Arguments par défaut

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   void cleanup();
9:   float setElem (float f ,
                    int index = 0 );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

args par défaut existent :

- Pour les constructeurs
  - Remplace le constructeur par défaut

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Arguments par défaut

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   void cleanup();
9:   float setElem (float f ,
                    int index = 0 );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

- Pour les constructeurs
  - Remplace le constructeur par défaut
- Pour les méthodes

➔ Alternative à la surcharge

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Arguments par défaut

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   void cleanup();
9:   float setElem (float f ,
                    int index = 0 );
10: void display();
11: private:
12: void reallocate();
13:};
14:#endif
```

- Pour les constructeurs
  - Remplace le constructeur par défaut
- Pour les méthodes

➔ Alternative à la surcharge

- Les arguments par défaut doivent se trouver à la fin
- Les fichiers .cpp sont inchangés

Constructeur de copie : problématique

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:
5:   DynTab tab ( 100 );
6:   DynTab tab2( tab );
7:   return 0;
8:  }
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Constructeur de copie : problématique

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:
5:   DynTab tab ( 100 );
6:   DynTab tab2( tab );
7:   return 0;
8:  }
```

Constructeur de copie :

- Nécessite Référence constante
- Existe un par défaut
  - Dangereux !
  - Copie bit à bit

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Constructeur de copie : syntaxe

cdyntab.hpp

```
1:class DynTab {
2:   private:
3:   int size;
4:   float* data;
5:
6:   public:
7:   DynTab(int size  = 100);
8:   DynTab(DynTab const & a );
9:
10: ~DynTab();
11:
12:};
13:#endif
```

cdyntab.cpp

```
#include "cdyntab.hpp"
DynTab::DynTab(int a_size)
: size(a_size), data( new float[size])
{ }

DynTab::DynTab(DynTab const & a_ref)
 : size(a_ref.size),
   data( new float[size])
{
   // ET ICI ?
}
DynTab::~DynTab () {
  delete[] data;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Propriétés des références

- Pointeur constant auto. déférencé

⇔ Pointeur sûr et fiable

- Une référence (&) est
  - Obligatoirement initialisée à sa création
  - Liée à une allocation (type primitif ou objet)
  - Exemple:

Rappel : Propriétés des références

- Pointeur constant auto. déférencé

⇔ Pointeur sûr et fiable

- Une référence (&) est
  - Obligatoirement initialisée à sa création
  - Liée à une allocation (type primitif ou objet)
- Une référence n'est jamais
  - égale à NULL
  - effacée par l'opérateur delete

Rappel: références

```
1: #include "cdyntab.hpp"
2: using namespace std;
3: int main(int argc, char* argv[] ) {
4:
5:    int foo = 5;
6:    int & r_foo = foo;
7:    cout << foo << " " << r_foot;
8:    r_foot++;
9:    cout << foo << " " << r_foot;
10:
     return 0; }
```

Référence sur un entier

foo et r_foot sont incrémentées

➔ r_foot est liée à l'allocation de foot

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Le retour des références

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:
5:    DynTab tab ( 100 );
6:    DynTab tab2( tab );

7:    std::string une_chaine;
8:    std::vector<int> un_vecteur;
      return 0;
}
```

Références :

- sur un objet de la classe DynTab

- objet de la classe std::string

- objet de la classe std::vector

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Propriétés des références

- Pointeur constant auto. déférencé

⇔ Pointeur sûr et fiable

- Une référence (&) est
  - Obligatoirement initialisée à sa création
  - Liée à une allocation (type primitif ou objet)

Utilisation des références

- Arguments
  - Fonction
    - ➔ cf. TD et fonction swap
  - Constructeur
  - Méthodes
- Retour de fonctions

Retourner des références

```
1: #include "cdyntab.hpp"
2: int& g( int & x ) {
3:   x++;
4: return x; }
5:
6:
7:
8:
9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: g(a);
12: return 0; }
```

L'appel à g(a):

Retourner des références

```
1: #include "cdyntab.hpp"
2: int& g( int & x ) {
3:   x++;
4: return x; }
5:
6:
7:
8:
9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: g(a);
12: return 0; }
```

L'appel à g(a):

- la variable a est passée et retournée par référence
- ➔ Impossible de le savoir
- ➔ La syntaxe le "cache"

Retourner des références

```
1: #include "cdyntab.hpp"
2: int& g( int & x ) {
3:   x++;
4: return x; }
5: int&  h() {
6:  int q;
7:  return q;
8: }
9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: g(a);
12: return 0; }
```

Retourner des références

```
1: #include "cdyntab.hpp"
2: int& g( int & x ) {
3:   x++;
4: return x; }
5: int&  h() {
6:  int q;
7:  return q;
8: }
9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: g(a);
12: return 0; }
```

Erreur q sera détruit à la sortie de h()

Référence sur objet constant

```
1: #include "cdyntab.hpp"
2: int& g( int const & x ) {
3:   x++;
4: return x; }

5: void  h2(int & x) { }

9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: h2(3);
12: return 0; }
```

Référence sur objet constant

```
1: #include "cdyntab.hpp"
2: int& g( int const & x ) {
3:   x++;
4: return x; }

5: void  h2(int & x) { }

9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: h2(3);
12: return 0; }
```

Erreur x ne peut être modifié !!!

