# Agrégation et Composition

- Collaboration entre deux classes

- Trois types de lien
  - Fort ➔ Composition
  - Moyen ➔Agrégation interne
  - Faible ➔ Agrégation externe

# La composition

- Contenance d'une autre classe

```
1: using namespace std;
2:   class A {
       private:
       Toto  t;
       string s;
       public:
         A() : t(100), s("toot")
         {}
     };

     class B : public A {
       public:
       B() :  A(3) {}
     };
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# La composition

- Contenance d'une autre classe
- Initialisation comme l'héritage

```
1: using namespace std;
2:   class A {
       private:
       Toto  t;
       string s;
       public:
         A() : t(100), s("toot")
         {}
     };

     class B : public A {
       public:
       B() : A() {}
     };
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# La composition

- Contenance d'une autre classe
- Initialisation comme l'héritage
- Destructeur de Toto automatiquement appelé

```
1: using namespace std;
2:   class A {
       private:
       Toto  t;
       string s;
       public:
         A() : t(100), s("toot")
         {}
    };

    class B : public A {
       public:
       B() :  A(3) {}
    };
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

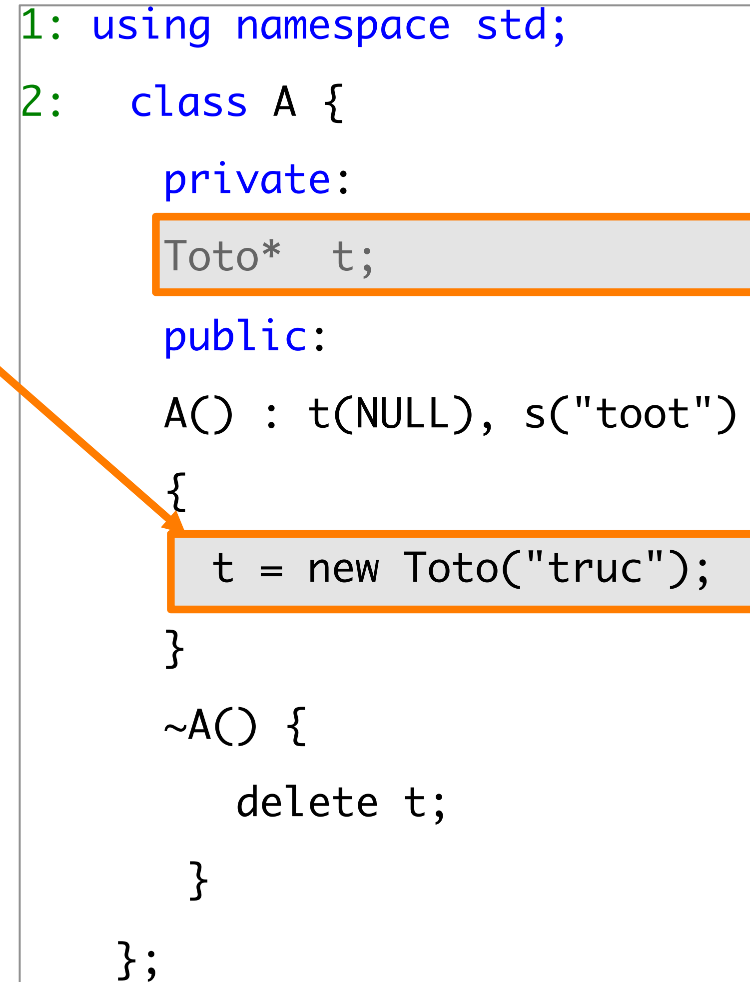
# L'agrégation

- Composition faible entre 2 classes via
  - pointeur
  - référence

- Deux types :
  - interne par pointeur
  - externe par référence
  - externe par pointeur

# L'agrégation interne

- t crée dans la classe
- t doit être libéré
  - Destructeur de A
- Quel accès externe pour t ?



# L'agrégation externe par référence

- Initialisation obligatoire dans le constructeur
- Pas de libération
- Quel accès externe pour t ?

```
1: using namespace std;
2:   class A {
      private:
      Toto&  t;
      public:
      A( Toto & a_t) : t(a_t )
      {}
      ~A() {
       }
    };
```

# L'agrégation externe par pointeur

- Initialisation non obligatoire

➔ Non respect de l'OO

- Convention

➔ Celui qui alloue libère

- Quel accès externe pour t ?

```
1: using namespace std;
2:   class A {
       private:
       Toto*  t;
       public:
       A( Toto & a_t) : t( &a_t )
       {}
       A( Toto * a_t ) : t(a_t )
       {}

       ~A() {}
     };
```

# L'agrégation : accès au membre

- Quel accès externe pour t ?

