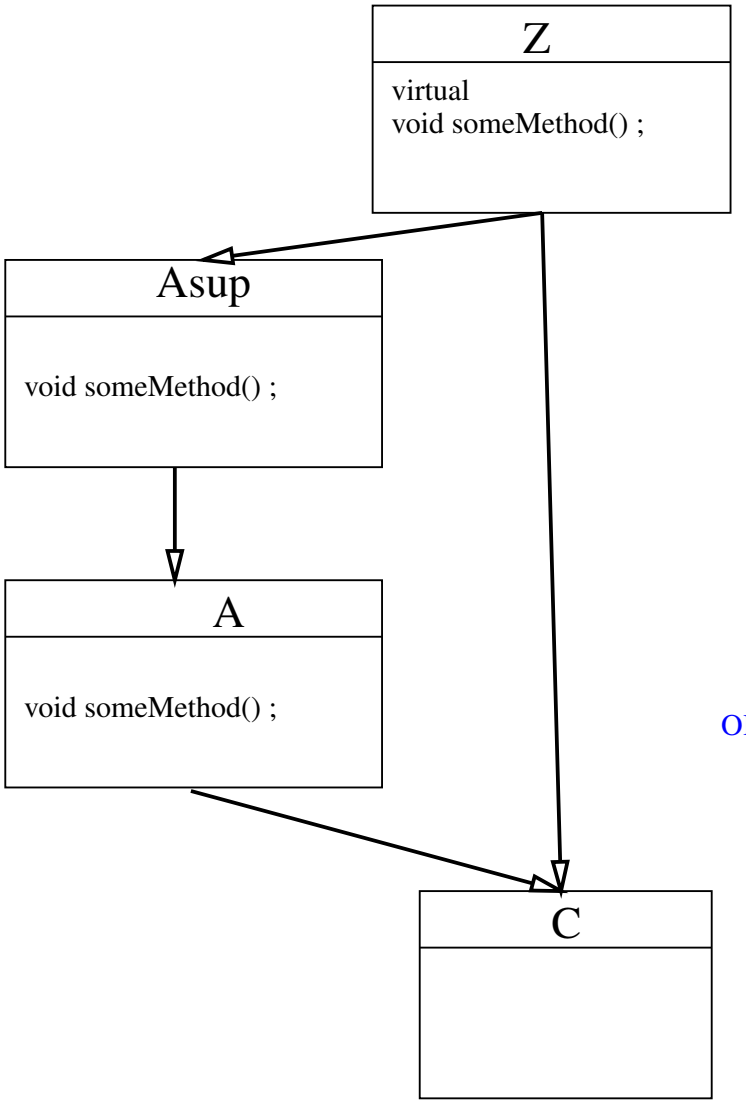
```
error: no unique final overrider for
'virtual void Z::someMethod()' in 'C'
```

Herencia múltiple : polimorfismo




```
error: no unique final overrider for
'virtual void Z::someMethod()' in 'C'
```

Herencia múltiple : polimorfismo



OK ! A::someMethod()

Herencia múltiple : polimorfismo

Eso implica la presencia de una sola jerarquía para una(s) función(es) virtual(es) :

```
class Z {
public:
  virtual void someMethod() {};
};
class Asup : virtual public Z {
public:
  void someMethod() {};
};
class A : public Asup {
public:
  void someMethod() {};
};
class C : public A, virtual public Z {
};
```

dynamic_cast y herencia múltiple

El dynamic_cast se ocupa de todo para hacer downcast hacia el tipo derivado :

```
Z *z = new C; // upcast
C *c = dynamic_cast<C*>(z); // downcast + check
A *a = static_cast<A*>(c); // Otro upcast
Z *zz= static_cast<Z*>(c); // Otro upcast
```

Outline

Templates : introducción

Hasta ahora, las herramientas que vimos (herencia, composición, polimorfismo), permiten re-usar código objeto; los patrones (o *templates*) permiten hacer una cosa tan interesante : re-usar código fuente en otros contextos que el para que estaba diseñado :

```
int sum(int a,int b) {
        return a+b;
}
float sum(float a,float b) {
        return a+b;
}
```

A veces no está muy astuto re-escribir código en funciones sobre cargadas, ya que el código puede ser exactamente lo mismo !