➔ x est une référence sur objet constant

Référence sur objet constant

```
1: #include "cdyntab.hpp"
2: int& g( int const & x ) {
3:   x++;
4: return x; }

5: void  h2(int & x) { }

9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: h2(3);
12: return 0; }
```

Référence sur objet constant

```
1: #include "cdyntab.hpp"
2: int& g( int const & x ) {
3:   x++;
4: return x; }

5: void  h2(int & x) { }

9: int main(int argc, char* argv[] ) {
10: int a = 0;
11: h2(3);
12: return 0; }
```

Erreur

- 3 est aussi une référence constante

Passage des arguments

- Que faut-il privilégier ?
  - Par valeur ou par référence (= par adresse)

  ➔ par défaut privilégiez une référence constante
- ➔ Empêche la modification de l'argument dans la boucle/fonction.
- ➔ Efficacité sur les gros objets
  - ➔ Par valeur nécessite : Constructeur + Destructeur
  - ➔ Par référence : juste une sauvegarde d'adresse

Création d'objets

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:
5:    DynTab tab ( 100 );              //Constructeur avec un arg. de type int
6:    DynTab tab2( tab );              //Constructeur par copie
7:    DynTab tab22 = tab2;
8:    tab  =  tab2;
9:
10:
11:   return 0;
}
```

Création d'objets

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:
5:    DynTab tab ( 100 );            //Constructeur avec un arg. de type int
6:    DynTab tab2( tab );            //Constructeur par copie
7:    DynTab tab22 = tab2;           //Constructeur par copie appelé implicitement
8:    tab  =  tab2;
9:
10:
11:   return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Création d'objets

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:
5:    DynTab tab ( 100 );            //Constructeur avec un arg. de type int
6:    DynTab tab2( tab );            //Constructeur par copie
7:    DynTab tab22 = tab2 ;          //Constructeur par copie appelé implicitement
8:    tab  =  tab2;                  // ????
9:
10:
11:   return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Création d'objets

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:
5:    DynTab tab ( 100 );            //Constructeur avec un arg. de type int
6:    DynTab tab2( tab );            //Constructeur par copie
7:    DynTab tab22 = tab2;           //Constructeur par copie appelé implicitement
8:    tab  =  tab2;                  // ➔ OPERATEUR =
9:
10:
11:   return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Surcharges des opérateurs

- *Syntaxic sugar*

  ⇔ Équivalent à un appel de fonction !

  `TYPE_RETOUR opérateur@( ARGUMENT)`

- Où @ représente (+, -, +=, /, ->, etc)
- Définition et déclaration
  - une fonction de classe == méthode
  - une fonction externe ou friend
    - *friend* (cf. TD e.g., operator<<)

Surcharges des opérateurs

- Impossible
  - de changer l'ordre de priorité des opérateurs
  - le nombre d'arguments requis
  - redéfinir . ou .* ou encore **
  - Ajouter ses propres opérateurs

- Opérateur =

➔ joue un rôle particulier

Opérateur =

- Nécessairement implémentée en méthode

⇔ membre de classe

```
5:    DynTab tab ( 100 );              //Constructeur avec un arg. de type int
6:    DynTab tab2( tab );              //Constructeur par copie
7:    tab  =  tab2;                    // ➔ OPERATEUR =
```

- l'opérateur =
  - Copie des données de tab2 dans tab

Opérateur =

```
1:class DynTab {
2:  private:
3:  int size; float* data;
6:  public:
7:  DynTab(int size  = 100);
8:  DynTab(DynTab const & a );
10: ~DynTab();
11: DynTab&  operator=(DynTab const &t) {
12:     size = t.size;
13:     delete[] data;
14:     data = new data[size];
15:     //Copie du contenu t.data dans data
16:     return *this;
17: }
19:};
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Opérateur =

```
1:class DynTab {
2:  private:
3:  int size; float* data;
6:  public:
7:  DynTab(int size  = 100);
8:  DynTab(DynTab const & a );
10: ~DynTab();
11: DynTab&  operator=(DynTab const &t) {
12:     size = t.size;
13:     delete[] data;
14:     data = new data[size];
15:     //Copie du contenu t.data dans data
16:     return *this;
17: }
19:};
```

- Copie du contenu
- Renvoie de *this
- tab = tab2

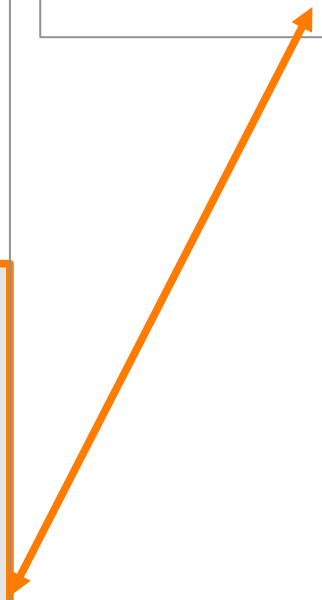
Pourquoi renvoyer *this?