```
1: using namespace std;
2:   class A {
       private:
       Toto*  t;
       public:
       A( Toto & a_t):  t( &a_t )
       {}
       A( Toto * a_t ): t(a_t )
       {}
       ~A() {}


};
```

# L'agrégation : accès au membre

- Quel accès externe pour t ?
- Règle générale :

➔ renvoyer une référence constante

```
1: using namespace std;
2:   class A {
       private:
       Toto*  t;
       public:
       A( Toto & a_t):  t( &a_t )
       {}
       A( Toto * a_t ): t(a_t )
       {}
       ~A() {}
      Toto const & getToto()
      {  return *t; }
     };
```

- Cours 1
- Cours 2
- Cours 3
- Cours 4
    - Templates
    - Exception

# Introduction aux templates

- Problématique Comment rendre le code générique et indépendant des types primitifs ?

- Héritage et composition
  - Réutilisation des objets
  - Mécanisme dynamique à l'éxecution
- Templates
  - Réutiliser le code source
  - Mécanisme statique à la compilation

# Un exemple simple : nombre complexe

```
class Complexe {
  private:
  double  _img;
  double  _real;
  public:
  Complexe( double img, double real)
    :  _img( img ), _real( real )
  {}

  double img() const { return _img; }
  double real() const { return _real; }
  // IMPLEMENTATION CONTINUE
};
```

double ou des long double à la place de float ?

Solutions:

1\ Dupliquer le code ☹

2\ C++ Templates à la rescousse

➔ Duplication automatisée ☺

# Un exemple simple : nombre complexe

```
class Complexe {
  private:
  double  _img;
  double  _real;
  public:
  Complexe( double img, double real)
    :  _img( img ), _real( real )
  {}

  double img() const { return _img; }
  double real() const { return _real; }
  // IMPLEMENTATION CONTINUE
};
```

double ou des long double à la place de float ?

# Un exemple simple : nombre complexe

```
class Complexe {
  private:
  double  _img;
  double  _real;
  public:
  Complexe( double img, double real)
    :  _img( img ), _real( real )
  {}

  double img() const { return _img; }
  double real() const { return _real; }
  // IMPLEMENTATION CONTINUE
};
```

double ou des long double à la place de float ?

Solutions:

1\ Dupliquer le code ☹

2\ C++ Templates à la rescousse

➔ Duplication automatisée ☺

# Un exemple simple : nombre complexe

```
template< typename T >
class Complexe {
  private:
  T  _img;
  T  _real;
  public:
  Complexe( T img, T real)
    :  _img( img ), _real( real )
  {}

  T img() const { return _img; }
  T real() const { return _real; }
  // IMPLEMENTATION CONTINUE
};
```

Type primitif paramétré

# Un exemple simple : nombre complexe

complexe.hpp

```
template< typename T >
class Complexe {
  private:
  T  _img;
  T  _real;
  public:
  Complexe( T img, T real)
     :  _img( img ), _real( real )
  {}

  T img() const { return _img; }
  T real() const { return _real; }
};
```

main.cpp

```
#include "complexe.hpp"
int main(int argc, char* argv[] ) {
 Complexe<double> c1(1.0,-2.0);
 Complexe<int> c2(1.0,-2.0);
 cout << " c1 = " << c1.img() << " " << c1.real();
 return 0;
}
```

Template instancié

Type primitif paramétré

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# Séparation implémentation / déclaration

complexe.hpp

```
template< typename T >
class Complexe {
  private:
  T   _img;
  T   _real;
  public:
  Complexe( T img, T real);
  T img() const;
  T real() const;
};
// IMPLEMENTATION CONTINUE A droite
```

complexe.hpp

```
// SUITE IMPLEMENTATION


template < typename T >
Complexe<T>::Complexe( T img, T real )
 : _img(img), _real( real )
{ }


template < typename T > T Complexe<T>::img() const
{ return _img; }


// ETC
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# La problématique du conteneur et du contenu

- std::vector peut-elle accepter n'import quel contenu?
  - Opérations d'accès  indépendantes du contenu ?