Templates : noción de containers

Una justificación muy importante de la introducción de los *templates* en el lenguaje C++ es la posibilidad de manejar *containers* genéricos. Por *container*, uno se refiere a una estructura que guarda una colección de objetos dados, especificando una manera de introducir/quitar objetos de esta colección. Un arreglo tal que lo propone C es un *container*, una pila también.

Templates : noción de containers

Hasta ahora, los *containers* que podemos hacer pueden :

- estar especializados en un tipo de objetos, por ejemplo enteros,
- manipular objetos por apuntadores,
- aceptar una jerarquía de objetos, y estar expresados en términos del objeto base; pero no tenemos una sola jerarquía en C++.

Templates : noción de containers

Frecuentemente, lo que se usarán son *containers* con tipos apuntadores : por ejemplo, en un sistema de manejo de ventanas. Esas ventanas pueden estar creadas/destruidas en cualquier momento, y no se puede predecir cuantas de ellas manejaremos.

En este caso para hacer cosas sobre todas las ventanas, y de todo modo para guardar en memoria todas las ventanas, se usará un *container* de apuntadores hacia ventanas, creadas por `new` y que necesitarán estar destruidos con `delete`.

Templates : noción de containers

Por ejemplo, una pila de enteros:

```
class StackInt {
  class Node {
    int value;
    Node* next;
    void initialize(int val, Node* nxt);
  };
  Node *head;
  void initialize();
  void cleanup();
public:
  StackInt();
  void push(int k);
  int peek();
  int pop();
};
```

Templates : noción de containers

Y si quiero una pila de float ?

Muy probablemente, tendré que escribir una clase $\mathrm{MUY\ SIMILAR}$, como $\mathrm{StackFloat}$, para implementarlo. De una manera, tendrá mucho código duplicado !

Templates : containers de objetos genéricos

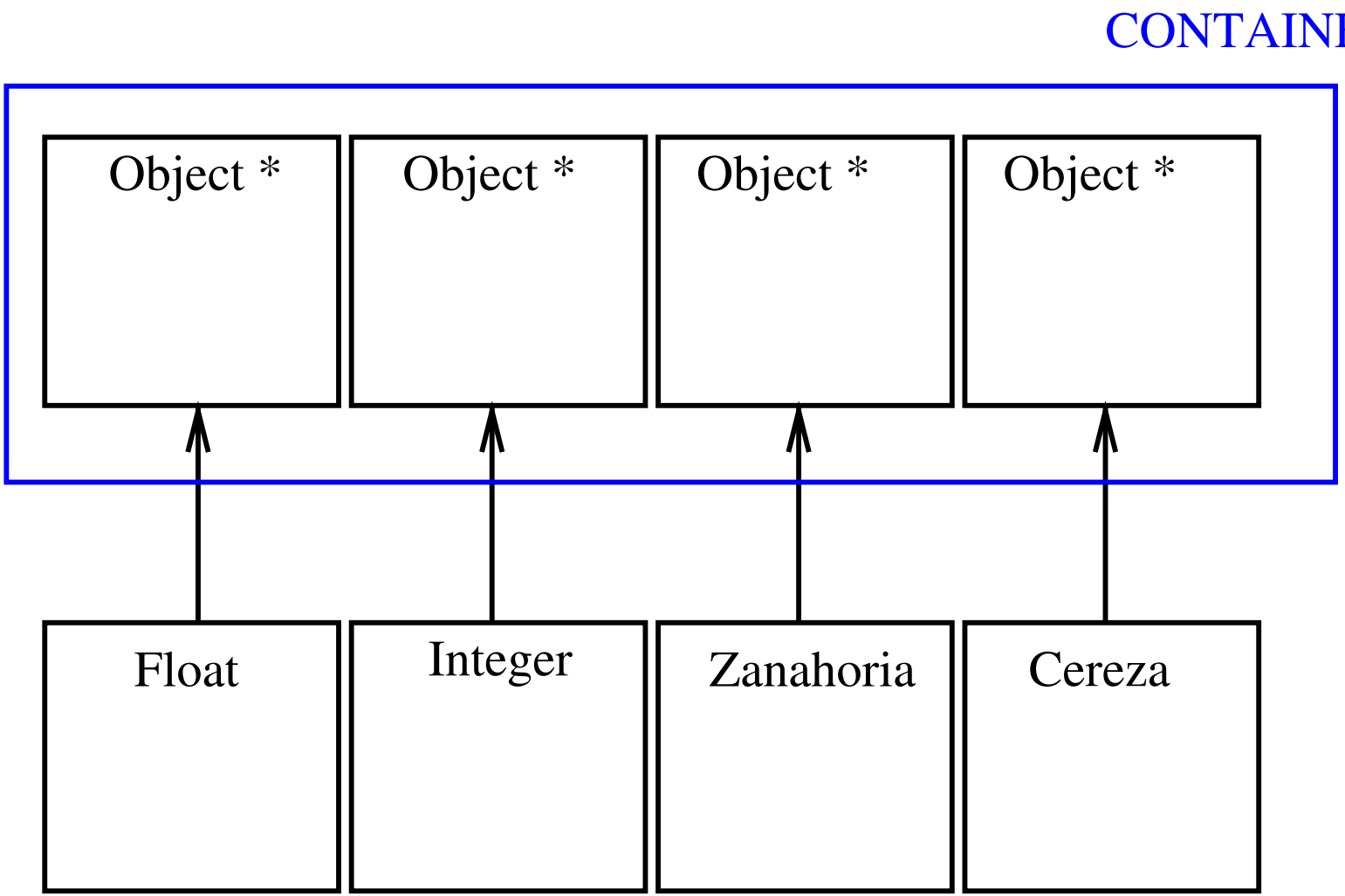
Una opción para salir de este problema es la que adoptan lenguajes como Java o SmallTalk : todos los objetos del sistema, hasta los mas básicos, son incluidos en una misma jerarquía. Gracias al polimorfismo, entonces, se puede escribir código para *containers* de puros Object.