Précaution avec Opérateur =

```
1:class DynTab {
2:   private:
3:   int size; float* data;
6:   public:
7:   DynTab(int size  = 100);
8:   DynTab(DynTab const & a );
10:  ~DynTab();
11:  DynTab&  operator=(DynTab const &t) {
12:      size = t.size;
13:      delete[] data;
14:      data = new data[size];
15:      //Copie du contenu t.data dans data
16:      return *this;
17:  }
19:};
```

main.cpp

```
5:    DynTab tab ( 100 );
6:    DynTab tab2( tab );
7:    tab  =  tab;
```



Précaution avec Opérateur =

```
1:class DynTab {
2:   private:
3:   int size; float* data;
6:   public:
7:   DynTab(int size  = 100);
8:   DynTab(DynTab const & a );
10:  ~DynTab();
11:  DynTab&  operator=(DynTab const &t) {
12:      size = t.size;
13:      delete[] data;
14:      data = new data[size];
15:      //Copie du contenu t.data dans data
16:      return *this;
17:  }
19:};
```

main.cpp

```
5:    DynTab tab ( 100 );
6:    DynTab tab2( tab );
7:    tab  =  tab;
```

OUCH !!!!

➔ DONNEES INEXISTANTES

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Précaution avec Opérateur =

```
1:class DynTab {
2:   private:
3:   int size; float* data;
6:   public:
7:   DynTab(int size  = 100);
8:   DynTab(DynTab const & a );
10:  ~DynTab();
11:  DynTab&  operator=(DynTab const &t) {
12:      if( &t == this ) { return *this; }
13:      size = t.size;
13:      delete[] data;
14:      data = new data[size];
15:      //Copie du contenu t.data dans data
16:      return *this;
17:  }
19:};
```

main.cpp

```
5:    DynTab tab ( 100 );
6:    DynTab tab2( tab );
7:    tab  =  tab;
```

Solution: éviter l'auto-assignement

Vérification que l'adresse de t est différente de l'appelant

Opérateur ++ et --

Règles du compilateur

- ++a ⇔ operator++(a)
- a++ ⇔ operator++(a, int)

La surcharge des opérateurs

➔ Expérimentations en TD

Création dynamique d'objets

Comment faire quand on ne connaît pas à l'avance le nombre d'objet dont on a besoin ?

➔Allocation dynamique

- Opérateurs new et delete et delete[]
- Opérateur new
  - Garantit l'appel au constructeur
  - Initialisation du pointeur this à la bonne adresse
  - Allocation sur le tas
- Attention aux fuites mémoires

Création dynamique d'objets

- Opérateur new
  - Garantit l'appel au constructeur
  - Initialisation du pointeur this à la bonne adresse
  - Allocation sur le tas = free store

  ➔ A chaque new doit exister un et un seul delete

Création dynamique d'objets

```
1: #include "cdyntab.hpp"
2:
3: int main(int argc, char* argv[] ) {
4:    DynTab tab ( 100 );                  // Allocation sur la pile
5:    DynTab* tab4 = new DynTab( 300 ); // Allocation sur le tas
6:    // Utilisation de tab4
7:    DynTab & r_tab4 = *tab4;             // Référence à partir d'un objet dynamique
8:    r_tab4.resize( 100 );                // Appel de méthode via référence
9:    tab4->resize( 200 );                 // vs appel de méthode via pointeur
10:   delete tab4;                         // libération mémoire NECESSAIRE
11:   //delete r_tab4;                     // ERREUR
12:   return 0;
}
```

Création dynamique d'objets : dans une fonction

```
1: #include "cdyntab.hpp"
2:
3: DynTab& foo() {
4:  DynTab* a = new DynTab( 100 );
5:  DynTab* b = new DynTab();
6:  return *b;
7: }
3: int main(int argc, char* argv[] ) {
9:   DynTab* tab4 = new DynTab( 300 ); // Allocation sur le tas = free store
10:  delete tab4;                      // Libération mémoire OK
11:  DynTab & tab5 = foo();
12:  delete &tab5;
13:  return 0;
}
```

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Création dynamique d'objets : dans une fonction

```
1: #include "cdyntab.hpp"
2:
3: DynTab& foo() {
4:  DynTab* a = new DynTab( 100 );
5:  DynTab* b = new DynTab();          // Allocation sur le tas
6:  return *b;
7: }
3: int main(int argc, char* argv[] ) {
9:   DynTab* tab4 = new DynTab( 300 ); // Allocation sur le tas = free store
10:  delete tab4;                      // Libération mémoire OK
11:  DynTab & tab5 = foo();            // OK référence initialisée
12:  delete &tab5;
13:  return 0;
}
```

Création dynamique d'objets : dans une fonction

```
1: #include "cdyntab.hpp"
2:
3: DynTab& foo() {
4:  DynTab* a = new DynTab( 100 );
5:  DynTab* b = new DynTab();          // Allocation sur le tas
6:  return *b;
7: }
3: int main(int argc, char* argv[] ) {
9:   DynTab* tab4 = new DynTab( 300 ); // Allocation sur le tas = free store
10:  delete tab4;                      // Libération mémoire OK
11:  DynTab & tab5 = foo();            // OK référence initialisée
12:  delete &tab5;                     // Fonctionne mais mauvaise pratique
13:  return 0;
}
```

Création dynamique d'objets : dans une fonction

```
1: #include "cdyntab.hpp"
2:
3: DynTab& foo() {
4:  DynTab* a = new DynTab( 100 );   // Allocation jamais liberee = fuite memoire
5:  DynTab* b = new DynTab();          // Allocation sur le tas
6:  return *b;
7: }
3: int main(int argc, char* argv[] ) {
9:   DynTab* tab4 = new DynTab( 300 ); // Allocation sur le tas = free store
10:  delete tab4;                       // Libération mémoire OK
11:  DynTab & tab5 = foo();             // OK référence initialisée
12:  delete &tab5;                      // Fonctionne mais mauvaise pratique
13:  return 0;
}
```

Plus de mémoire?