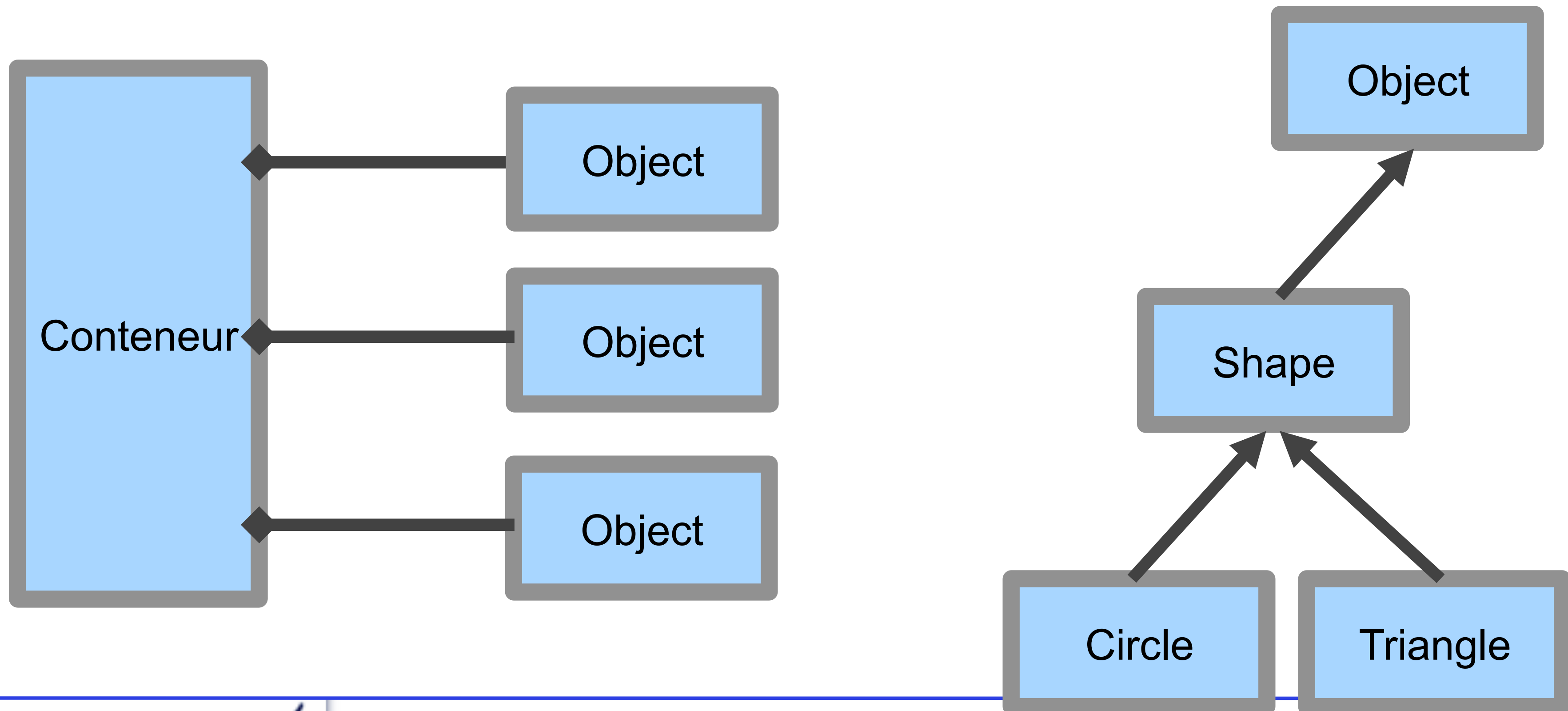
```
std::vector<string> v1; std::vector<float>  v3;
v1[0] = ""; v3[1] = 1.0f;
v1.push_back(std::string("TOTO") ); v3.push_back( 3.0f) ;
```

➔ Quelle implémentation?

- Solution Objet:
  - Hiérarchie tout objet (SmallTalk, Java)

# La problématique du conteneur et du contenu

Solution Objet: Hiérarchie tout objet (Java,…)