Templates : containers de objetos genéricos

Si fuera el caso en C++...

```
class Stack {
  class Node {
    Object *value;
    Node* next;
    void initialize(Object *val, Node* nxt);
  };
  Node *head;
  void initialize();
  void cleanup();
public:
  Stack();
  void push(Object *k);
  Object *peek();
  Object *pop();
};
```

Templates : containers de objetos genéricos



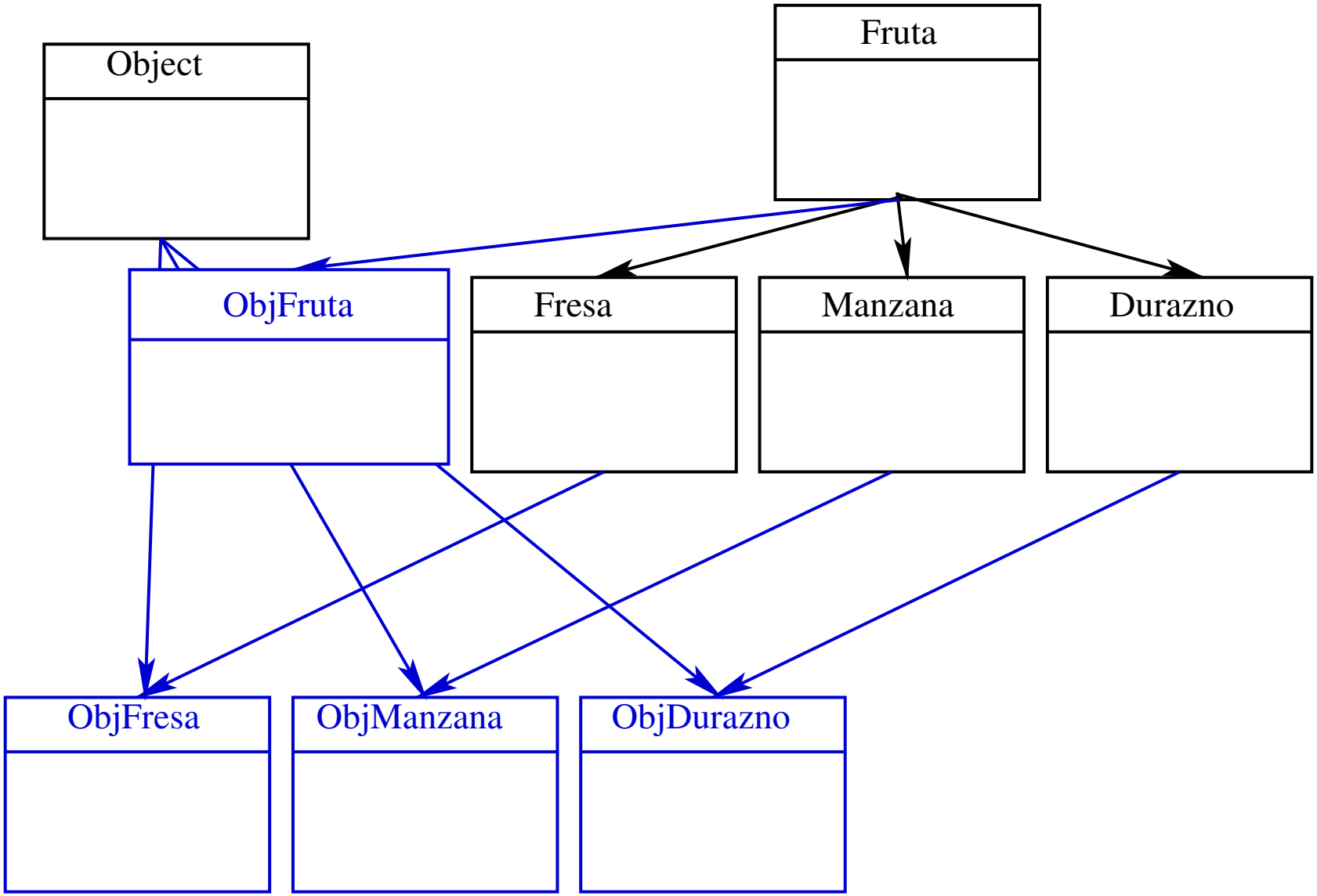
Templates : containers de objetos genéricos