- Opérateur new
  - Levée d'une exception s'il n'y a plus de mémoire
- *new-handler*
  - responsable pour récupérer les problèmes
  - dépend si les classes surchargent new et delete

```
1: #include "cdyntab.hpp"
2: #include <new>
3: void out_of_memory() {  std::cerr << " Memoire saturee. Bye bye " << std::endl;
4:   std::exit(-1); }
5:  int main() {
6:   std::set_new_handler( out_of_memory );
7:   while( 1 ) { count++; new int[1000]; }
       return 0;}
```

- Cours 1
- Cours 2
- Cours 3 : Relations entre Objets
  - Héritage
  - Agrégation et Composition
- Cours 4

Organisation et communication entre classes

- Héritage

  ➔ Factoriser du code entre plusieurs classes

- Composition et Agrégation

  ➔ Déléguer des actions à une autre classe.

  ➔ Collaboration ± étroite entre deux classes

Héritage

- Une classe B hérite de A

⇔ A a accès aux méthodes et aux données de B en fonction des qualificateur d'accès

- Type commun
  - Upcasting
  - Nécessaire pour les conteneur : vector<A*>
- Méthodes spécifiques

  ➔ Polymorphisme (virtual)
- Notation UML

L'héritage

- 3 formes
  - public, private et protected
- Héritage multiples possibles
- Syntaxe:

```
1: #include "cdyntab.hpp"
2:  DynTab::DynTab(int size )
3:   : A("toto"), B(3)
4:  {}
```

```
1: #ifdef DYNTAB_HPP
2: #define DYNTAB_HPP
3: class DynTab : public A,B
4: {
5:     public:
6:     DynTab( int size );
7:     // ETC
```

L'héritage

- 3 formes
  - public, private et protected
- Héritage multiples possibles
- Syntaxe:

```
1: #include "cdyntab.hpp"
2:  DynTab::DynTab(int size )
3:    : A("toto"), B(3)
3:  {}
```

```
1: #ifdef DYNTAB_HPP
2: #define DYNTAB_HPP
3: class DynTab : public A,B
4: {
5:     public:
6:     DynTab( int size );
7:     // ETC
```

L’héritage : la rédéfinition des méthodes

- Cas
  - B Hérite de A
  - B implémente méthode f()
  - A aussi

  ➔ Redéfinition de la méthode f()
- Exemple:

L'héritage : la redéfinition des méthodes

```
1: using namespace std;
2:   class A {
       private:
       int _a;
       public:
         A(int a) _a( a) {}
         void f() {  cout << " Inside A " endl; }
    };
    class B : public A {
       public:
       B() :  A(3) {}
       void f() { cout << " Inside B " endl; }
   };
```

```
int main(int argc, char* argv[] ) {
   A a1(10);
   B b1;
   b1.f();
   return 0;
}
```

L'héritage : la redéfinition des méthodes

```
1: using namespace std;
2:   class A {
       private:
       int _a;
       public:
         A(int a) : _a( a) {}
         void f() {  cout << " Inside A " endl; }
    };
    class B : public A {
       public:
       B() :  A(3) {}
       void f() { cout << " Inside B " endl; }
   };
```

?

```
int main(int argc, char* argv[] ) {
   A a1(10);
   B* b1 = new B();
   b1->f();
   return 0;
}
```

L'héritage : la redéfinition des méthodes

```
1: using namespace std;
2:   class A {
       private:
       int _a;
       public:
         A(int a) _a( a) {}
         void f() {  cout << " Inside A " endl; }
    };
    class B : public A {
       public:
       B() :  A(3) {}
       void f() { cout << " Inside B " endl; }
   };
```

```
int main(int argc, char* argv[] ) {
   A a1(10);
   B* b1 = new B();
   b1->f();
   return 0;
}
```

"Inside B" s'affiche

L'héritage : la redéfinition des méthodes

```
1: using namespace std;
2:   class A {
       private:
       int _a;
       public:
         A(int a) _a( a) {}
         void f() {  cout << " Inside A " endl; }
    };
    class B : public A {
       public:
       B() :  A(3) {}
       void f() { cout << " Inside B " endl; }
   };
```

?

```
int main(int argc, char* argv[] ) {
   A a1(10);
   // a1.f();
   B* b1 = new B();
   b1->f();
   A* b2 = new B();      // UPCASTING is safe
   b2->f();
   return 0;
}
```

L'héritage : la redéfinition des méthodes

```
1: using namespace std;
2:   class A {
       private:
       int _a;
       public:
         A(int a) _a( a) {}
         void f() {  cout << " Inside A " endl; }
    };
    class B : public A {
       public:
       B() :  A(3) {}
       void f() { cout << " Inside B " endl; }
   };
```

```
int main(int argc, char* argv[] ) {
   A a1(10);
   // a1.f();
   B* b1 = new B();
   b1->f();
   A* b2 = new B();      // Sous-typage
   b2->f();
   return 0;
}
```

"Inside A" s'affiche

L’héritage : les opérateurs

- opérateur = n'est pas hérité automatiquement

➔ Vous devez le surcharger et appeler l'opérateur = de la classe mère !!!