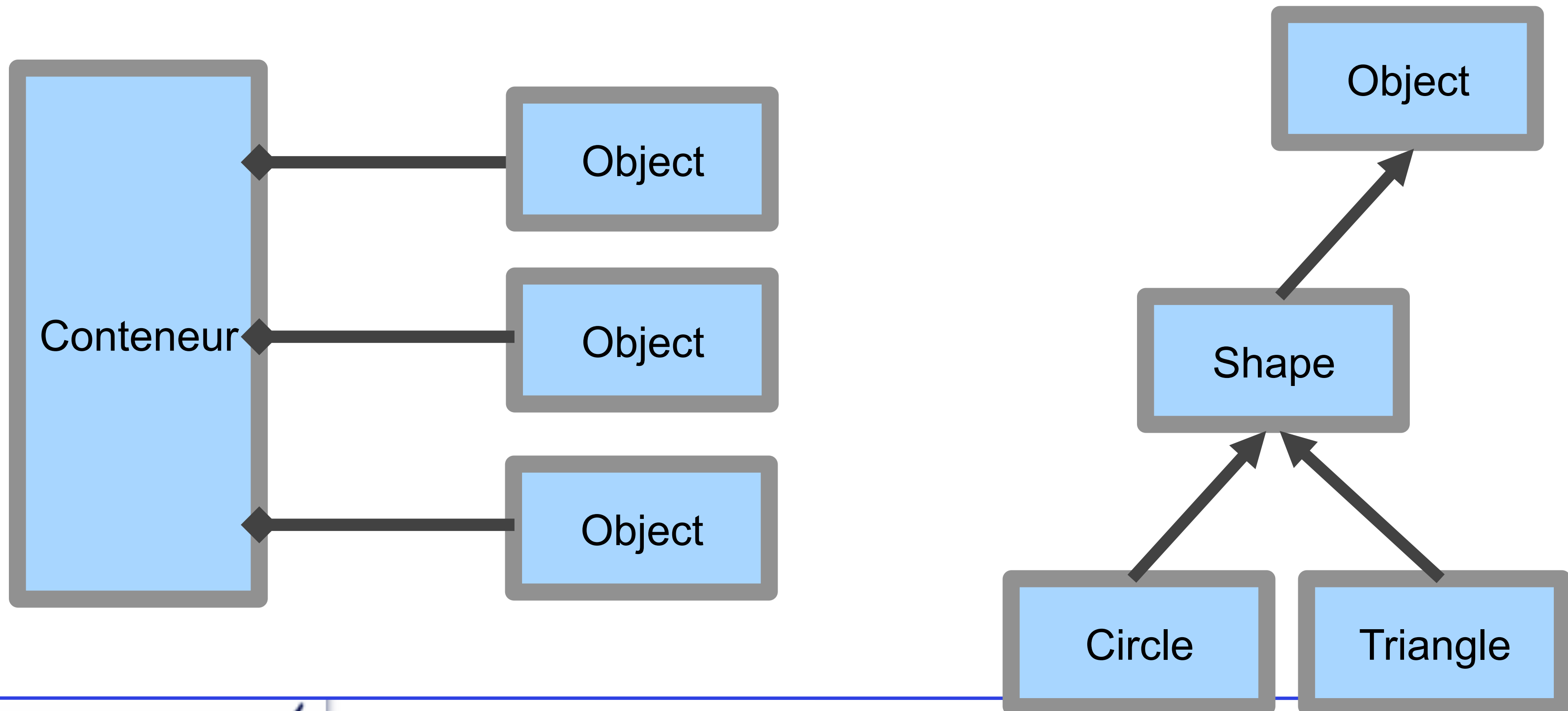



INSTITUT d'OPTIQUE GRADUATE SCHOOL

# La problématique du conteneur et du contenu

Solution Objet: Hiérarchie tout objet (Java,…)

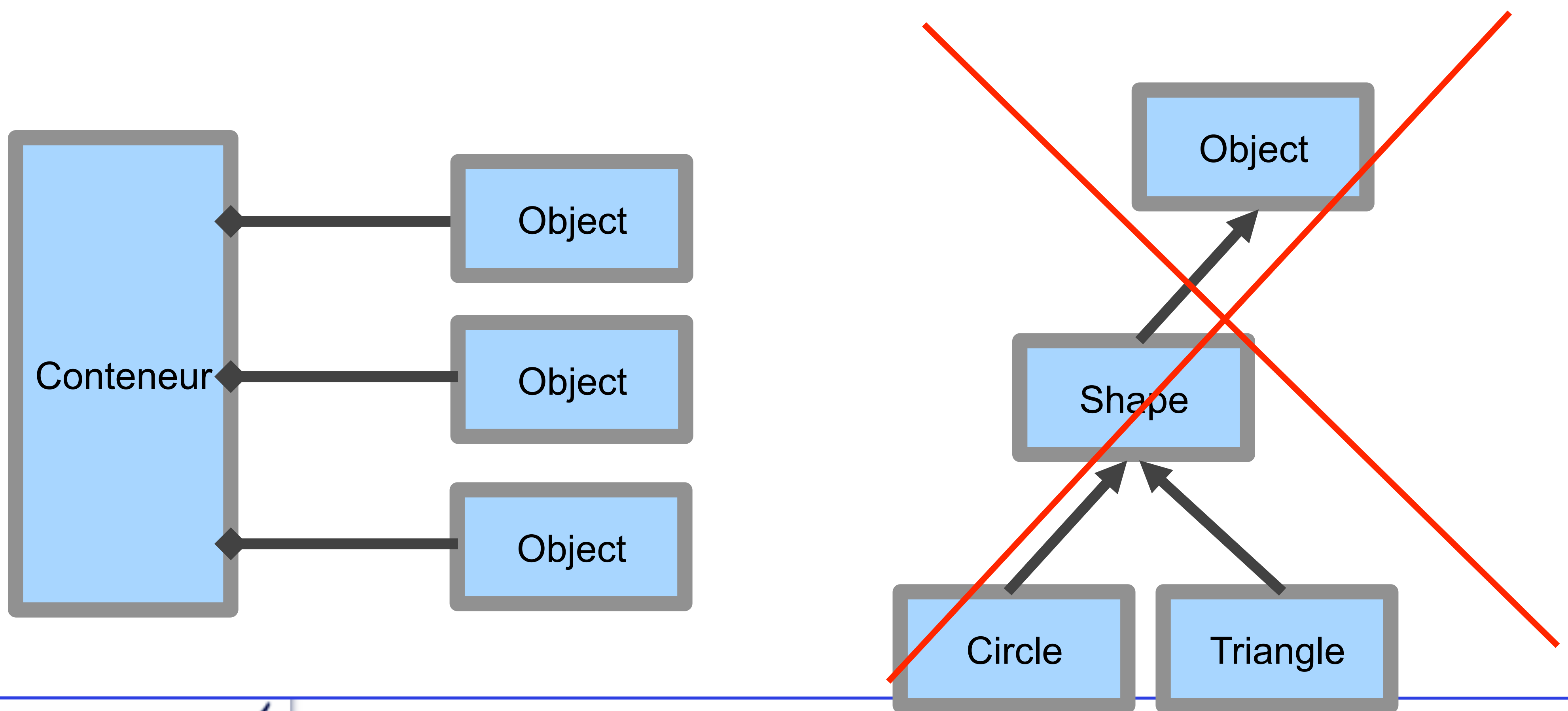
Problème: En C++, l'héritage multiple existe et pas de super Object



