Programmation C++

2ème année Informatique

J. Allali

Prog. C++

ENSEIRB-MATMECA

Plan

Introduction

Historique (wikipedia)

- 198x: Bjarne Stroustrup (AT&T Bell): “C with classes”:
  - Classes
  - fonctions virtuelles
  - surcharge d’opérateurs
  - héritage multiple
  - …

Normes:

- 1998: Normalisation du C++ par l’ISO (International Organization for Standardization) et ANSI ISO/CEI 14882:1998
- 2003: ISO/CEI 14882:2003
- 2011: ISO/CEI 14882:2011
- 2014: ISO/CEI 14882:2014
- 2017: ISO/IEC 14882:2017
- 2020: ISO/IEC 14882:2020

C (++)

Le C++ est un language impératif, *orienté objets*:

L'ajout de fonctionnalités permettant la mise en oeuvre de concepts objets dans le language C:

- l'objet: attributs(données internes) + méthodes (comportements), encapsulation.
- Typage des objets.
- Polymorphisme: un objet peut avoir plus d'un type.
- redéfinition.
- classe: description et génération des objets.

Il permet aussi la *programmation générique*: écriture de fonctions (et d'objets) indépendantes du **type** de ces arguments. C'est l'idée du "*code à trous*".

Incompatibilité en le C et le C++

Source C qui ne compile (invalidité syntaxique) pas en C++:

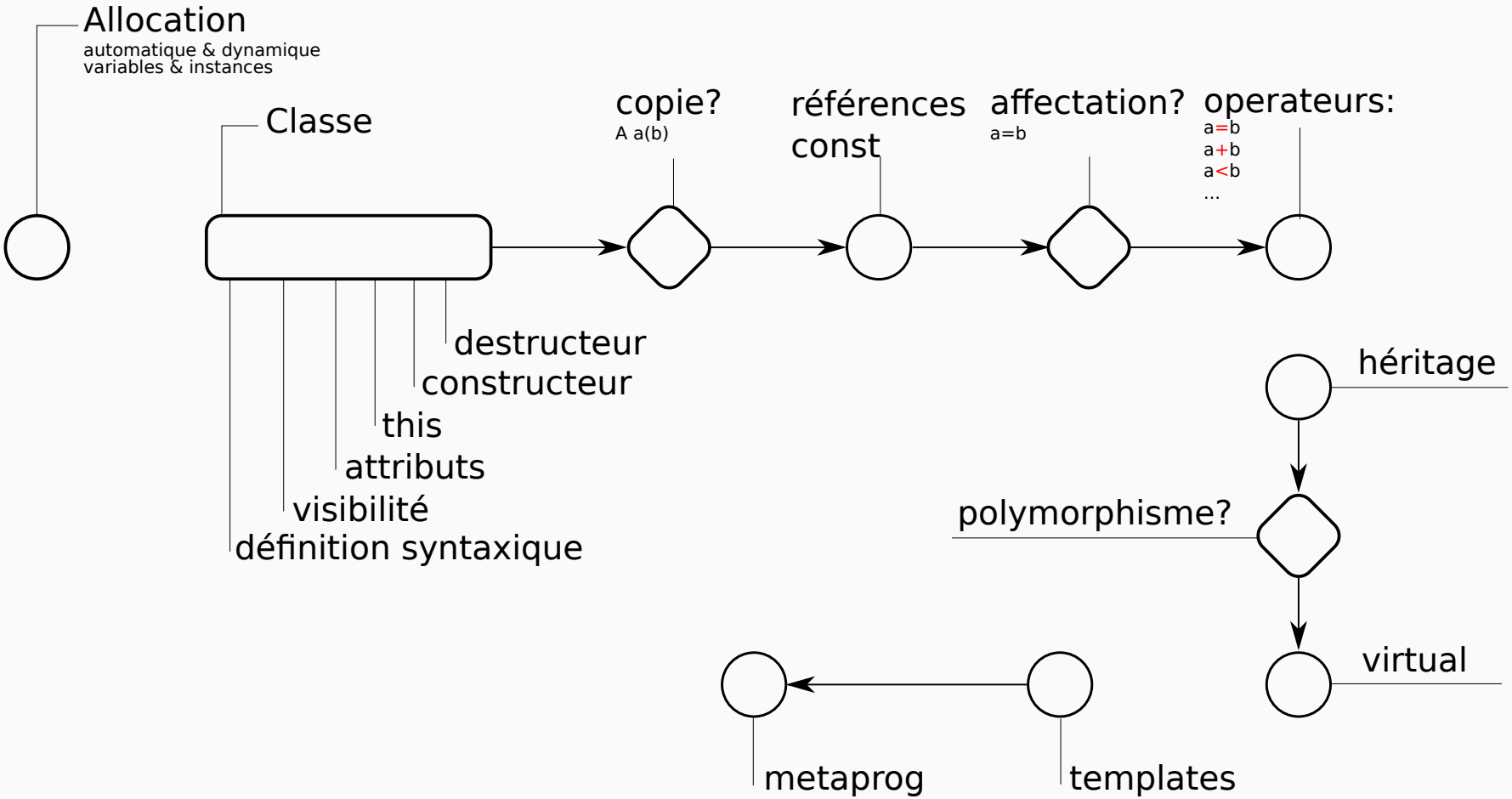
- Si le source contient un des nouveaux mots clés du C++

- Le C autorise la conversion implicite de void * en n'importe quel autre type de pointeur:

exemple

```
int *i=malloc(4);
```

Trame du cours



Plan

Allocation

Allocation automatique

L'allocation automatique se fait dans la pile.

Exemple

```
int i;
```

La variable i est allouée dans la pile automatiquement.

Ainsi, &i correspond à une adresse dans la pile à laquelle sizeof(int) octets sont réservés.

À la sortie du bloc dans lequel est déclarée i, il y a dépilement et donc l'adresse &i correspond à une zone mémoire qui n'est plus réservée.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

```
new int
new int [10]
```

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le deuxième cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple		
delete (**new** int);	*ou*	int *i=**new** int; **delete** i;
delete [] (**new** int [10]);	*ou*	int *p=**new** int[10]; **delete** [] p;

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée à cette adresse ne l'est plus.

Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int[10];
delete i;
delete [] t;
```

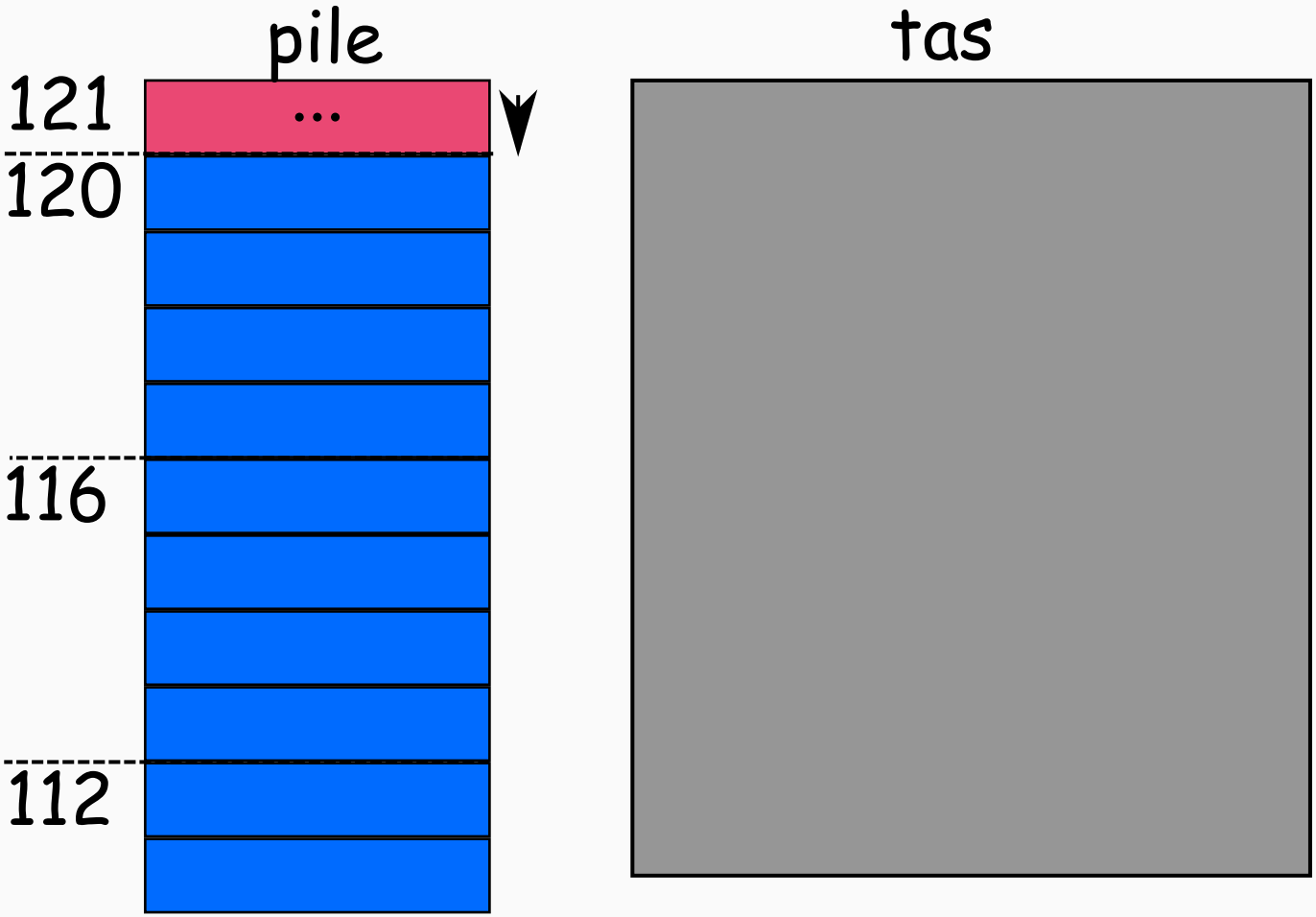
Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

- A) 1
- B) 2
- C) 3
- D) 4

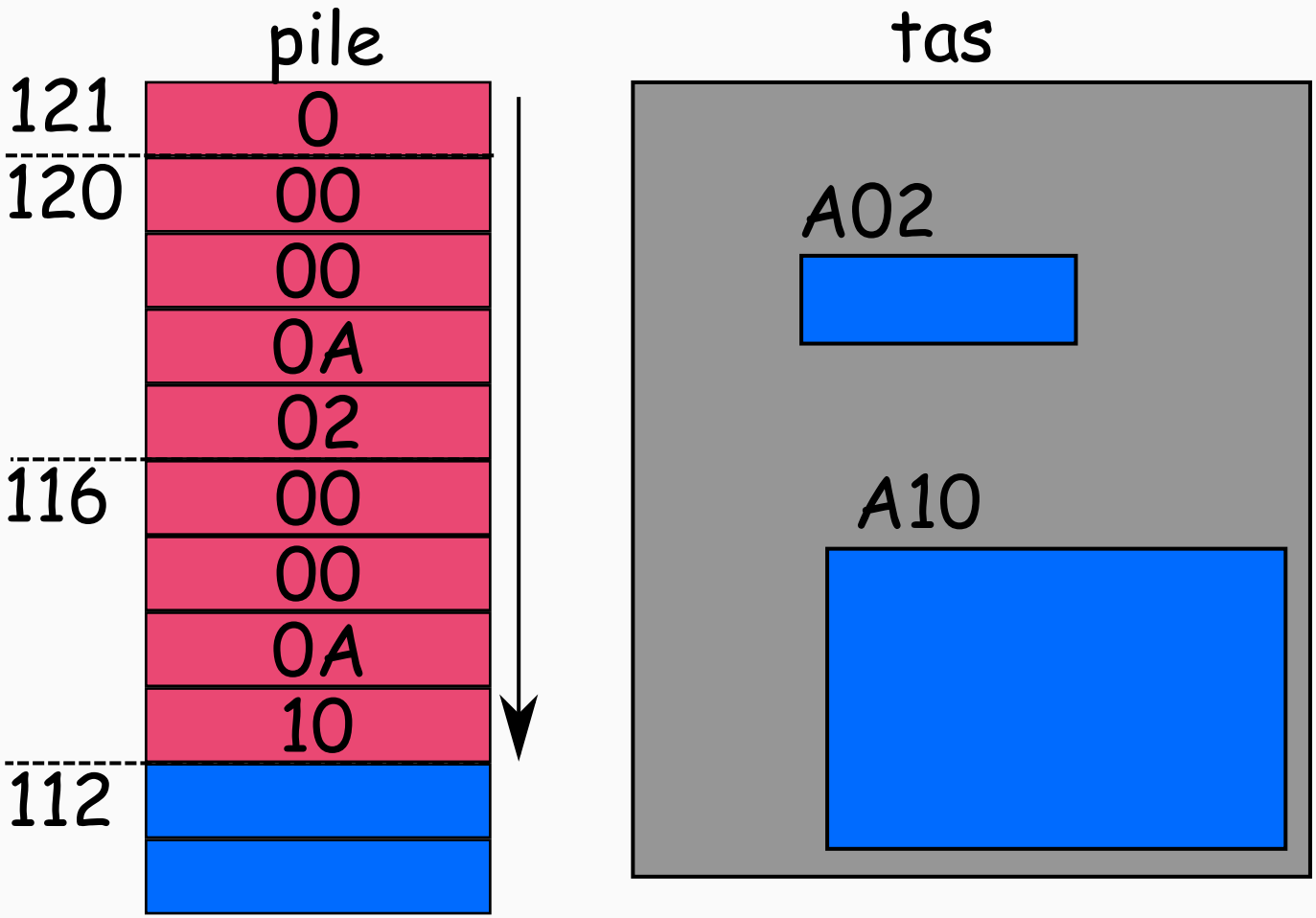
Réponse: 4, 2 automatiques et 2 dynamiques.

En effet, il y a deux allocations automatiques de sizeof(int *) pour les variables i et t

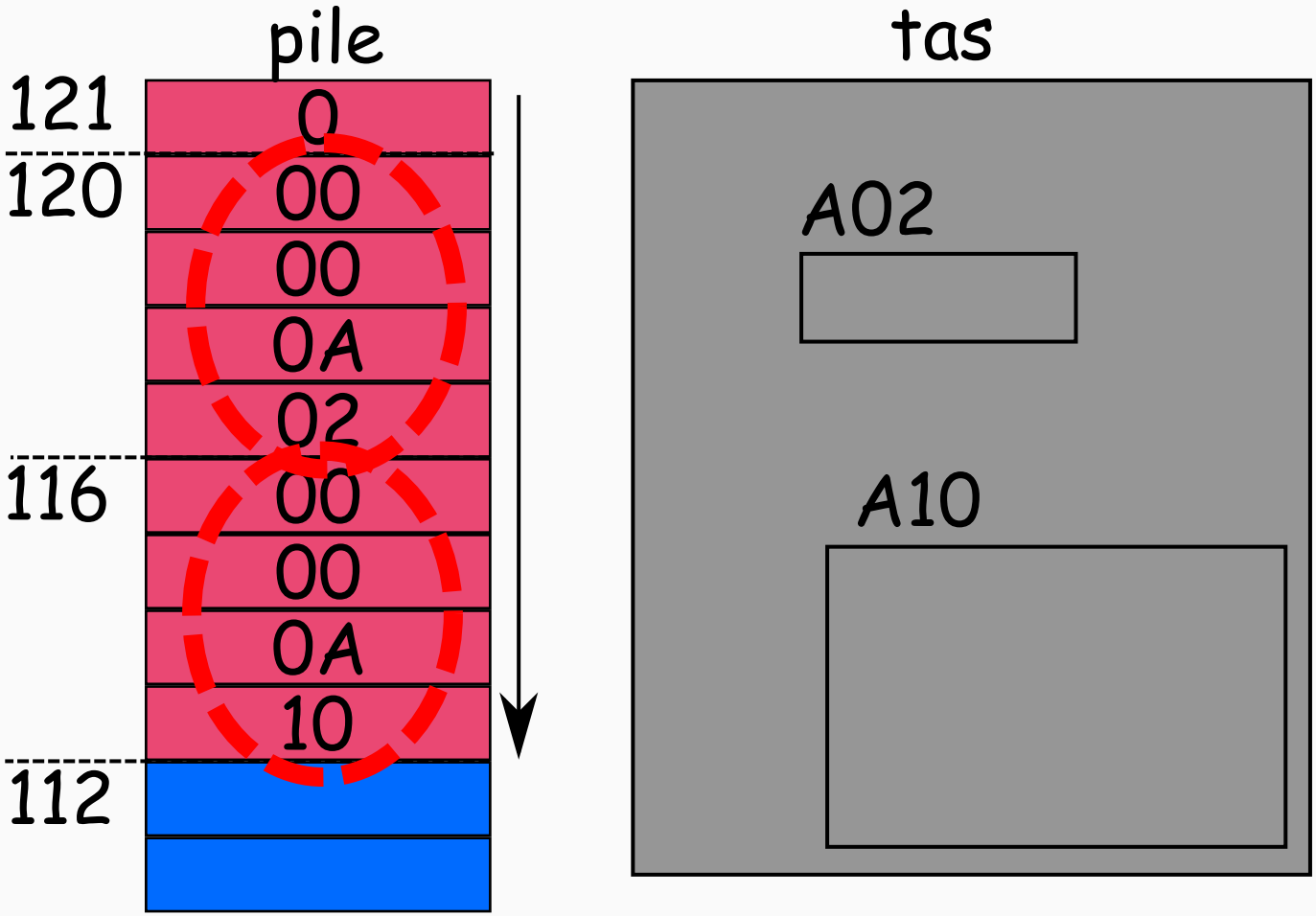
Allocation: exemple complet



Allocation: exemple complet



Allocation: exemple complet



Typage

Comme le C, le C++ dispose des types primitifs de base: **int**, **float**, **double**, **char**, **wchar_t**, **bool** et **void**. A chacun de ces types est associée une taille en octets.

À cela s'ajoute les pointeurs: type *. Une caractéristique importante est que, quelque que soit *type*, un pointeur fait toujours la même taille (4 octets en 32 bits et 8 octets en 64 bits).

Toute variable doit être typée. Le type indiquera le nombre d'octets réservés pour cette variable (dans la pile ou le segment de données) et l'interprétation associée à ces octets.

Le mot clé auto

La norme C++-11 introduit un nouveau sens au mot clé: **auto**.

auto déclenche une inférence de type statique (à la compilation) en fonction d'une affectation faite au moment de la déclaration d'une variable:

exemple

```
auto i=0; // i est de type int
auto d=10.0; // d est de type double
auto f=2.0f; // f est de type float
auto s="hello"; // s est de type const char *
```

Le mot clé auto

Cela fonctionne également avec le type de retour d'une fonction:

fonction

```
int f();
auto i=f(); // i est de type int
auto t = new int [10]; // t est de type int *
```

tableau

```
auto s="hello"; // s est de type const char *
auto c=s[0]; // c est de type char et non const char
```

On reviendra plus tard sur **auto**

initialisation de variable = et {}

Lors de la déclaration d'une variable, on utilise **=** pour initialiser celle-ci:

exemple

```
int i=0;
int j=12.2; // j vaut 12
char c='d'; // c vaut 100
char d=35.0; // d vaut 35
char e=35.8; // e vaut 35
float f=2.5; // f vaut 2.5
```

L'utilisation du **=** repose sur des mecanismes de conversion.

initialisation de variable = et {}

Il est possible d'utiliser {} pour l'initialisation:

exemple

```
int i{0};
int j{12.2}; // ERREUR
char c{'d'}; // c vaut 100
char d{35.0}; // ERREUR
float f{2.5}; // f vaut 2.5
double x{3.0};
float y{x}; // WARNING
```

Les {} implique un contrôle d'imbrication: un float peut être contenu dans un double mais pas l'inverse par exemple.

initialisation de variable = et {}

{} pour l'initialisation d'une variable est plus "sécurisée" que =

Nous verrons d'autres usages de {} par la suite.

Plan

Classes

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Les classes: déclaration

Une classe consiste en un regroupement de méthodes et d'attributs.

Exemple

```
class NomClasse {
attributs
...
methodes
...
};
```

L'ordre des déclarations ne compte pas.

La classe est une description des données internes et comportements qu'aura une instance générée par cette classe.

Les classes: instanciation

L'instanciation (ou réification), c'est à dire la création d'un objet instance (ressource) à partir de l'objet classe (description/générateur) peut se faire de façon dynamique:

allocation dynamique

```
A *x=new A();
A *y=new A{};
```

ou automatique:

allocation automatique

```
A z;
A t{};
```

Dans les deux cas, une ressource (adresse mémoire) est associée à l'instance. Dans l'allocation dynamique, cette ressource est située dans le tas, dans le cas automatique elle est située dans la pile.

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Visibilité: définition

On contrôle l'accès aux méthodes et attributs pour une sous-classe ou un objet exterieur avec `public,protected et private:`

	`la classe`	`sous-classe`	`exterieur`
public	`oui`	`oui`	`oui`
protected	`oui`	`oui`	`non`
private	`oui`	`non`	`non`

Remarque: Les membres protected et private ne peuvent être accédés qu'à partir d'une fonction membre de l'instance

Visibilité: déclaration

La visibilité par défaut dans une classe est **private**. On modifie la visibilité de la façon suivante

Source

```
class NomClasse {
attributs et méthodes privés
public:
attributs et méthodes publiques
protected:
attributs et méthodes protégés
...
};
```

On peut à tout moment changer la visibilité, celle-ci sera appliquée à toutes les déclaractions suivantes jusqu'au prochain changement de visibilité.

Les structures: déclaration

Les structures se déclarent comme en C:

Exemple

```
struct Nom {
...
attributs et méthodes publiques
...
};
```

En C++, les structures sont des classes dont la visibilité par défaut est **public**.

Par conséquent, elle peuvent contenir des attributs et des méthodes.

Deux noms de type sont associés à la structure: struct Nom et Nom

on voit que les structures C sont alors un cas particulier des structures C++

Les structures: instanciation

L'instanciation des structures se fait comme pour les classes.

allocation dynamique:

```
struct A { ... };
A *x = new A(); // erreur si fonction A
A *x = new struct A();
struct A *x=new struct A();
```

allocation automatique:

```
struct A { ... };
A x;
struct A x;
```

Friend: définition

Dans une classe A, le mot clé **friend** permet de donner à une fonction ou une autre classe les même droits qu'une méthode de A.

	la classe	friend	sous-classe	exterieur
public	oui	oui	oui	oui
protected	oui	oui	oui	non
private	oui	oui	non	non

une fonction f amie d'une classe A pourra accéder aux attributs privés de A ainsi qu'aux attributs protected d'une des classes parents de A

Friend: déclaration

La déclaration se fait dans la classe en indiquant soit le prototype de la fonction amie soit le nom de la classe amie:

Classe amie

```
class NomClasse {...
  friend class NomClasseAmie ;
};
```

Fonction amie

```
class NomClasse { ...
  friend void fonctionAmie(int,char,NomClasse) ;
};
```

la visibilité courante n'importe pas pour déclarer une classe ou fonction amie.

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Attributs et méthodes: définition

Une classe est un regroupement d'attributs, propriétés internes qu'aura une instance de la classe, et de méthodes, comportements qu'aura une instance.

Ces attributs sont déclarés à l'intérieur de la classe.

L'implémentation des méthodes peut se faire au moment de la déclaration ou à l'extérieur de la classe. On l'écrira dans la classe pour:

- les classes template
- les classes à usage locale au fichier
- l'inlining

Sinon on écrit un fichier entête "NomClasse.hpp" et un fichier source "NomClasse.cpp".

Exemple de classe: 1

Un seul fichier

```
class Test {
  int i; // attribut privé
public:
  void print(){ /* méthode publique */
    printf("Test\n");
  }
};

Test t;
t.print();
```

La compilation se fait avec la commande:

```
g++ -Wall Test.cpp -o test
```

Exemple de classe: 1

Un seul fichier

```
class Test {
  int i; // attribut privé
public:
  void print(){ /* méthode publique */
    printf("Test\n");
  }
};

Test *t = new Test;
t->print();
```

La compilation se fait avec la commande:

```
g++ -Wall Test.cpp -o test
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {
  int i; //attribut privé
public:
/* méthode publique */
  void print();
};
```

Test.cpp:

```
#include"Test.hpp"
void Test::print(){
  printf("Test\n");
}
```

Exemple.cpp:

```
Test t;
t.print();
```

La compilation se fait en deux temps:

```
g++ -Wall -c Test.cpp -o Test.o
g++ -Wall -c Exemple.cpp -o Exemple.o
g++ Test.o Exemple.o -o Exemple
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {
  int i; //attribut privé
public:
/* méthode publique */
  void print();
};
```

Test.cpp:

```
#include"Test.hpp"
void Test::print(){
  printf("Test\n");
}
```

Exemple.cpp:

```
Test *t = new Test;
t->print();
```

La compilation se fait en deux temps:

```
g++ -Wall -c Test.cpp -o Test.o
g++ -Wall -c Exemple.cpp -o Exemple.o
g++ Test.o Exemple.o -o Exemple
```

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Tout membre de classe a pour nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Cas automatique

```
ClasseA x;
x.i = 1 ; //i est un attribut public de ClasseA
x.ClasseA::i = 1 ; // idem
x.methode(); // appel de méthode
x.ClasseA::methode(); // idem
```

Dans ce cas l'instance se trouve dans la pile.

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Tout membre de classe à pour nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Cas dynamique

```
ClasseA *y = new ClasseA();
y->i = 1 ; //i est un attribut public de ClasseA
y->ClasseA::i = 1 ; // idem
y->methode(); // appel de méthode
y->ClasseA::methode(); // idem
```

Dans ce cas l'instance est allouée dynamiquement dans le tas.

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Le mot clé this

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source (site d'appel).

Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de* **const**.)

Exemple

```
class A{
  int attributPrive;
public:
  void setAttribut(int valeur){
    this->attributPrive = valeur;
  }
};
```

Le mot clé this

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source (site d'appel).

Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de* **const**.)

Exemple

```
class A{
  int attributPrive;
public:
  void setAttribut(int valeur) {
    attributPrive = valeur ;
  }
};
```

Lorsqu'il n'y a pas de variable locale de même nom que l'attribut, on peut omettre **this**.

Opérateur de portée

L'opérateur de portée :: peut-être utilisé pour indiquer précisement la variable que l'on souhaite manipuler.

Exemple:

```
int lunatique; // variable globale, beurk!
class A {
 char lunatique; // attribut privé de la classe
public:
 void printAllVariable(double lunatique){// variable locale
                                          // à la méthode
  lunatique;//fait référence à la variable locale
  A::lunatique;//fait référence à l'attribut d'instance
  this->A::lunatique;//fait référence à l'attribut d'instance
  this->lunatique;//fait référence à l'attribut d'instance
  ::lunatique;//fait référence à la variable globale
 }
};
```

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisés pour structurer le code et pour éviter les problèmes de collisions. Pour déclarer un espace de nom on utilise le mot clé **namespace**:

Espace de nom:

```
namespace Nom { // début de l'espace de nom
  class A {
    ...
  };
  int i;
  int fonction();
}// fin de l'espace de nom
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

A.hpp

```
namespace tec {
    class A {
      public:
      void m();
    };
}
```

A.cpp

```
#include "A.hpp"
namespace tec {
    void A::m(){
    ...
    }
}
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de portée.

A.hpp

```
namespace tec {
    class A {
      public:
        void m();
    };
}
```

A.cpp

```
#include "A.hpp"
void tec::A::m(){
...
}
```

Il n'y pas pas de limite sur le nombre d'espaces de nom différents pouvant être définis dans un fichier ni sur le niveau d'imbrication de ces espaces. Cependant la taille des noms de fonctions est limitée et les espaces de nom font partis du nom de la fonction. D'une certaine manière, une classe peut-être vue comme un espace de nom particulier.

Espace de noms: using

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
Navette n; //Erreur!
void starWars(){
    Navette n; //Erreur!
}
```

Espace de noms: using

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        spatial::Navette n; //ok
}
spatial::Navette n; //ok
void starWars(){
    spatial::Navette n; //ok!
}
```

Utilisation de l'opérateur de portée

Espace de noms: using

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
using spatial::Navette;
namespace missions {
        Navette n; //ok
}
Navette n; //ok
void starWars(){
    Navette n; //ok!
}
```

Espace de noms: using

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
using spatial::Navette;
Navette n; //ok
void starWars(){
    Navette n; //ok!
}
```

Espace de noms: using

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        using spatial::Navette;
        Navette n; //ok
}
Navette n; //Erreur!
void starWars(){
    Navette n; //Erreur!
}
```

Espace de noms: using

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
Navette n; //Erreur!
void starWars(){
    using spatial::Navette;
    Navette n; //ok
}
```

Espace de noms: using

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
Navette n; //Erreur!
void starWars(){
    using namespace spatial;
    Navette n; //ok
}
```

Using

Le **using** crée des synonymes locaux entre les espaces de nom:

Fichier exemple.hpp:

```
namespace math{
  const double pi=3.14;
}

namespace cercle{
  using math::pi;
  double surf(double);
}
```

Fichier exemple.cpp:

```
#include''exemple.hpp''

double cercle::surf(double r)
{
  return pi*r*r;
}
```

- Permet de remplacer facilement une référence externe par une autre.
- Permet aussi l'introduction de modularité sans modification profonde du code.

"using namespace nom;" permet de créer des synonymes pour tout les éléments de nom

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Constructeur: définition

Un constructeur est une méthode de classe qui est appelée après la réservation des ressources nécessaires à la création d'une instance de la classe et qui a pour objectif d'initialiser les attributs de cette instance.

En C++, le constructeur est une méthode sans valeur de retour de même nom que la classe.

On appelle constructeur par défaut le constructeur ne prenant aucun argument.

Dans le cas où la classe ne comporte aucun constructeur, le compilateur ajoute un constructeur par défaut. Celui-ci n'est plus présent à partir du moment où l'on a écrit au moins un constructeur.

Constructeur: exemple 1 fichier

Exemple.cpp

```
class Exemple  {
  double pop;
  public:
  Exemple() { // Constructeur par défaut
    pop=0;
  }
  Exemple(double d){ // Constructeur
    pop=d;
  }
};
```

Constructeur: exemple 2 fichiers

Exemple.hpp

```
class Exemple{
  double pop;
  public:
  Exemple();
  Exemple(double );
};
```

Exemple.cpp

```
#include "Exemple.hpp"

Exemple::Exemple(){
  pop=0;
}
Exemple::Exemple(double d){
  pop=d;
}
```

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{
        int pop;
      public:
        A(int value):pop(value){
        }
};
```

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{
        int pop;
      public:
        A(int pop):pop(pop){
        }
};
```

Il n'y a pas d'ambiguïté sur les noms de variables.

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{
        int pop;
      public:
        A(int pop):pop(pop*2){
        }
};
```

On peut effectuer des opérations

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
intdecremente(int i){
    return i-1;
}
class A{
          int pop;
        public:
          A(int pop):pop(decremente(pop)*2){
          }
};
```

On peut appeler des fonctions

Constructeur: initialisation des attributs

Les attributs de l'instance peuvent être initialisés de la façon suivante:

Initialisation:

```
class A{
        int pop;
      int m(){
        return 1;
      }
      public:
        A(int pop):pop((m()+pop)*2){
        }
};
```

On peut appeler des methodes!

Constructeur: initialisation des attributs C++11

La norme C++11 autorise le placement de valeur par défaut pour les attributs non statiques à la déclaration:

```
class A{
  int v=10; // ou v{10};
  std::string s{"hello"};
  public:
  A(){
    printf("%d␣%s\n",v,s.c_str());
    // affiche: 10 hello
  }
  };
```

Constructeur: initialisation des attributs

On peut initialiser plusieurs attributs de cette façon. Cependant l'ordre des initialisations se fait toujours selon celui des déclarations des attributs dans la classe:

Source:

```
class A {
  int a;
  double d;
  char c;

public:
  A();
};

A::A():c('a'),a(0){} //Warning
```

Constructeur: initialisation des attributs

On peut initialiser plusieurs attributs de cette façon. Cependant l'ordre des initialisations se fait toujours selon celui des déclarations des attributs dans la classe:

Source:

```
class A {
  int a;
  double d;
  char c;

public:
  A();
};

A::A():a(0),c('a'){} //ok
```

Constructeur: et instanciation

Dans le cas de l'allocation automatique, il existe deux façons d'indiquer le constructeur à appeler:

Parenthésée

```
A x;
A x(1);
A x('c');
A x(); // Erreur:
       // pas d'instanciation!
```

Affectation

```
A y;
A y=1;
A y='c';
```

L'utilisation du `'='` au moment de l'instanciation fait nécessairement référence à un constructeur.

Constructeur: et instanciation

Pour le cas dynamique, seule la version parenthésée est légale.

Exemple

```
A *p=new A; // Construction par defaut
A *p=new A(3); // Utilisation du constructeur
          // prenant un entier
A *p=new A(); //Construction par defaut
A *p=new A=5; // Erreur !
```

Constructeur: et instanciation

Cette syntaxe est valable pour les types primitifs:

Affectation

```
int i=0;
char c='a';
char *p=NULL;
```

Parenthésée

```
int i(0);
char c('a');
char *p(NULL);
```

Constructeur: et instanciation

Il est possible d'instancier des classes de façon anonyme (aucun nom de variable n'est associé à l'instance):

Source:

```
A(); // Instanciation utilisant le
     // constructeur par défaut
A(1); // Avec le constructeur A(int );
A; // Erreur ! syntaxe invalide.
```

Ce type d'instanciation peut-être utilisé lors du passage d'un argument à une fonction ou lors d'une levée d'exception.

Constructeur: et instanciation

Source:

```
void f(A a){...}


...
f(A(1)); //1. Appel de fonction
f(1); //2. Utilisation de l'instanciation implicite
...
throw A(); //3. Levée d'exception
```

L'appel au constructeur dans le cas 2 se fait de façon implicite. Le compilateur cherche une fonction `f(int)`, il liste l'ensemble des fonctions disponibles: `f(A)` et cherche une façon de construire A à partir d'un entier.

Constructeur: et tableaux

Lors de l'allocation de tableau, les instances du tableaux doivent être créées à l'aide du constructeur par défaut.

Source:

```
A *t=new A[10]; // dynamique

A tableau2[10]; //automatique
```

⇒ Une classe ne comportant pas de constructeur par défaut ne peut pas être utilisée avec les tableaux.

⇒ On utilisera un tableau de pointeurs, chaque élément du tableau sera instancié séparément avec le bon constructeur.

Constructeur: et tableaux

Tableaux:

```
class A{
public:
   A(int);
};
...
A t[10]; // Erreur !
A *t[10]; // Tableau de pointeurs
for(int i=0;i<10;i++)
   t[i]=new A(3); // utilisation du constructeur
                  // allocation dynamique
...
for(int i=0;i<10;i++)
   delete t[i];
```

Constructeur: conversion

Comme nous l'avons vu, l'appel au constructeur peut se faire de façon implicite.

Exemple

```
int f(A p);
...
f(1);
```

Le conversion se fait sur au plus un niveau

Constructeur: conversion

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

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

void f(B q);

f(1);//Ne marche pas!
f(A(1)); // ok
```

La notation `A a=1;` fait appel au système de conversion.

Constructeur: explicit

Un constructeur déclaré **explicit** ne sera pas utilisé pour effectuer une conversion:

```
class A{
public:
    explicit A(int );
};
A::A(int i){} // explicit n'est pas reporte ici
...
void f(A p);
...
f(1);//Erreur: ne trouve pas la fonction f(int)
A a=1; // Erreur!
A b(1); // ok
```

Constructeur: par recopie

Le constructeur par recopie est le constructeur qui permet d'instancier une classe à partir d'une autre instance de cette classe.

Ce constructeur est utilisé entre autre lors de la transmission de paramètres à une fonction et lors d'une valeur de retour de fonction.

```
A f(A p) { return p;}
....
A x;
f(x)
```

- La variable `p`, locale à la fonction `f`, est allouée automatiquement et instanciée avec le constructeur par recopie à partir de l'instance `x`.
- La valeur de retour de `f` est instanciée par recopie de la valeur locale.

Constructeur: par recopie

L'idée est donc d'avoir un constructeur dans la classe A qui prend une instance de type A en argument:

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

Le problème est que pour instancier la variable locale x au constructeur il nous faut le constructeur par recopie!

La solution repose sur l'utilisation des références:

```
class A{
public:
   A(const A&x);
   A(A &x);
};
```

Références

Une référence peut être vue comme un “alias” sur une instance.

Une référence doit obligatoirement être initialisée au moment de sa création et ne peut référencer une autre instance par la suite.