- Les autres opérateurs sont hérités automatiquement

Transtypage : Upcasting/Downcasting

Changement de type dans une hiérarchie

- Vers la classe mère: upcasting
  - toujours possible
  - pas besoin d'opérateur explicite
- Vers une classe fille: downcasting
  - T* a = dynamic_cast<T*>(pointeur_classe);
  - Tester si a != null
  - Ne fonctionne qu'avec une hiérarchie polymorphique

Polymorphisme: mot clé *virtual*

- Le mot clé virtual
  - Permet la surcharge
  - *Late binding*
- Méthode
- Impact sur la gestion de la mémoire
  - Destructeur non virtual

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  in f() " << endl; }
5:   virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
    cout << "Testing A a " << endl;
    a.f();
    a.g();
```

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  in f() " << endl; }
5:   virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
    cout << "Testing A a " << endl;
    a.f();
    a.g();
```

Testing A a

A: in f()

A: in g()

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  in f() " << endl; }
5:   virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
   cout << " Testing B b "  << endl;
    b.f();
    b.g();
```

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  in f() " << endl; }
5:   virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
   cout << " Testing B b "  << endl;
    b.f();
    b.g();
```

Testing B b

B. f()

B. g()

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  in f() " << endl; }
5:   virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
    A* c = new B();  //UPCASTING IS SAFE
    c->f();
    c->g();
  return 0;
}
```

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  in f() " << endl; }
5:   virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
    A* c = new B();  //UPCASTING IS SAFE
    c->f();
    c->g();
  return 0;
}
```

A: in f()

B. g()

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  In f() " << endl; }
5:   virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    virtual void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
    A* c = new B();  //UPCASTING IS SAFE
    c->f();
    c->g();
  return 0;
}
```

Polymorphisme sur les méthodes : exemple

```
class A {
    public:
    void f() const
    { cout <<  " A:  in f() " << endl; }
5:  virtual void g() const
    { cout << " A:  in g() "<< endl; }
};
```

```
class B : public A {
    public:
    void f() const
    { cout << " B. f() " << endl; }
    virtual void g() const
    { cout << "B. g() " << endl;  }
};
int main(int argc, char* argv[] ) {
    A a; B b;
    A* c = new B();  //UPCASTING IS SAFE
    c->f();
    c->g();
  return 0;
}
```

A: in f()

B. g()

Polymorphisme sur le destructeur: exemple

```
class A {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
   {  cout <<  " B. Constructor " << endl; }
    ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   A* c = new B();
   delete c;
   return 0;
```

Polymorphisme sur le destructeur: exemple

```
class A {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
   {  cout <<  " B. Constructor " << endl; }
    ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   A* c = new B();
   delete c;
   return 0;
```

A: Constructor

B. Constructor

A. Destructor

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Polymorphisme sur le destructeur: exemple

```
class A {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

A: Constructor

B. Constructor

A. Destructor

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
   {  cout <<  " B. Constructor " << endl; }
    ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   A* c = new B();
   delete c;
   return 0;
```

Polymorphisme sur le destructeur: exemple

```
class A {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    virtual ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
   {  cout <<  " B. Constructor " << endl; }
    ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   A* c = new B();
   delete c;
   return 0;
```

Polymorphisme sur le destructeur: exemple

```
class A {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    virtual ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
   {  cout <<  " B. Constructor " << endl; }
    ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   A* c = new B();
   delete c;
   return 0;
```

A: Constructor

B. Constructor

B. destructor

A. Destructor

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Polymorphisme sur le destructeur: exemple

```
class A {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    virtual ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
   {  cout <<  " B. Constructor " << endl; }
    virtual ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   A* c = new B();
   delete c;
   return 0;
```

Polymorphisme sur le destructeur: exemple

```
class A {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    virtual ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
    {  cout <<  " B. Constructor " << endl; }
    virtual ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   A* c = new B();
   delete c;
   return 0;
```

A: Constructor

B. Constructor

B. destructor

A. Destructor

Polymorphisme: mot clé *virtual*

- Méthode polymorphe ➔ *virtual*
- Constructeur n'est jamais *virtual*
- Destructeur devrait toujours être *virtual*
  - Rien ne vous y force ☹
- Le mot clé virtual ne s'utilise que dans le .hpp

Le concept d'Interface

Objectif:

- Définition d'un ensemble de méthode pour une même classe sans aucune implémentation
- Toute classe qui hérite/implémente l'interface doit garantir son implémentation

➔ Garantir un comportement en faisant abstraction des données de la classe

Le concept d'Interface

- Interface
  - Concept POO qui repose sur l'abstraction

- L'abstraction en C++
  - Méthode abstraite
  - Classe abstraite

L'abstraction = *pure virtual*

- Une méthode abstraite = *pure virtual*

➔ syntaxe déclarative: `virtual void foo() = 0;`

- Terminologie :
  - Classe abstraite :
    - Classe possèdant **au moins une** méthode *pure virtual*
  - Interface :
    - Classe possèdant **uniquement** des méthodes *pure virtual*

Classe abstraite: exemple