# La problématique du conteneur et du contenu

Solution Objet: Hiérarchie tout objet (Java,...)

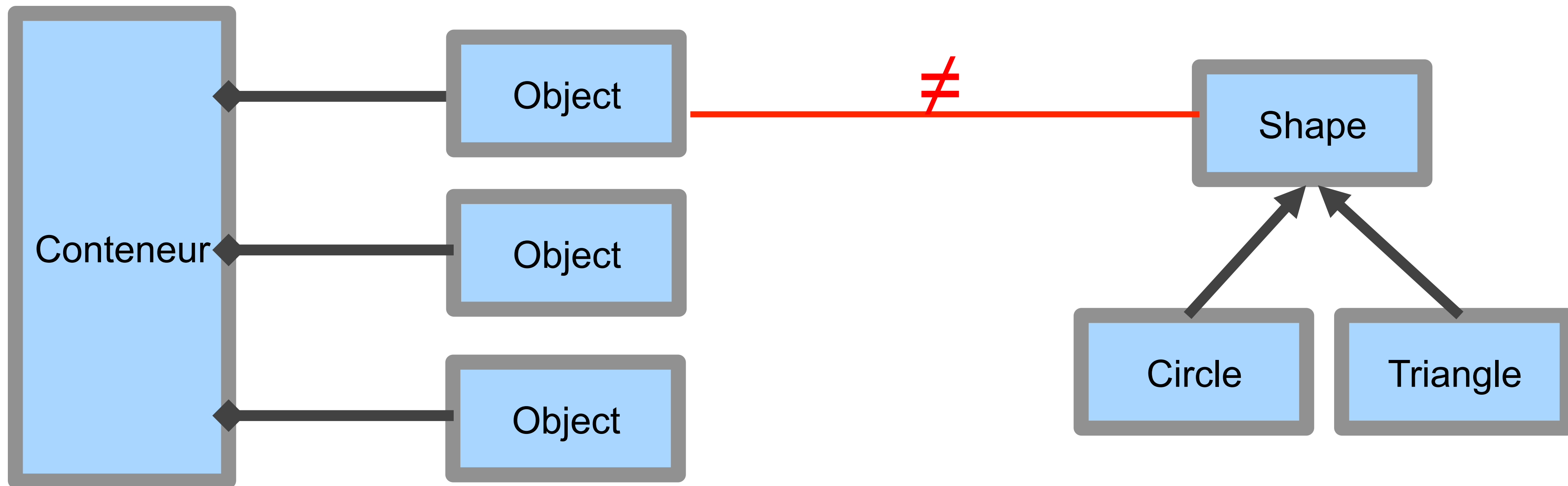
Problème: En C++, l'héritage multiple existe et pas de super Object



# La problématique du conteneur et du contenu

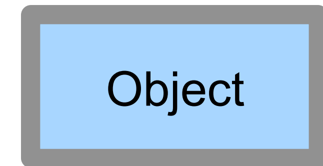
Solution Objet: Hiérarchie tout objet (Java,…)