Eso es la técnica mas puramente OO, pero no funciona en C++, o al menos no puede ser tan genérica : el C++ soporta varias jerarquías independientes y no hay tipo base común a todas las clases, sobre todo.

Templates : containers de objetos genéricos

La primera solución adoptada fue usar la propiedad de C++ de permitir herencia múltiple : entonces, hacer clases especiales que heredan de tus clases originales Y de Object. . . Pero nos lleva todos los problemas de la herencia múltiple. . .

Templates : containers de objetos genéricos



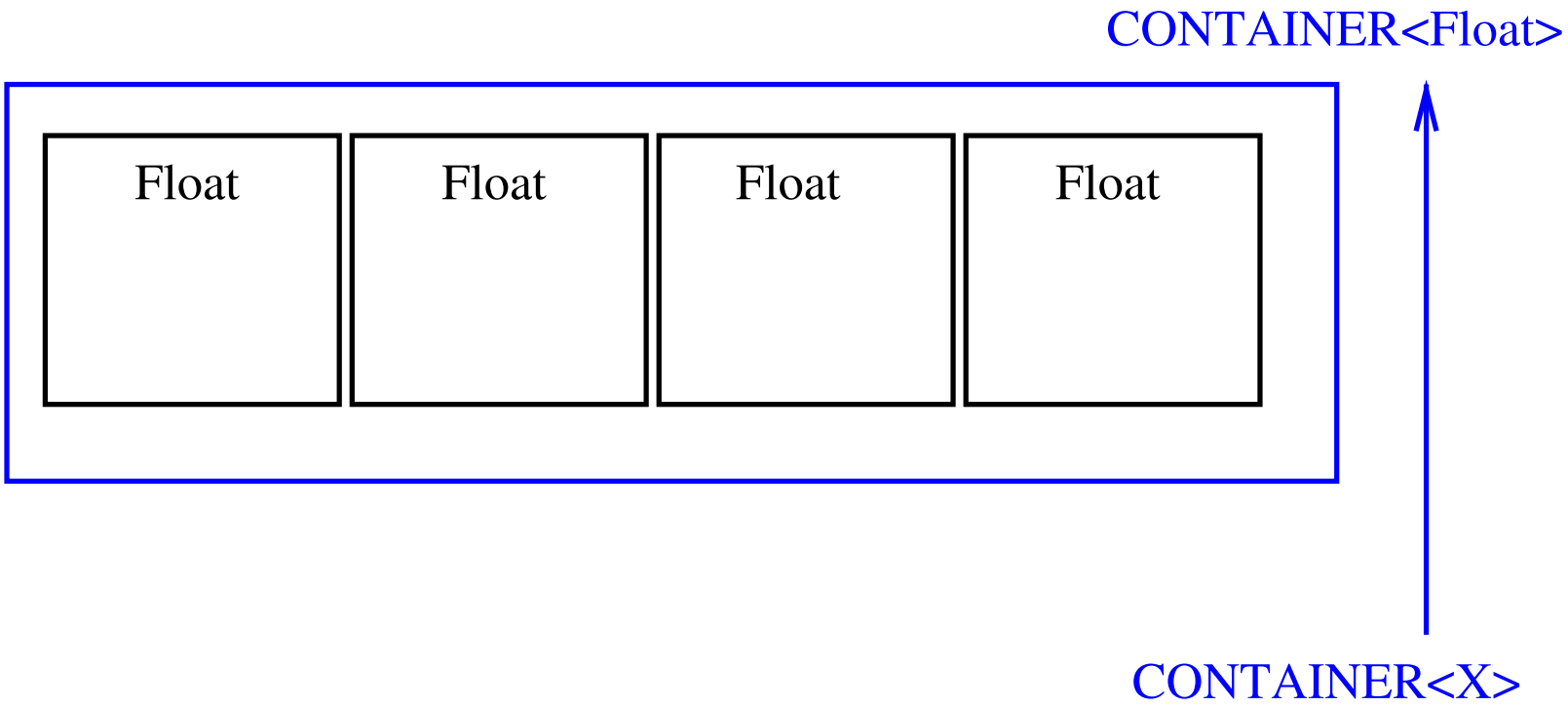
Templates

La idea es muy básica : haces un código que manipula objetos de un tipo genérico, que es un parámetro "como otros" dentro de este código :

```
void fonc(T &someParam) {
        T intermediateValue;
        indermediateValue = someParam * 2;
        ...
}
```

Luego es tarea del compilador de poner un valor a este tipo y de intentar compilar el código resultante, que eventualmente no tiene sentido, en función de los usos que intentas hacer de esa función/esa clase.

Templates : containers de objetos genéricos



Implementación genérica $\longrightarrow$ cuando se necesita para un tipo en particular, se compila.

Templates

El proceso se parece al uso de macros C, en una versión mucho mas sofisticada. La primera versión de los "templates" por Stroustrup era de hecho basada en macros, y la versión actual esta mucho mas avanzada :

- Ya no se necesita usar artificialmente jerarquías de objetos para poder usar containers.
- Está tal vez mas intuitiva la manera de hacer de los templates : se programa por ejemplo containers independientemente de cualquier tipo (después de todo, un container como una pila no es definido en función de un tipo sino en función de funcionalidades que ofrece sobre los objetos contenidos).