```
class A : public Interface {
    public:
    A( ) {
     cout << " A: Constructor " << endl;
    }
    virtual ~A() {
     cout <<  " A. Destructor " << endl;
    }
};
```

```
class Interface {
    public:
    virtual void f() const = 0;
    virtual void g() const = 0;
};
```

```
class B : public A {
    private:
    float*  _d;
    public:
    B() : A(), _d( new float[10] )
   {  cout <<  " B. Constructor " << endl; }
    virtual ~B() {
     cout << " B. destructor" << endl;
     delete[] _d;
    }
};
int main(int argc, char* argv[] ) {
   Interface* c = new B();
   c->f();
   return 0;
```

Classe abstraite: exemple

```
class A : public Interface {

    public:

    A( ) { cout << " A: Constructor " <<
    endl; }

    virtual ~A() { cout <<  " A. Destructor "
    << endl; }

    void f() const { cout <<  " A:  In f() " <<
    endl;}

    virtual void g() const { cout << " A:  in
    g() "<< endl;}

};
```

```
class Interface {

    public:

    virtual void f() const = 0;

    virtual void g() const = 0;

};
```

```
class B : public A {

    private:

    float*  _d;

    public:

    B() : A(), _d( new float[10] )
    {  cout <<  " B. Constructor " << endl; }

    virtual ~B() {

     cout << " B. destructor" << endl;

     delete[] _d;

    }

};

int main(int argc, char* argv[] ) {

   Interface* c = new B();

   c->f();

   return 0;
```

Retour sur le Coplien

Cadre pour une classe qui contient de l'allocation dynamique :

- Constructeur par défaut
- Constructeur par copie
- Destructeur (virtuel si hiérarchie)
- Surcharge de l'opérateur =

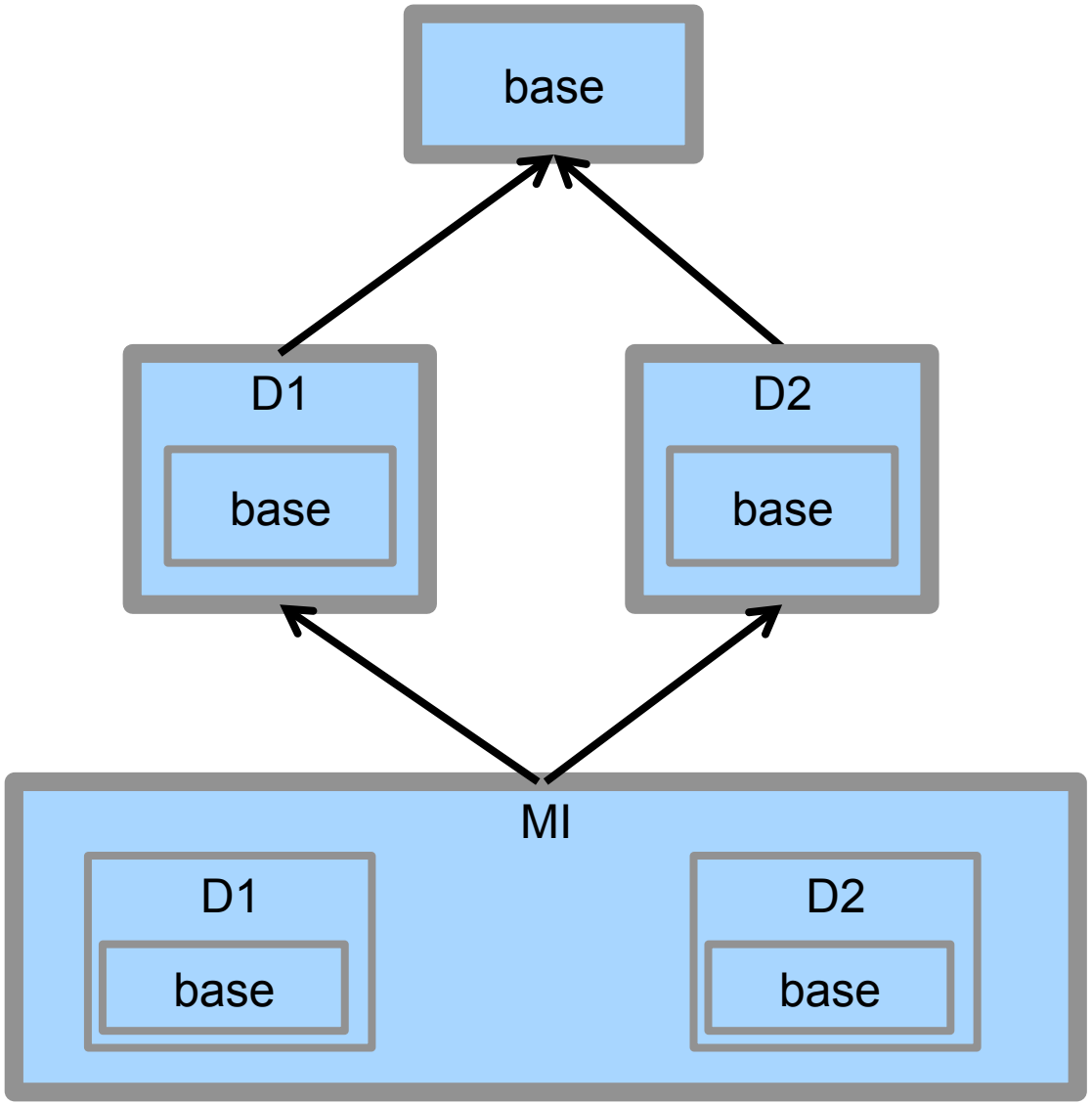
L’héritage multiple

- Possible mais à éviter

➔ Sujet/Technique avancée

➔ Ambiguités à lever

Héritage multiple et conséquences



- Upcast vers base depuis MI ?
- Quel objet base? celui de D1 ou D2?

Exemple en C++