Problème: En C++, l'héritage multiple existe et pas de super Object

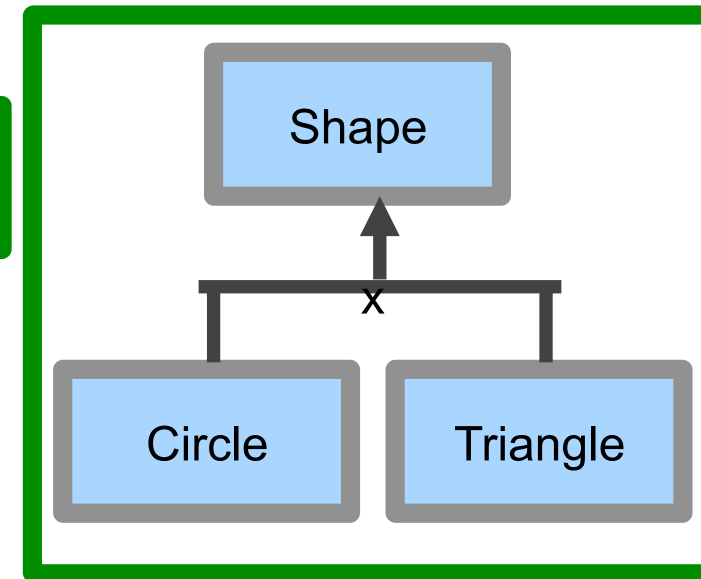


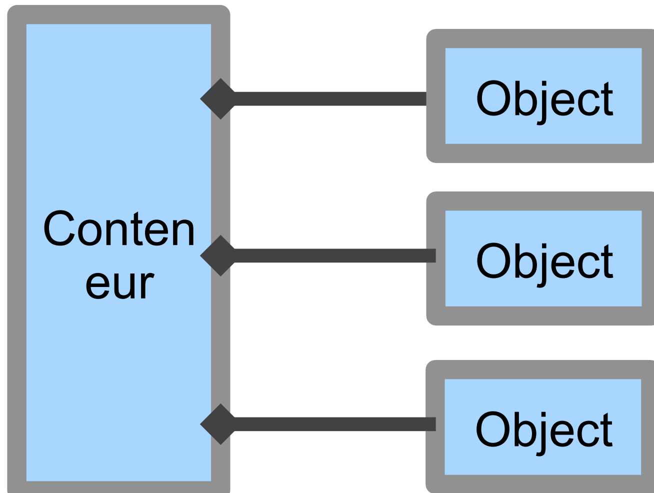
# La problématique du conteneur et du contenu