```
int i;
int &r=i;
```

Après l’initialisation:

- les variables i et r représentent la même donnée
- l’adresse de r (&r) est égale à l’adresse de i (&i).

$\Rightarrow$ contrairement à un pointeur, une référence “pointe” toujours sur une instance.

Références

Exemple:

```
void swap(int &i,int &j){
  int k=i;
  i=j;
  j=k;
}
```

```
int a=3,b=5;
swap(a,b);
int t[5];
swap(t[1],t[2]);
```

On peut aussi utiliser les références comme valeur de retour:

```
int &element(int *t,int i){
  return t[i];
}

element(t,5)=10; // motifie t[5]
```

Règle: Sauf pour les types primitifs, on ne prendra plus les instances par recopie mais par référence.

Références

Parmis les avantages des références, on pourra noter qu'une référence ne peut pas être nulle:

```
void swap(int *i, int *j){
  assert(i!=NULL);
  assert(j!=NULL);
  int k=*i;
  *i=*j;
  *j=k;
}
```

```
void swap(int &i,int &j){
  int k=i;
  i=j;
  j=k;
}
```

Sauf si l'on souhaite laisser la possibilité de passer le pointeur `NULL` ou bien de pouvoir modifier la valeur du pointeur, on remplace le pointeur par une référence.

Références

Pour les valeurs de retour, on ne peut retourner une référence que si l'objet renvoyé existe en dehors de la fonction (comme pour les pointeurs):

```
int &f(){
  int i;
  return i;
}

int &g(int j){
  return j;
}

int &h(int &k){
  return k;
}
```

```
int &p(int *l){
  return *l;
}

f()=0;
int x;
g(x)=0;
h(x)=0;
p(&x)=0;
p(NULL)=0;
```

Références

```
void print(int &i){
  // 1 entier
}

void print(Matrix &m){
  // 1 instance
}
```

```
void print(int i){
  // 2 entiers
}

void print(Matrix m){
  // 2 instances
}
```

```
int j;
print(j);
Matrix n(10000);
print(n);
```

Problème de l'intégrité des données : comment être sûre que la fonction ne va pas modifier l'instance passée en argument ?

const: définition

Le mot clé **const** sert à indiquer qu'une donnée ne peut être modifiée:

```
const int i=5;
int j=10;
const int &r=j;
r=4; // erreur!
j=4; // ok
i=1; // erreur!
```

```
const int *p=0; // int const * p=0;
p=new int[10];
p[0]=5; // erreur!
int *const q=new int[10];
q=NULL; // erreur!
q[0]=5; // ok
```

Un variable déclarée **const** doit être initialisée lors de sa déclaration.

déclaration:	const type	const *	const *	const p
lecture seule:	**p=	**p=	*p=	p=

Lorsque l'on dispose d'une référence constante sur une instance de classe, comment garantir que l'appel à une méthode de cette instance de va pas modifier l'instance ?

const: et instances

Illustration du problème:

```
class A{
  int i;
public:
  int get(){return i;}
  void set(int v){
    this->i=v;
  }
};
```

```
void f(const A &x){
  x.get();
  x.set(0); // !
}

A y;
f(y);
```

La référence est constante (l'instance ne doit pas être modifiée) mais l'appel de méthode modifie l'objet !

$\Rightarrow$ On distingue deux types de méthodes: celles qui sont susceptibles de modifier l'objet et celles qui ne modifient pas l'objet.

const: méthodes

Une méthode de classe peut être typée **const**. Cela indique que dans cette méthode le type de **this** est const NomClasse const *

L'ensemble des attributs de la classe deviennent alors constants eux aussi:

```
class A{
  int i;
  int *d;
public:
  void m() const;
  void p() ;
};
```

```
void A::p() {
  //  this: A *const
}

void A::m() const{
  //  this: const A * const
  //  this: A const * const
  //  this->i: const int
  //  this->d: int   * const
}
```

const

Lorsque l'on dispose d'une variable constante sur une instance, seules les méthodes indiquées comme constantes peuvent être appelées sur cette instance:

```
A x;
const A & y=x;

x.m(); // ok
x.p(); // ok
y.p(); // erreur!
y.m(); //ok
```

Une méthode **const** peut appelée sur tout type de variable tandis qu'une méthode non const ne peut être appelée que sur une variable non constante.

Règle: lors de l'écriture d'une méthode, on ne doit pas se demander si la méthode doit être const mais plutôt si on a besoin qu'elle ne soit pas const.

Règle: Les arguments de fonction (non primitifs) seront des références constantes sauf si l'on veut modifier l'argument.

const: surcharge

Il est possible de surcharger une méthode en utilisant le **const**:

```
class A{
  public:
  void m(){
    printf("le␣site␣d'appel
␣␣␣␣n'est␣pas␣constant");
  }
  void m() const{
    printf("le␣site␣d'appel
␣␣␣␣est␣constant")
  }
};
```

```
A x;
const A &y=x;
A const *p=&x;
x.m();  //?
y.m();  //?
p->m(); //?
```

Le **const** fait partie de la signature de la méthode, il doit être présent dans le `.cpp` et le `.hpp`.

const: argument de fonction

Exemple d'utilisation du **const** pour les arguments d'une fonction.

```
class A{
  int i;
public:
  int get() const {return i; }
};

void f(A a){ // copie
  printf("%d\n",a.get());
}

void g(A &b){
  printf("%d\n",b.get());
}
```

```
void h(const A &c){
  g(c); //!
  f(c);
  c.get();
}

A x;
const A &y=x;
h(x);
g(x);
g(y);
```

Constructeur: recopie (suite)

Ainsi, le constructeur par recopie peut s'écrire de deux façons:

```
class A{
public:
   A(const A &x); // 1
   A(A &x);   // 2
};
```

Si l'on n'écrit pas de constructeur par recopie, le compilateur en fournit un. Ce constructeur fera un recopie attribut par attribut en utilisant les constructeurs par recopie des types des attributs.

Le constructeur fournit sera de type 1, si tous les constructeurs par recopie des attributs sont de type 1, sinon il sera de type 2.

Constructeur: recopie par le compilateur

```
class Comp{
  int i;
  std::string s;
  Matrix m;
  public:
  Comp(const Comp&c):i(c.i),s(c.s),m(c.m){}
};
```

Constructeur: par recopie

On préférera toujours écrire le constructeur par recopie sous la forme 1.

Règle: Toute classe nécessitant une copie profonde (allocation dynamique des attributs) devra avoir un constructeur par recopie.

Constructeur: par recopie

```
class A{
  int *i;
  public:
  A():i(new int(0)){}
  ~A(){ delete i; }
};
```

```
A x;
A y(x);
```

Ici, il y a une erreur car on libère deux fois le même espace mémoire.

Constructeur: par recopie

```
class A{
  int *i;
  public:
  A():i(new int(0)){}
  ~A(){ delete i; }
  A(const A &a):i(new int(*(a.i))){}
};
```

```
A x;
A y(x);
```

Constructeur: protected et private

Une classe n'ayant que des constructeurs protected ne pourra être instanciée que par

- Une sous-classe
- Une classe ou fonction amie
- Une classe ou fonction amie d'une sous-classe
- Une méthode statique

Une classe n'ayant que des constructeurs private ne pourra être instanciée que par

- Une classe ou fonction amie
- Une méthode statique

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Destructeur: définition

Le destructeur est une méthode d'instance n'ayant pas de type de retour et ayant pour nom: `~NomClasse`

```
class Nom{
public:
    ~Nom(){}
};
```

```
void f(){
  Nom n;
  Nom *p=new Nom;
  delete p; // Appel de p->~Nom()
} // Appel de n.~Nom()
```

Règle: toute classe ayant allouée dynamiquement ces attributs devra écrire un destructeur.

Si l'instance a été allouée dynamiquement alors l'appel au destructeur est déclenché par l'opérateur delete.

Si l'instance a été allouée automatiquement alors l'appel au destructeur se fait à la sortie du bloque d'instructions qui contient cette instance.

Les ressources mémoires associées à l'instance sont libérées après l'appel au destructeur.

Destructeur: exemple

```
class Tab{
  int *data;
public:
  Tab(int s){
    data= new int[s];
  }
  ~Tab(){
    delete [] data;
  }
};
```

```
int main(){
Tab t(10); //allocation
} //liberation
```

Il est ainsi possible de "simuler" un tableau dans la pile avec une désallocation automatique.

Destructeur: auto

```
class AutoInt{
  int *t;
public:
  AutoInt(int *t):t(t);
  int get (int i) const {
    return t[i];
  }
  void set(int i,int v){
    t[i]=v;
  }
  ~AutoInt(){
    if (t!=NULL)
      delete [] t;
  }
};
```

```
//Utilisation:
int main(){
  AutoInt t(new int[100]);
  t.get(0);
  t.set(0,0);
} // libération auto.
```

Nous verrons comment améliorer ce code avec les opérateurs, cependant attention à la recopie!

Destructeur: auto

```
class AutoInt{
  int *t;
public:
  AutoInt(int *t):t(t);
  int get(int i) const;
  void set(int i,int v);
  ~AutoInt(){
    if (t!=NULL)
      delete [] t;
  }
  AutoInt(AutoInt &p){
    t=p.t;
    p.t=NULL;
  }
};
```

```
//Utilisation:

void f(AutoInt ai){
}// libération auto.

AutoInt tab(){
  AutoInt x(new int[100]);
  return x;
}
int main(){
  AutoInt q(new int[100]);
  f(q);

  AutoInt r(tab());
} // libération auto.
```

Une solution consiste à transferer le pointeur...

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Static: définition

Une variable statique est une variable dont la durée de vie est égale à celle du programme et dont la portée peut être réduite à une fichier, une classe, une fonction ou méthode selon l'endroit où elle est déclarée.

Fichier.hpp

```
void f();

class A{
  static int i;
  public:
  static void m();
};
```

Fichier.cpp

```
#include "Fichier.hpp"

static double v;

int A::i=0;

void A::m(){
  this; //erreur!
}
void f(){
  static int compteur=0;
}
```

Static: et classes

Dans le contexte des classes, une méthode publique statique est équivalente à une fonction amie (sauf pour la portée).

On pourra utiliser les méthodes statiques en jonction avec un constructeur privé.

On peut utiliser le mot clé **static** en jonction avec **const** afin de définir des propriétés constantes de classes:

```
class Chaine {
  public:
  static const char END='\0';
};
```

Les variables **static** et **const** sont les seules à pouvoir être initialisées lors de la déclaration dans le fichier `.hpp`.

Static: et classes

Dans le contexte des classes, une méthode publique statique est équivalente à une fonction amie (sauf pour la portée).

On pourra utiliser les méthodes statiques en jonction avec un constructeur privé.

On peut utiliser le mot clé **static** en jonction avec **const** afin de définir des propriétés constantes de classes:

```
class Chaine {
  public:
  static const char END;
};
const char Chaine::END='\0';
```

L'initialisation peut se faire à la déclaration dans ce cas. Chaine::END n'a pas d'existence réelle durant l'execution (équivalent à une macro).

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

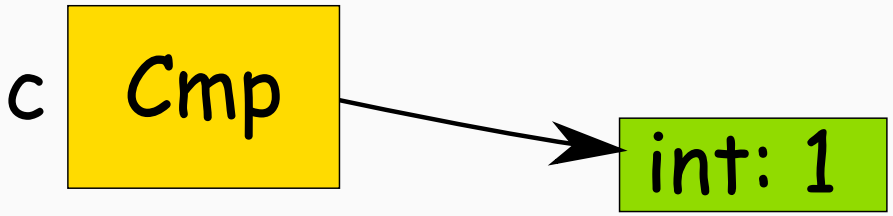
Une classe compteur

L'objectif est d'écrire une classe `Compteur` telle que plusieurs instances de cette classe puissent partager un même compteur qui sera ici implanté par un entier.

Ceci implique que le compteur soit alloué dynamiquement. En effet, on souhaite qu'il puisse être transmis entre des instances.

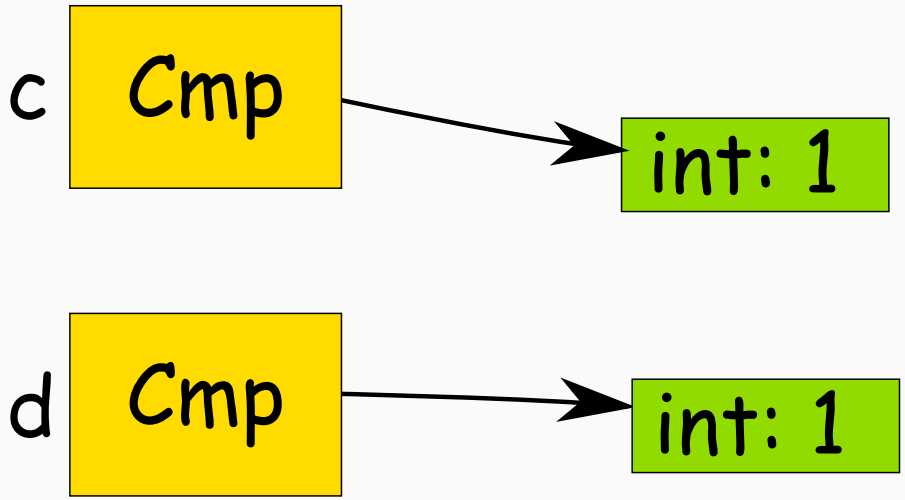
Compteur: Exemple1

Cmp c;



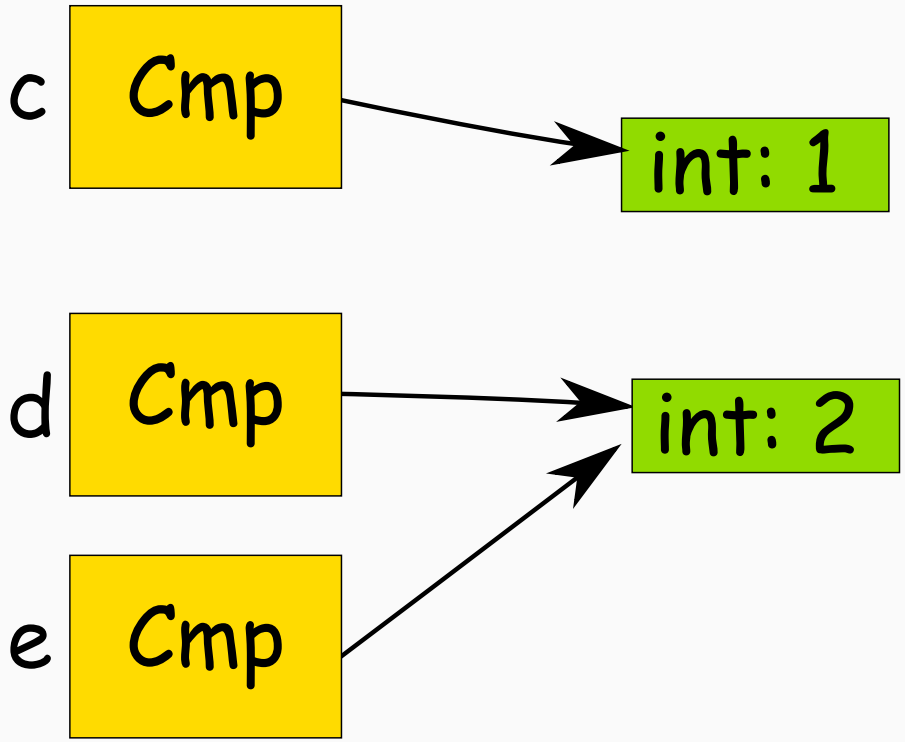
Compteur: Exemple1

```
Cmp c;
Cmp d;
```



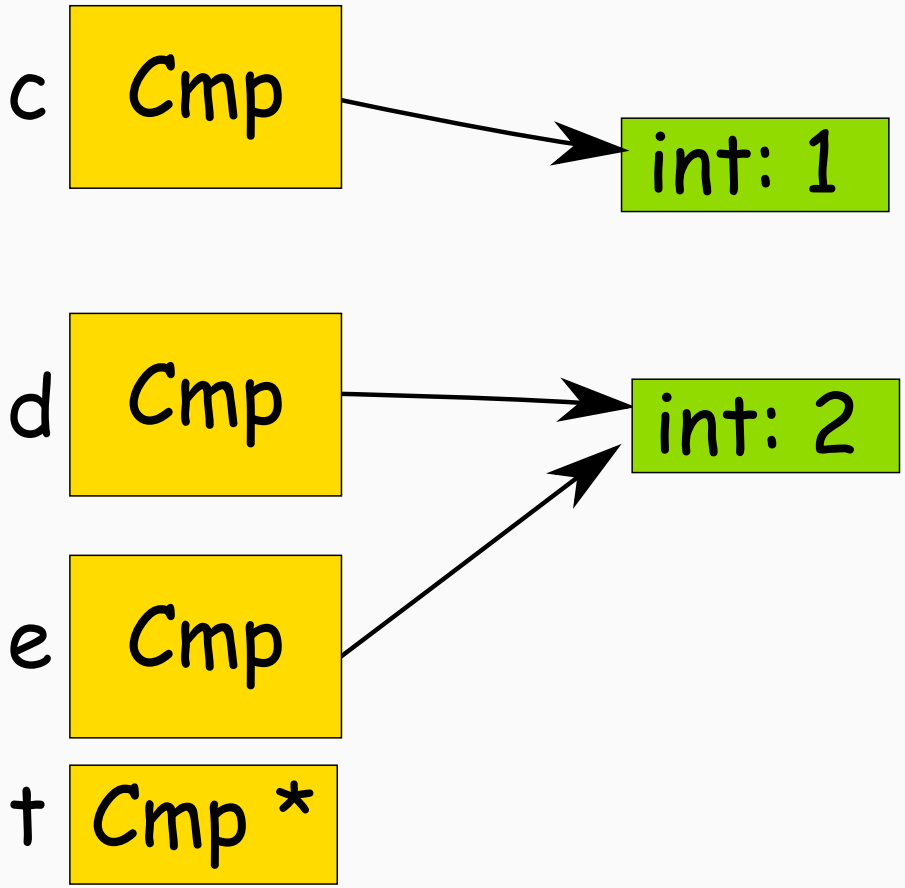
Compteur: Exemple1

```
Cmp c;
Cmp d;
Cmp e=d;
```



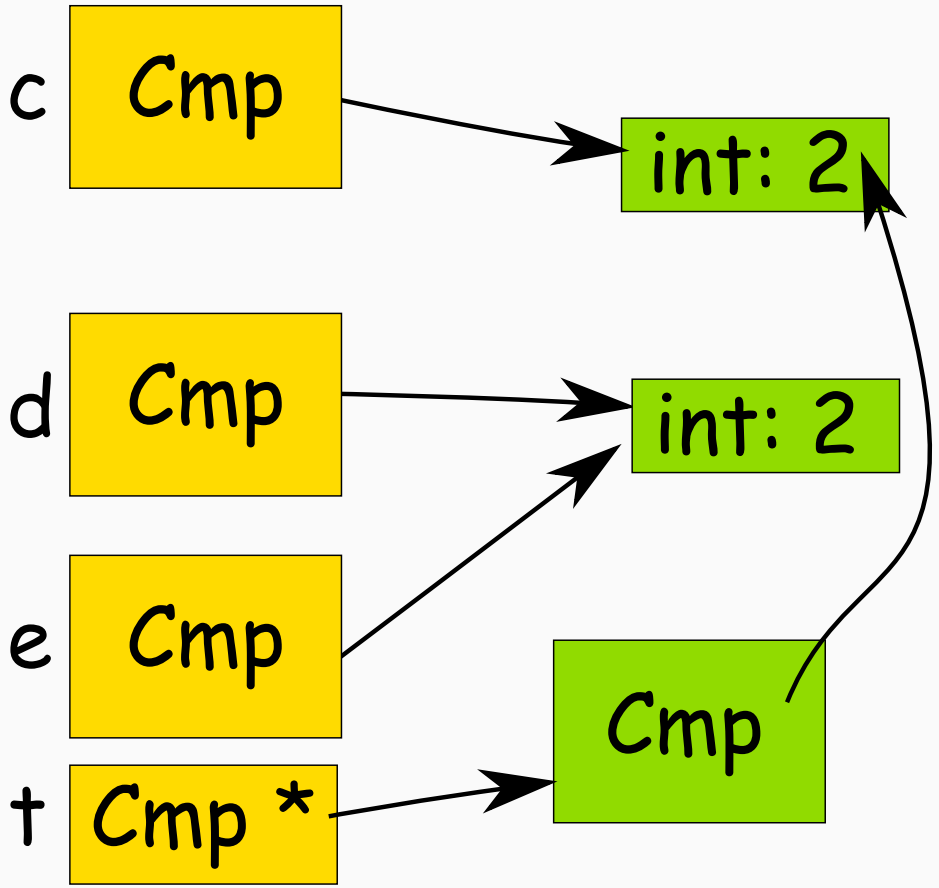
Compteur: Exemple1

```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
```



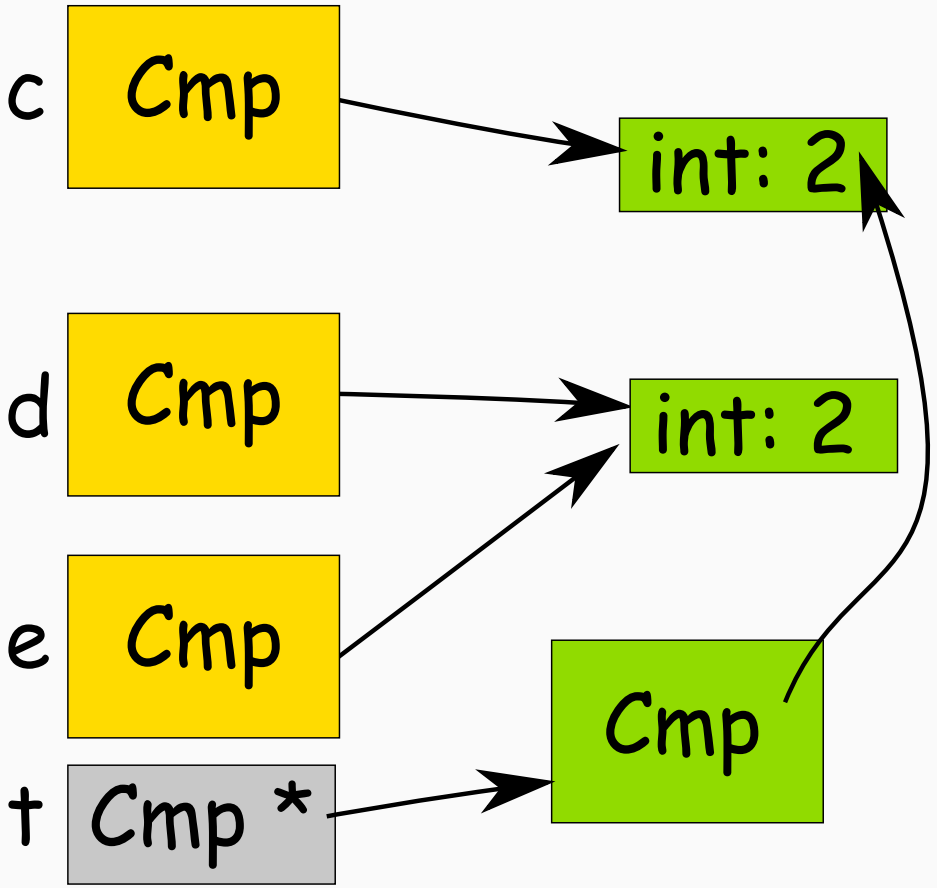
Compteur: Exemple1