```
class MBase {
  public:
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};
class D1 : public MBase {
public:
char* vf() const { return "D1"; }
};
class D2 : public MBase {
public:
char* vf() const { return "D2"; }
};
```

```
class MI : public D1, public D2 {
};
```

Problème 1 :

Appel à vf() est ambigü

➔ Erreur de compilation

Exemple en C++

```
class MBase {
  public:
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};
class D1 : public MBase {
public:
char* vf() const { return "D1"; }
};
class D2 : public MBase {
public:
char* vf() const { return "D2"; }
};
```

```
class MI : public D1, public D2 {
  public:
  char* vf() const { return D1::vf();}
};
```

Problème 1 :

Appel à vf() est ambigü

➔ Erreur de compilation

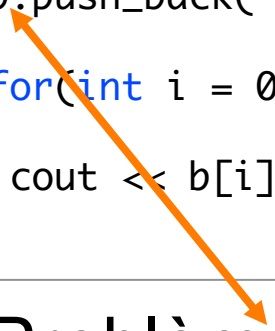
Solution :

➔ Surcharge dans MI de vf

Exemple en C++

```
class MBase {
  public:
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};
class D1 : public MBase {
public:
char* vf() const { return "D1"; }
};
class D2 : public MBase {
public:
char* vf() const { return "D2"; }
};
```

```
int main() {
 vector<MBase*> b;
 b.push_back(new D1);
 b.push_back(new D2);
 b.push_back( new MI );
 for(int i = 0; i < b.size(); i++)
  cout << b[i]->vf() << endl;
}
```



Problème 2 : Quel upcast ?

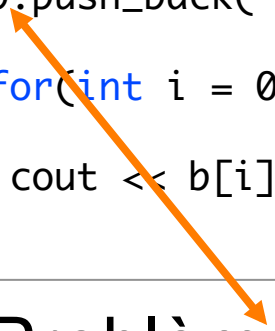
➔ Erreur de compilation

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Exemple en C++

```
class MBase {
  public:
  virtual char* vf() const = 0;
  virtual ~MBase() {}
};
class D1 : virtual public MBase {
public:
char* vf() const { return "D1"; }
};
class D2 : virtual public MBase {
public:
char* vf() const { return "D2"; }
};
```