## Simulation C++ d'une hiérarchie tout objet




Vendeur EXTERNE



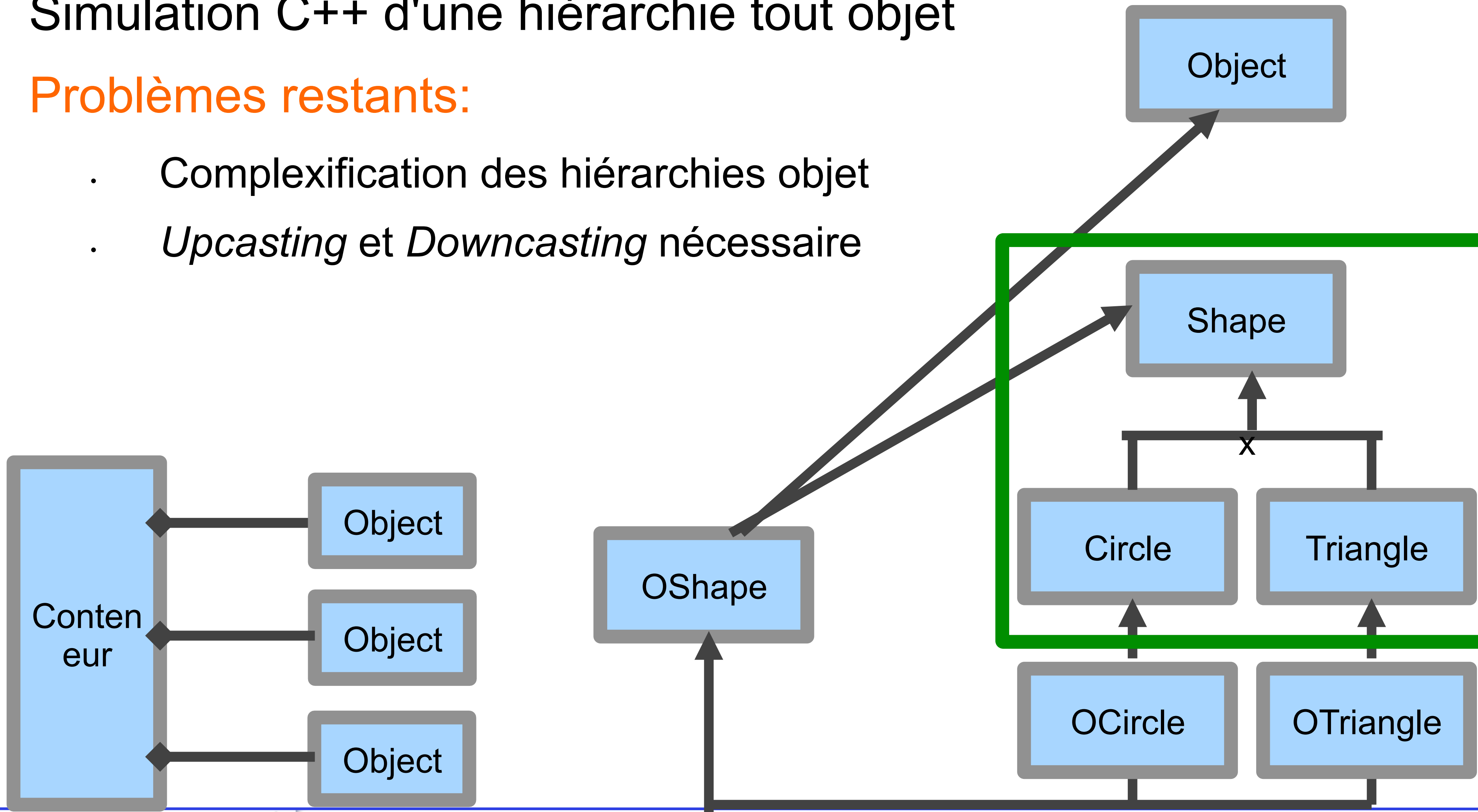



# La problématique du conteneur et du contenu

Simulation C++ d'une hiérarchie tout objet

## Problèmes restants:

- Complexification des hiérarchies objet
- *Upcasting* et *Downcasting* nécessaire

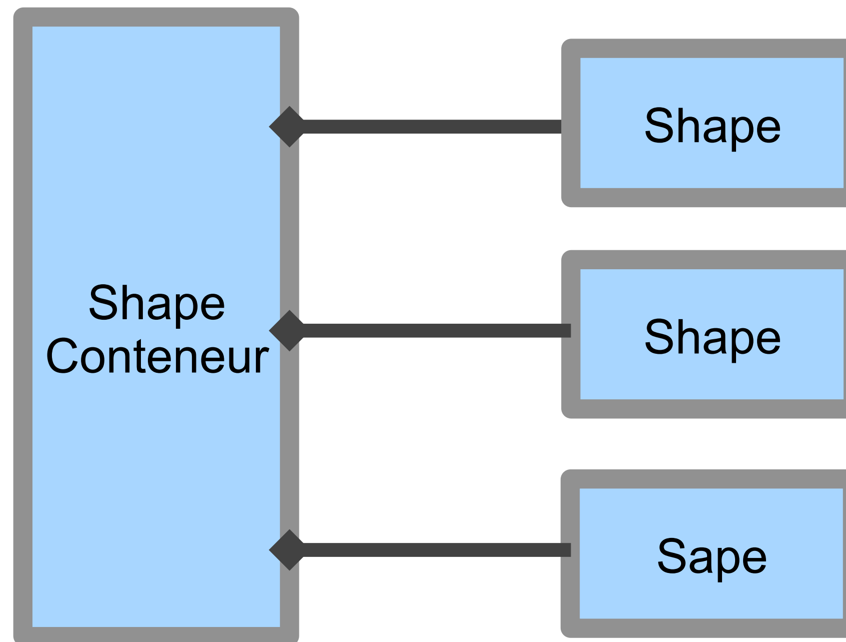



INSTITUT d'OPTIQUE GRADUATE SCHOOL

# La problématique du conteneur et du contenu

## Solution Template

➔ Définit à la compilation un conteneur approprie



# Conteneur *templaté*

```
template< typename T>
class Array {
  private:
  T  _data[100];
  public:
  Array(){}
  T const &
   operator[](unsigned int index) const
  { return _data[index]; }


  T&
   operator[](unsigned int index)
  { return _data[index]; }
};
```

Comment mettre une classe?

# Conteneur *templaté*

```
template< class T>
class Array {
  private:
  T  _data[100];
  public:
  Array(){}
  T const &
   operator[](unsigned int index) const
  { return _data[index]; }


  T&
   operator[](unsigned int index)
  { return _data[index]; }
};
```

Comment mettre une classe?

➔ Mot-clé *class*

Bénéfice : le conteneur peut contenir type primitif ou type objet

# Conteneur *templaté*

```
template< class T, unsigned int SIZE >
class Array {
  private:
  T  _data[SIZE];
  public:
  Array(){}
  T const &
   operator[](unsigned int index) const
  { return _data[index]; }


  T&
   operator[](unsigned int index)
  { return _data[index]; }
};
```

Comment spécifier la taille du conteneur à la compilation ?

→ paramètre template constant

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# Conteneur *templaté*

```
template< class T,
          unsigned int SIZE = 100>
class Array {
  private:
  T  _data[SIZE];
  public:
  Array(){}
  T const &
   operator[](unsigned int index) const
  { return _data[index]; }


  T&
   operator[](unsigned int index)
  { return _data[index]; }
};
```

Comment spécifier la taille du conteneur à la compilation ?

- Paramètre template constant
- Peut prendre une valeur par défaut

# Paramètre template template

```
template< class T,
          ??????????????????????????? >
class Stack{
  private:
  CONT<T>  data;
  public:
  Stack(){}
  void push( T const & );
  T    top() const;

};
```

Comment spécifier que le paramètre d'un template peut lui même être un template ?