```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
t=new Cmp(c);
```



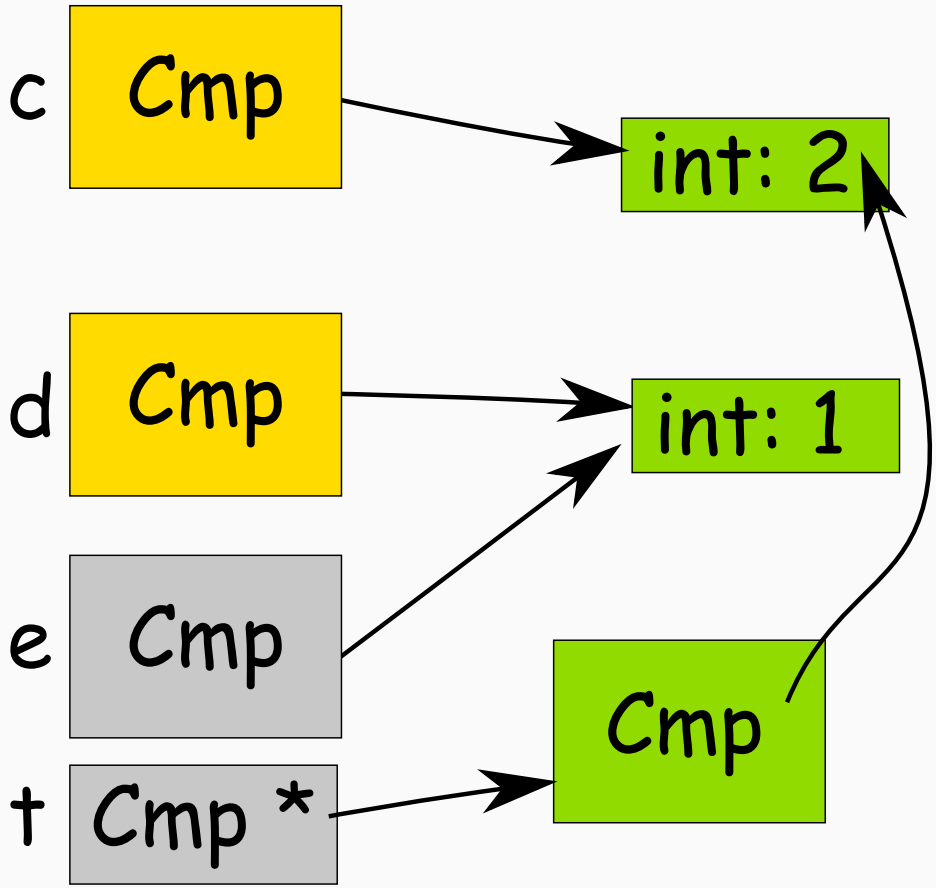
Compteur: Exemple1

```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
t=new Cmp(c);
...
}
```



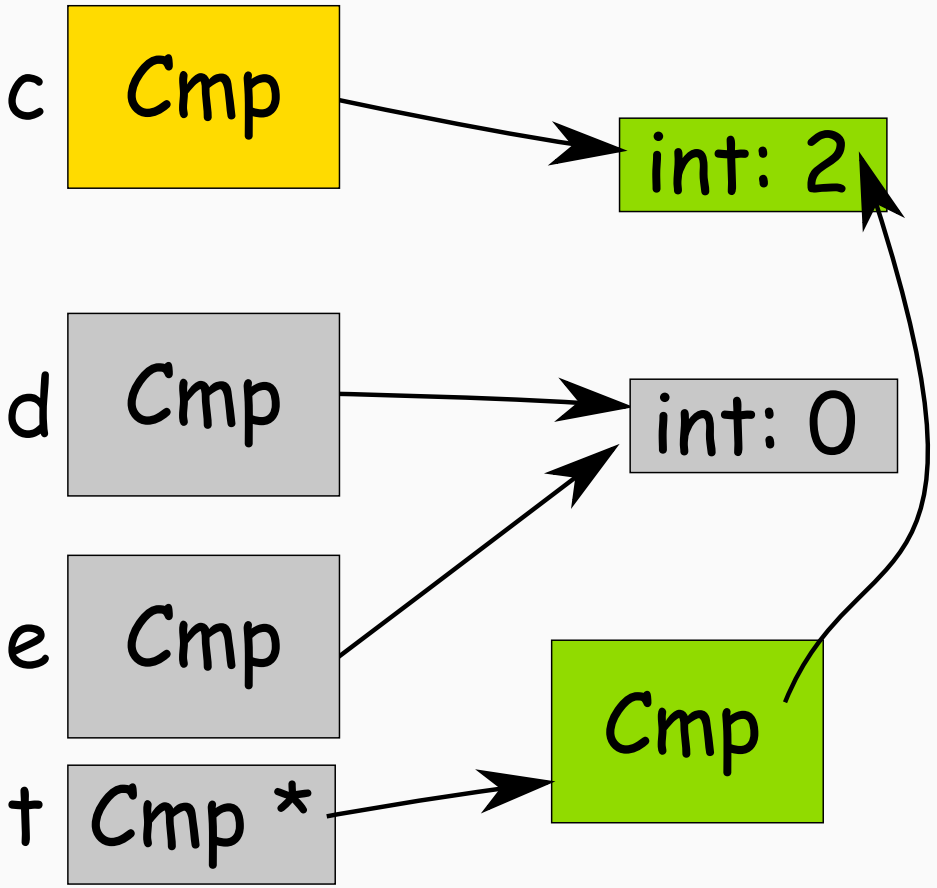
Compteur: Exemple1

```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
t=new Cmp(c);
...
}
```



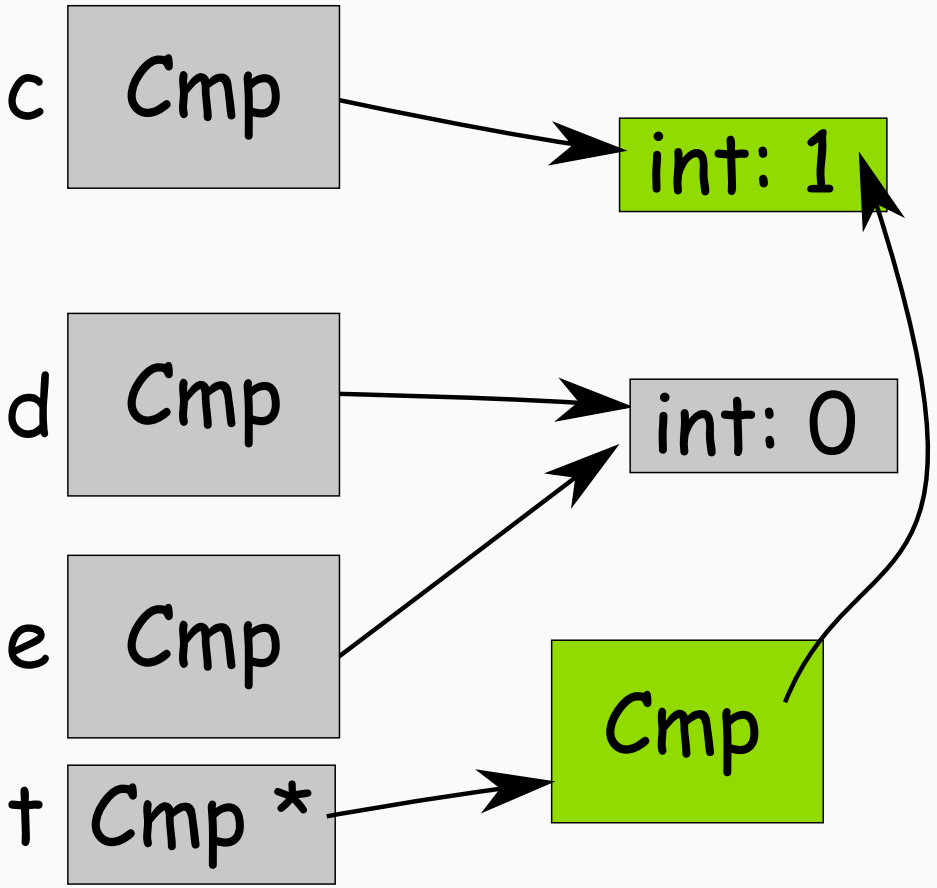
Compteur: Exemple1

```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
t=new Cmp(c);
...
}
```



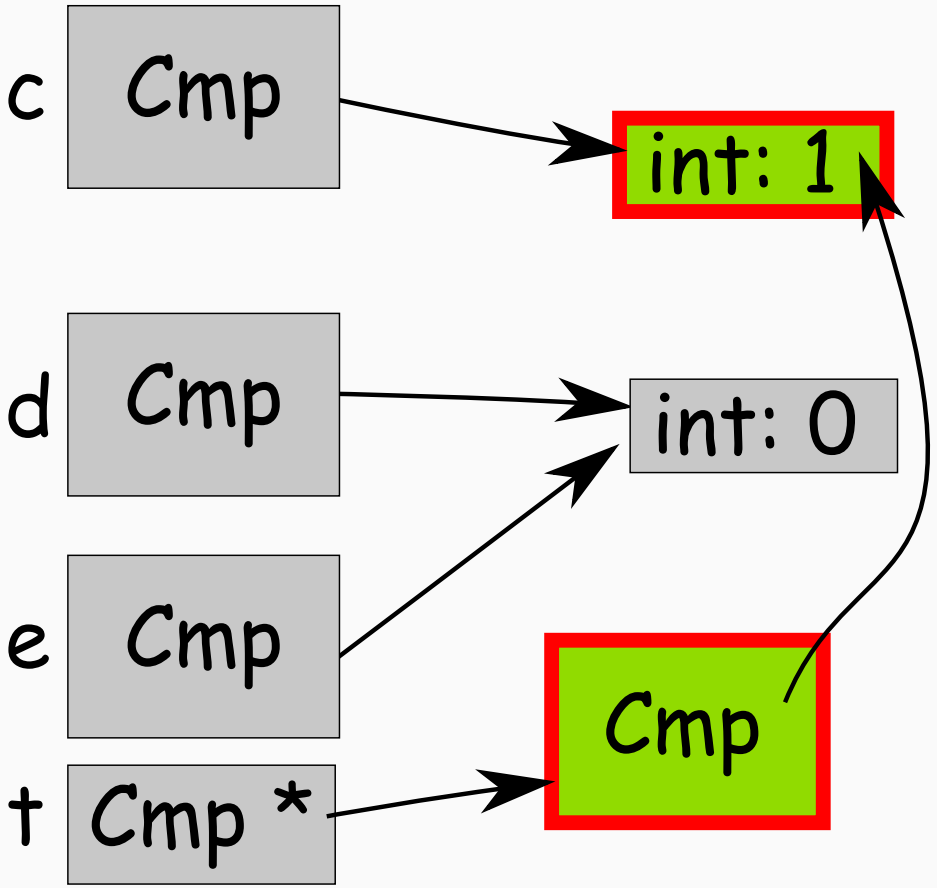
Compteur: Exemple1