```
int main() {
 vector<MBase*> b;
 b.push_back(new D1);
 b.push_back(new D2);
 b.push_back( new MI );
 for(int i = 0; i < b.size(); i++)
  cout << b[i]->vf() << endl;
}
```



Problème 2 : Quel upcast ?

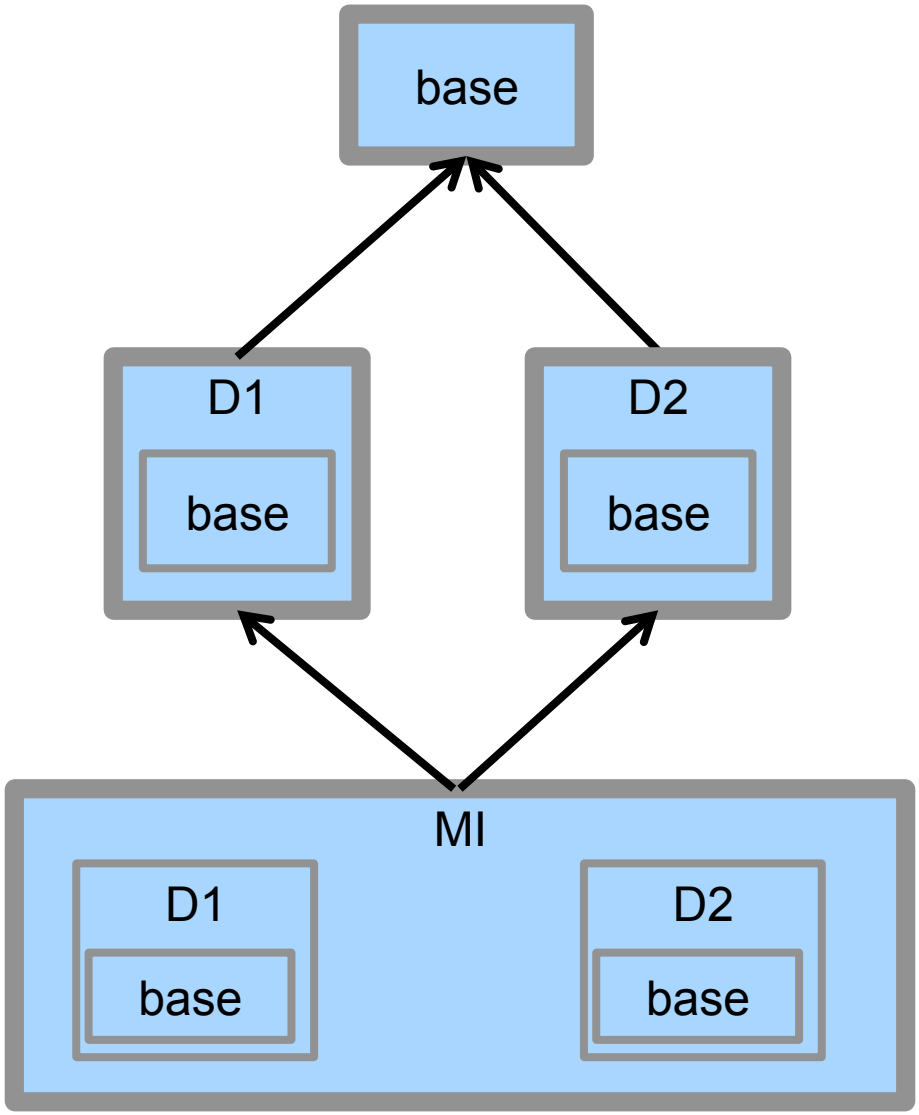
➔ Erreur de compilation

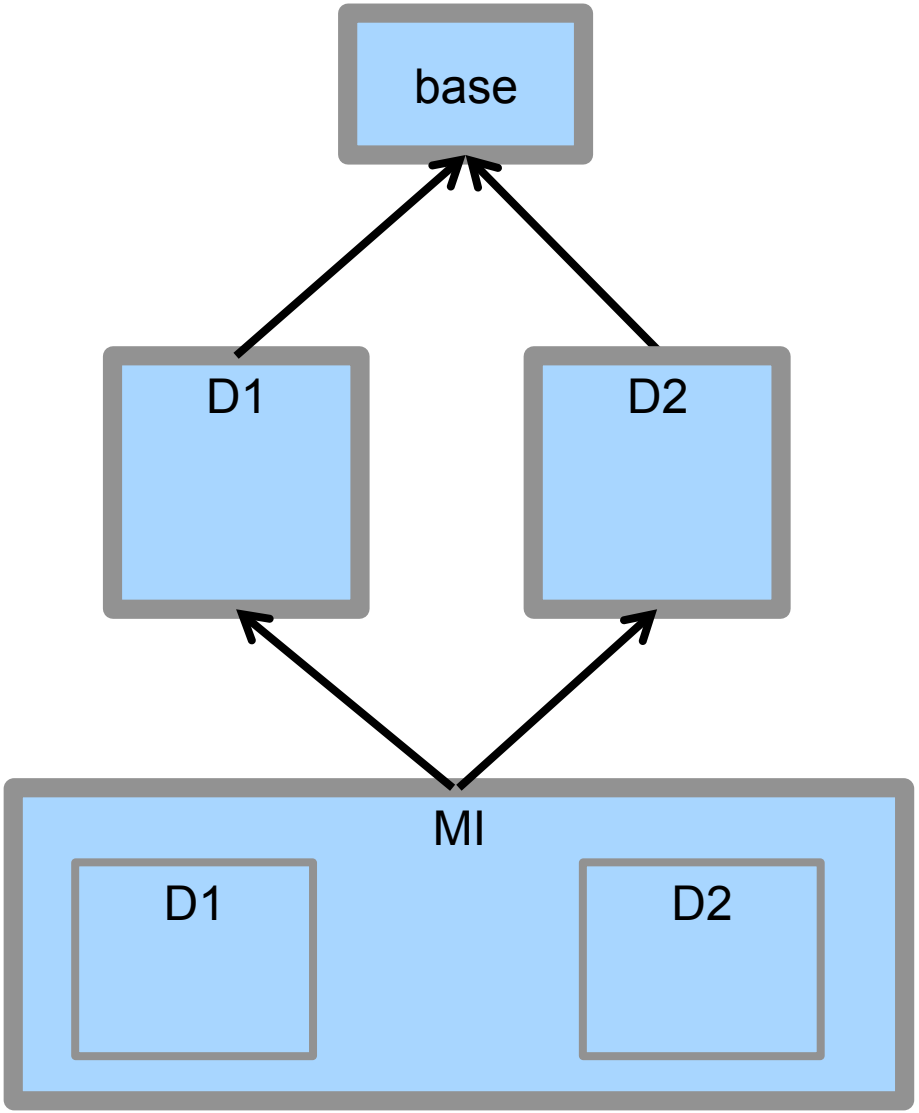
Solution :

➔ virtual à l'héritage

INSTITUT d'OPTIQUE GRADUATE SCHOOL

Virtual à l'héritage





INSTITUT d'OPTIQUE GRADUATE SCHOOL