→ *template template parameter*

But: S'affranchir du "genre" du conteneur ET du type du contenu

Exemples:

- List, vector, map,…

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# Paramètre template template

```
template< class T,
         template< class T2 > class CONT >
class Stack{
  private:
  CONT<T2>  data;
  public:
  Stack(){}
  void push( T const & );
  T    top() const;

};
```

Comment spécifier que le paramètre d'un template peut lui même être un template ?

➔ *template template parameter*

But:  S'affranchir du "genre" du conteneur ET du type du contenu

Exemples:

- List, vector, map,…

# La Standard Template Library (STL)

- Entrées sorties
- Limites numériques
- Exceptions
- Algorithmes
- Conteneurs
  - list<string>, map<string,int>, vector,set, queue,
  - Parcours, Accès, Ajout d'éléments indépendants

  ➔ Itérateurs

# Conteneurs et itérateurs

Pour conteneur<T>, 4 types d'iterateurs

- T*, T const * :
  - iterator, const_iterator
  - reverse_iterator,  const_reverse_iterator
- Exemple:

# Conteneurs et itérateurs

- Exemple:

```
std::vector<string> students;
std::vector<string>::const_iterator it;
for( it  = students.begin(); it!= students.end(); ++it) {
   cout << (*it) << endl;
}
```

# Polymorphisme et templates

- Il est possible de mixer les templates
  - Héritage
  - Polymorphisme

# Le Concept *Exception*

- Gestion des erreurs et où?
  - Librairie
    - Retourner des Codes d'erreurs
      - Convention nécessaire
      - Ignorables
  - Développeur
    - Imparfait puisqu'humain ☹
  - Compilateur
    - Incomplet (erreur à l'execution vs erreur à la compilation)
  - Langage
    - Gérer les erreurs depuis le langage ➔ Exception

# Exception

- Objet lancé (*thrown*) si erreur détectée
- Objet qui peut être rattrapé (*catch*)
- Objet
  - ➔ Hiérarchie d'Exception
  - ➔ Traitement spécifique (*ExceptionHandler)*
- Code plus lisible
  - Séparation du flot normal vs. flot gestion de l'erreur
- Impossible d'ignorer les exceptions
- Formellement concept non-objet.

# Exception en C++

```
try {
// Code susceptible de lancer une
   exception
 f(3.0);
}
catch( Type_Exception & ref_exception )
{
// Code de gestion d'exception
  throw; // Relance d'exception
}
```

# Exception en C++

```
try {
// Code susceptible de lancer une
   exception
 f(3.0);
}
catch( Type_Exception & ref_exception )
{
// Code de gestion d'exception
  throw; // Relance d'exception
}
```

peut être remplacé par : …

➔ Toutes les exceptions sont interceptées

Que se passe-t-il après le lancement d'une exception ?

Absence de try/cacht

➔ *unexpected/uncaught exception*

Par défaut prog. s'arrête

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# Déclenchement d'exception non attrapée

*Throw*

Que se passe-t-il après le lancement d'une exception ?

Absence de try/cacht ➔ *unexpected exception*

Par défaut prog. appelle *terminate()* → *abort()*

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw(); // C++98
terminate_handler set_terminate(terminate_handler f) noexcept; // C++11

void terminate();  // C++98
void terminate() noexcept;    // C++11
```

# *Terminate* personnalisée

```
#include <exception>
using namespace std;
void myQuit() {
  cout << "Terminating due to uncaught exception\n";
  exit(5);
}
set_terminate(myQuit);
```

# std::exception

```
class exception
{
   // code omis
   virtual char const *
   what () const throw ()
   {
     return "ptr vers chaine" ;
   }
};
```

- Classe fournie par la STL
- Définir vos exceptions en héritant de std::exception et surchargant en redéfinissant la méthode what()

# std::exception

```
class exception
{
   // code omis
   virtual char const *
   what () const throw ()
   {
     return "ptr vers chaine" ;
   }
};
```

- Classe fournie par la STL
- Définir vos exceptions en héritant de std::exception et surchargant en redéfinissant la méthode what()

# std::exception

```
class exception
{
   // code omis
   virtual char const *
   what () const throw ()
   {
     return "ptr vers chaine" ;
   }
};

class Toto{
  void f() throw (MyException)
  { throw MyException(); }
};
```

- Classe fournie par la STL
- Définir vos exceptions en héritant de std::exception et surchargant en redéfinissant la méthode what()
- Spécificateur d'exception

(ignorable en C++11)

INSTITUT d'OPTIQUE GRADUATE SCHOOL

# unexpected vs uncaught exception

- *uncaught*
  - L'exception lancée n'a pas été attrapée dans un bloc try/catch
- *unexpected*
  - L'exception lancée ne correspond pas au type spécifiée par le spécificateur d'exception (throw (TYPE_SPECIFIE) )
- *unexpected* arrive avant *uncaught*

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();   // C++98
unexpected_handler set_unexpected(unexpected_handler f) noexcept; // C++11
```

# Questions