Templates : uso

Un ejemplo (1) :

```
template<class T>
class Arreglo {
  T *datos;
  unsigned int size;
public:
   Arreglo(unsigned int s=1) : size(s) {
        if (!s)
          throw 1;
        datos = new T[s];
  }
  ~Arreglo() {
        delete[] datos;
  }
```

Templates : uso

Un ejemplo (2) :

```
    T& operator[](unsigned int index) {
        if (index < 0 && index >= size) {
            throw 1;
        }
        return datos[index];
    }
};
int main() {
        Arreglo<double> arrD;
        Arreglo<int> arrI;
}
```

Templates : uso

Código objeto correspondiente :

```
00000076 S __ZN7ArregloIdEC1Ej
000001d4 S __ZN7ArregloIdEC1Ej.eh
00000164 S __ZN7ArregloIdED1Ev
00000254 S __ZN7ArregloIdED1Ev.eh
000000de S __ZN7ArregloIiEC1Ej
00000200 S __ZN7ArregloIiEC1Ej.eh
00000146 S __ZN7ArregloIiED1Ev
0000022c S __ZN7ArregloIiED1Ev.eh
```

Me generó código objeto "duplicado" a partir de mi patrón, para enteros y dobles. Hubiera funcionado con cualquier otro tipo.

Templates : uso

- La única cosa que hacer es introducir la definición por :

  ```
  template<class T>
  ```

- T puede ser cualquiera cosa a priori (clase o tipo elemental).
- Por supuesto, puede haber código que no dependa del tipo "parámetro".
- Si no se usa el patrón (con un tipo T en particular), nada es compilado en código objeto.

Templates : uso en funciones no inline

Si no quiero usar funciones inline :

```
template<class T>
Arreglo<T>::Arreglo(unsigned int s) : size(s) {
  if (!s)
    throw 1;
  datos = new T[s];
}
```

Se le necesita dar al compilador lo necesario para construir correctamente los nombres de funciones.