```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
t=new Cmp(c);
...
}
```



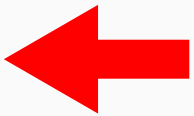
Compteur: Exemple1

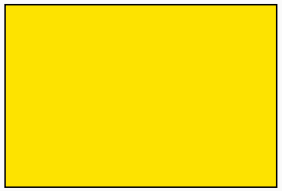
```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
t=new Cmp(c);
...
}
```



Compteur: Exemple2

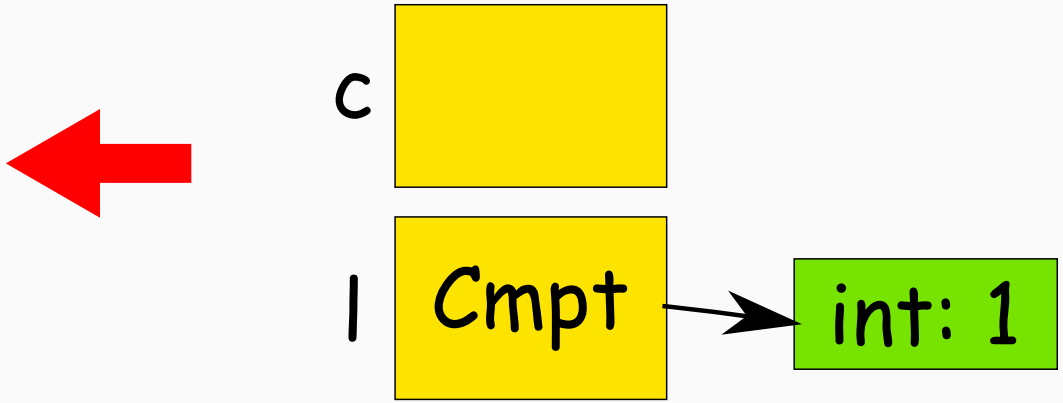
```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```



c 

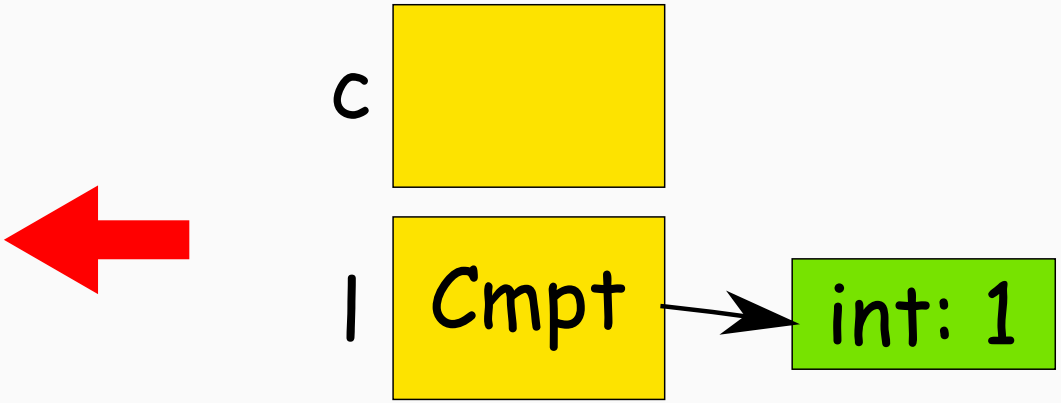
Compteur: Exemple2

```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```



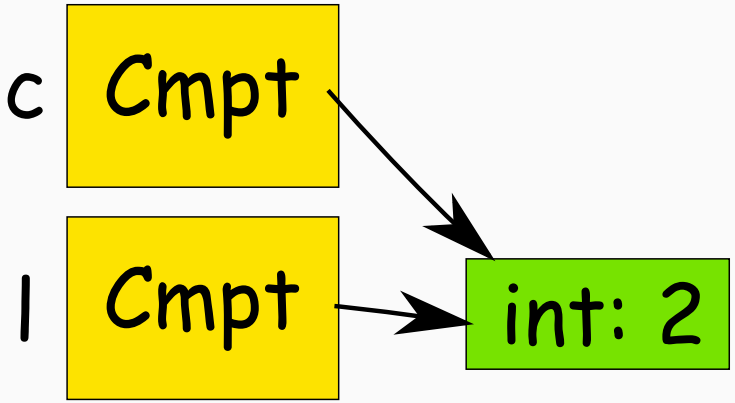
Compteur: Exemple2

```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```



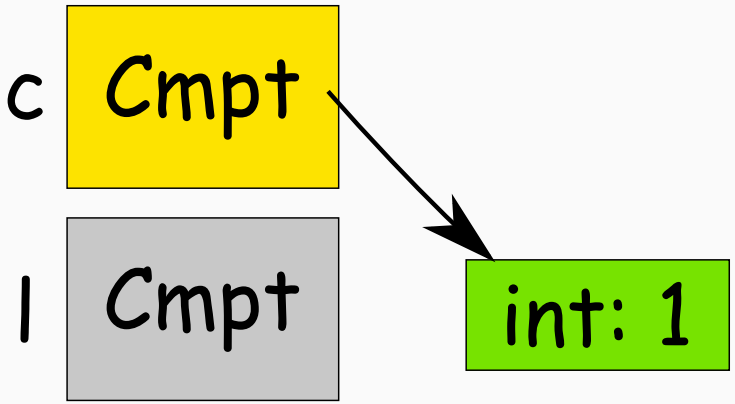
Compteur: Exemple2

```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```



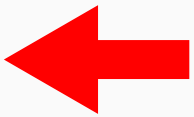
Compteur: Exemple2

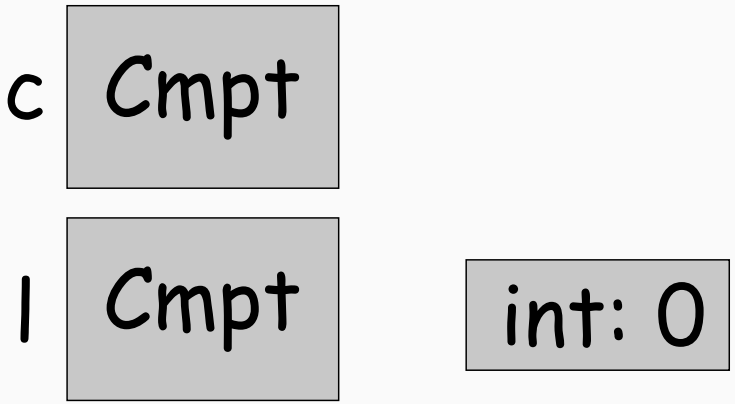
```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```



Compteur: Exemple2

```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```





La classe Compteur

La classe Compteur doit avoir les méthodes suivantes:

- **void** `incremente()` : **private**
- **void** `decremente()` : **private**
- **bool** `dernier()` : **public**

Ainsi que les constructeurs par défaut, recopie et destructeur.

- `Compteur();`
- `Compteur(`**const** `Compteur \&);`
- `~Compteur();`

implémentation de cette classe...

Problème:

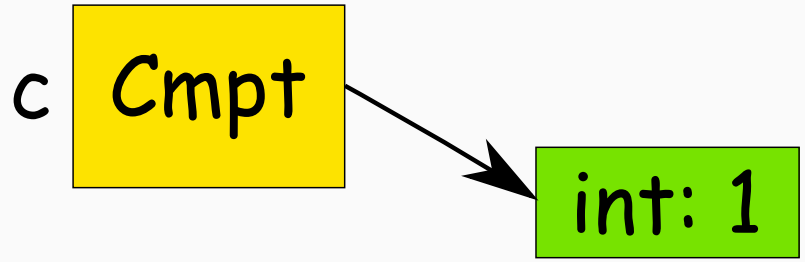
Examinons le source suivant:

```
int main(){
  Compteur c;
  Compteur d;

  d=c; // Recopie de c dans d
}
```

La recopie se fait de la même façon que la construction par recopie. Le compilateur ajoute ici une recopie champs par champs!

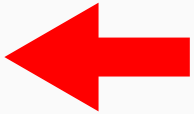
Problème:

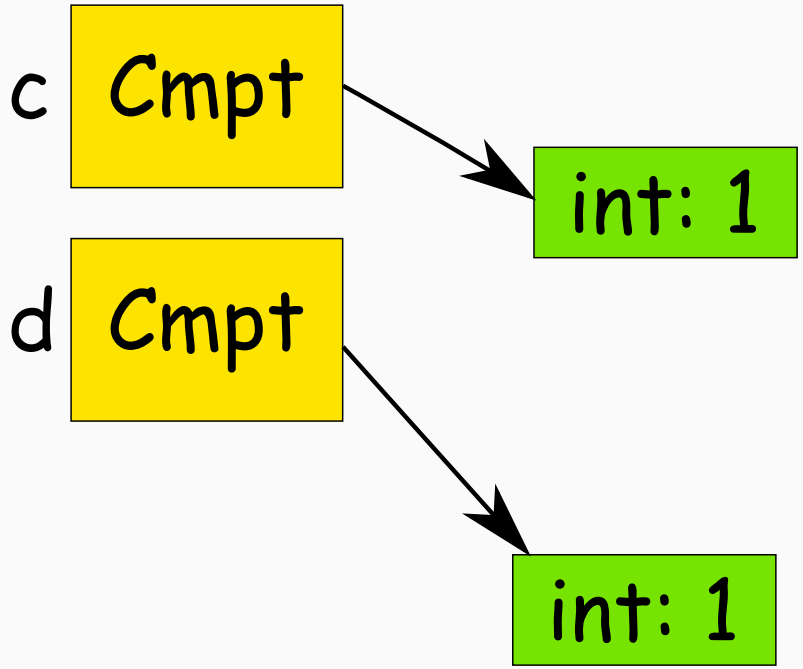
```
int main(){
  Cmpt c;
  Cmpt d;

  d=c;
}
```



Problème:

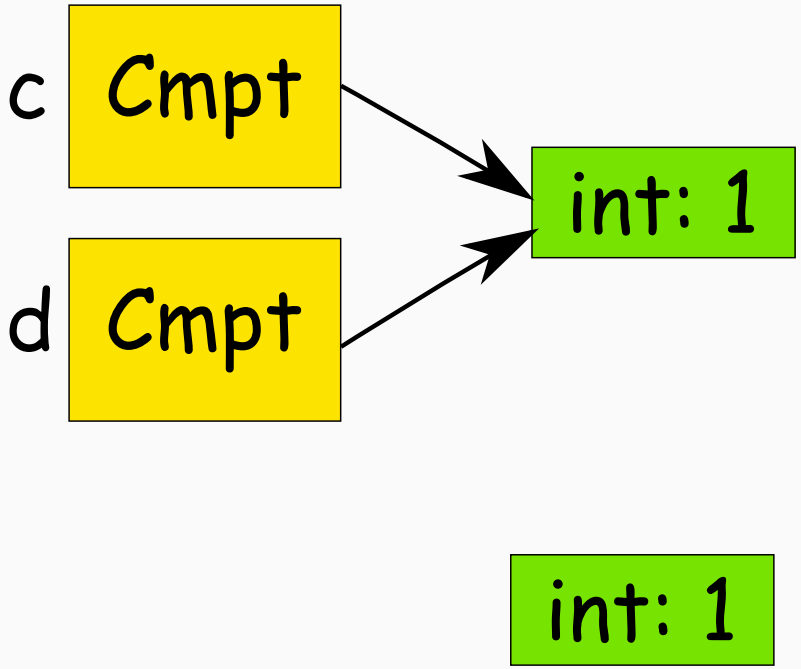
```
int main(){
  Cmpt c;
  Cmpt d;

  d=c;
}
```





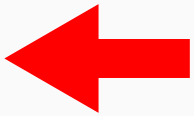
Problème:

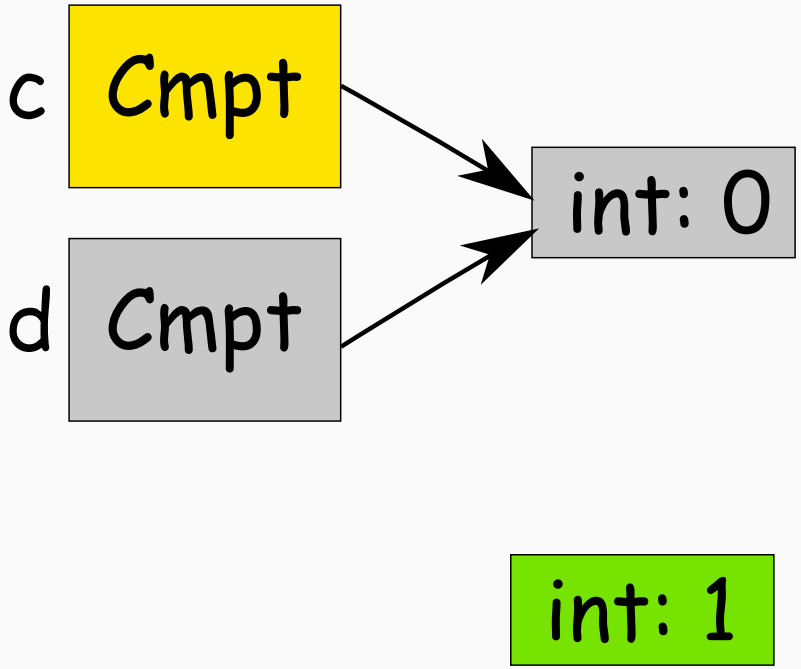
```
int main(){
  Cmpt c;
  Cmpt d;

  d=c;
}
```



Problème:

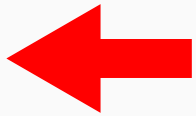
```
int main(){
  Cmpt c;
  Cmpt d;

  d=c;
}
```

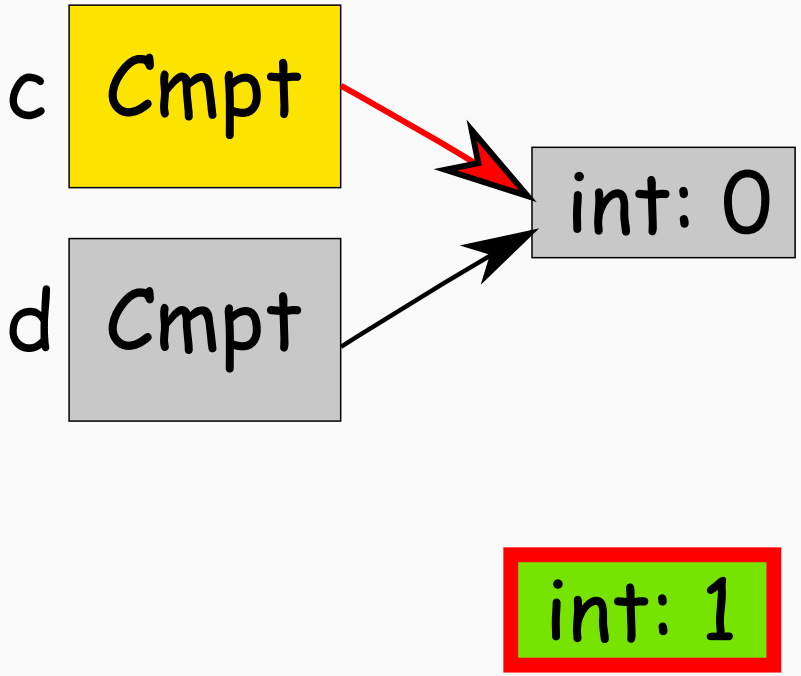




Problème:

```
int main(){
  Cmpt c;
  Cmpt d;

  d=c;
}
```





Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Exemple d'une classe basic Point

Point.hpp

```
class Point{
  double x,y;
  public:
  Point();
  Point(double x,double y);
};
```

Point.cpp

```
#include "Point.hpp"

Point::Point():x(0),y(0){
}
Point::Point(double x,double y):x(x),y(y){
}
```

```
#include <Point.hpp>
int main(){
  Point p(2,3);
  Point q;
  Point *t=new Point[20];
  delete [] t;
}
```

Exemple d'une classe basic Point

Point.hpp

```
class Point{
  double x,y;
  public:
  Point();
  Point(double x,double y);
  double getx() const;
  void setx(double );
};
```

Point.cpp

```
#include "Point.hpp"

Point::Point():x(0),y(0){
}
Point::Point(double x,double y):x(x),y(y){
}
double Point::getx() const{ return x;}
void Point::setx(double d){x=d;}
```

```
#include <Point.hpp>
int main(){
  Point p(2,3);
  Point q;
  Point *t=new Point[20];
  q.getx();
  t[0].getx();
  (t+5)->setx(10);
  delete [] t;
}
```

Exemple d'une classe basic Point

Point.hpp

```
class Point{
  double x,y;
  public:
  Point();
  Point(double x,double y);
  void add(const Point &p);
  Point sum(const Point &p)const;
};
Point fsum(const Point &a,const Point &b);
```

Point.cpp

```
#include "Point.hpp"

Point::Point():x(0),y(0){
}
Point::Point(double x,double y):x(x),y(y){
}
void Point::add(const Point &p){x+=p.x; y+=p.y;}
Point Point::sum(const Point &p)const{
  return Point(x+p.x,y+p.y);
}
Point fsum(const Point &a,const Point &b){
  return Point(a.getx()+b.getx(),a.gety()+b.gety());
}
```

```
#include <Point.hpp>
int main(){
  Point p(2,3);
  Point q;
  Point *t=new Point[20];
  t[2]=p.sum(q);
  q.add(p);
  t[4] = fsum(t[5],t[6]);
  delete [] t;
}
```

Plan

Classes

- Définition et déclaration
- Visibilité, friend, struct
- Attributs et méthodes
- this
- espace de nom
- constructeur
- destructeur
- static
- exemple
- exemple point
- exemple matrix

Exemple d'une classe Matrix

Matrix.hpp

```
class Matrix{
  int l,c;
  double *data;
  public:
  Matrix(int l,int c);
};
```

Matrix.cpp

```
#include "Matrix.hpp"

  Matrix::Matrix(int l,int c):l(l),c(c){
    data= new double[l*c];
}
```

```
#include <Matrix.hpp>
int main(){
  Matrix m(3,3);
}// -> fuite memoire, ajout d'un destructeur
```

Exemple d'une classe Matrix

Matrix.hpp

```
class Matrix{
  int l,c;
  double *data;
  public:
  Matrix(int l,int c);
  ~Matrix();
};
```

Matrix.cpp

```
#include "Matrix.hpp"

  Matrix::Matrix(int l,int c):l(l),c(c){
    data= new double[l*c];
  }
  Matrix::~Matrix(){
    delete [] data;
  }
```

```
#include <Matrix.hpp>
int main(){
  Matrix m(3,3);
} // ok
```

Exemple d'une classe Matrix

Matrix.hpp

```
class Matrix{
  int l,c;
  double *data;
  public:
  Matrix(int l,int c);
  ~Matrix();
};
```

Matrix.cpp

```
#include "Matrix.hpp"

  Matrix::Matrix(int l,int c):l(l),c(c){
    data= new double[l*c];
  }
  Matrix::~Matrix(){
    delete [] data;
  }
```

```
#include <Matrix.hpp>
int main(){
  Matrix m(3,3);
  Matrix n(m);
} // erreur! double free,
// => ajout d'un constructeur par copie
```

Exemple d'une classe Matrix

Matrix.hpp

```
class Matrix{
  int l,c;
  double *data;
  public:
  Matrix(int l,int c);
  Matrix(const Matrix &x);
  ~Matrix();
};
```

Matrix.cpp

```
#include "Matrix.hpp"

  Matrix::Matrix(int l,int c):l(l),c(c){
    data= new double[l*c];
  }
  Matrix::Matrix(const Matrix &x):l(x.l),c(x.c){
    data = new double[l*c];
    for(int i=0;i<l*c;++i)
       data[i]=x.data[i];
  }
  Matrix::~Matrix(){
    delete [] data;
  }
```

```
#include <Matrix.hpp>
int main(){
  Matrix m(3,3);
  Matrix n(m);
} // ok
```

Exemple d'une classe Matrix

Matrix.hpp

```
class Matrix{
  int l,c;
  double *data;
  public:
  Matrix(int l,int c);
  Matrix(const Matrix &x);
  ~Matrix();
};
```

Matrix.cpp

```
#include "Matrix.hpp"

  Matrix::Matrix(int l,int c):l(l),c(c){
    data= new double[l*c];
  }
  Matrix::Matrix(const Matrix &x):l(x.l),c(x.c){
    data = new double[l*c];
    for(int i=0;i<l*c;++i)
       data[i]=x.data[i];
  }
  Matrix::~Matrix(){
    delete [] data;
  }
```

```
#include <Matrix.hpp>
int main(){
  Matrix m(3,3);
  Matrix n(m);
  m=n;
} // erreur! double free + fuite memoire
```

Plan

Operateurs

Opérateurs: définition

Un opérateur est une notation raccourcie permettant de représenter une opération fréquemment utilisée, par exemple:

```
a+b*c;
```

On ajoute à `a` le résultat de la multiplication de `b` par `c`

Si `a` est un pointeur, le sens change: on ajoute à `a`, `b*c*`**sizeof**`(T)` octets (si `T` est le type pointé par `a`).

Dans le cas où `a` n'est pas un type primitif, cette notation est invalide sauf si l'on fournit la fonction permettant d'effectuer cette opération.

Opérateurs: liste

Voici la liste des opérateurs pouvant être définis en C++:

Unaire:

```
!   &   *
+ ++
- --
~
```

spéciaux:

```
new
delete
[ ]
()
conversion
```

Binaire:

```
%     %=     *     *=
-     -=     +     +=
/     /=

<     <=     >     >=

!=    ==     &&    ||

&     &=     |     |=
^     ^=    <<    <<=
>>    >>=
```

Opérateurs: conditions

Pour pouvoir écrire un opérateur, au moins une des opérandes de cet opérateur doit correspondre à un type utilisateur (pas primitif). Par exemple, on ne peut pas modifier l'addition sur les entiers.

Les opérateurs ne sont pas considérés comme commutatifs par le compilateur. C'est à dire que si l'on a écrit un opérateur permettant de faire l'addition d'un `A` avec un `B`,

```
B()+A();
```

ne sera pas résolu pas cet opérateur.

La précédence des opérateurs ne peut être modifiée.

Opérateurs: syntaxe

Un opérateur α s'écrira sous le nom d'une fonction ou méthode non statique de nom `operator` α.

Le type de retour de cette fonction est libre.

C'est le premier paramètre qui est prioritaire pour la résolution. Dans le cas d'une méthode, le premier paramètre (correspondant à **this** dans la méthode) sera une référence du type de la classe.

```
class A{
  public:
  int operator+(int i);
};
```

Il faut voir cette méthode comme définissant l'opération + entre une instance de A non constante et un entier.

Opérateurs: syntaxe

On peut écrire les opérateurs comme des fonctions ou des méthodes:

fonction:

```
class Dix{ };

int operator+(const Dix &d,int i){
  return i+10;
}

int main(){
  Dix g;
  int douze=g+2;
  douze=operator+(g,2);
  operator+(2,g); // Erreur!
}
```

Opérateurs: syntaxe

On peut écrire les opérateurs comme des fonctions ou des méthodes:

méthode:

```
class Dix{
  public:
  int operator+(int i) const{
    return i+10;
  }
};

int main(){
  Dix g;
  int douze=g+2;
  douze=g.operator+(2);
  2.operator+(g); //Erreur!
}
```

Opérateurs: précédence

La précédence des opérateurs ne peut être modifiée, l'évaluation se fait donc obligatoirement selon le respect des précédences standards.

exemple

```
class A{};
class B{};

A operator+(const A&,const B&);
B operator*(const A&,const A&);

int main(){
  A x;
  B y;
  x+y*x; // Erreur !
  (x+y)*x; // ok
}
```

Opérateurs: unaires

On écrira les opérateurs unaires dans les classes:

```
class Point{
  int x,y;
  public:

  Point operator-() const {
    return Point(-x,-y);
  }
};
```

On écrira l'opérateur comme fonction si l'on n'a pas accès à la classe.

Opérateurs: binaires

On écrira les opérateurs binaires dans la classe si les deux opérandes sont de même type et que l'on a accès à la classe.

```
class Point{
  int x,y;
  public:
  Point operator+(const Point &e) const{
    return Point(x+e.x,y+e.y);
  }
};
```

Opérateurs: binaires

Sinon on écrira l'opérateur à l'extérieur de la classe:

```
class Point{
  int x,y;
  ...
};
Point operator+(const Point &p,const Entier &e){
  return Point(p.getx()+e.v(),p.gety()+e.v());
}
Double operator+(const Entier &f,const Point &q){
  return q+f; // operator+(q,f)
}
```

Eventuellement, on pourra déclarer l'opérateur comme fonction amie s'il est nécessaire d'accéder à des attributs privés (???).

Opérateurs: conception

Lorsque l'on écrit un nouvel opérateur, il faut faire attention à ce que l'introduction de ce "raccourci" n'apporte pas d'ambiguité: le but de l'opérateur est de simplifier, pas de compliquer.

En cas de doute, préférer toujours l'utilisation de fonctions ou méthodes classiques.

Un exemple de mauvaise utilisation des opérateurs: la bibliothèque standard!

Opérateurs: iostream

La bibliothèque standard `iostream` fournit un modèle orienté-objet des entrées sorties, elle se decompose en

- `ios` : bases pour les entrées/sorties
- `istream` : entrées
- `ostream` : sorties
- `streambuf` : flux bufferisés
- `iostream` : entrées et sorties
- `fstream` : fichiers
- `sstream` : chaines de caractères

L'ensemble des classes font parties de l'espace de nom `std`

Outre un ensemble de fonctionnalités permettant la gestion des flux, il a été ajouté un support pour la lecture et l'écriture basé sur les opérateurs

Opérateurs: iostream

L'opérateur << est utilisé pour écrire dans un flux de type `std::ostream`:

```
int main(){
  int i;
  std::cout<<"un entier:"<<i<<std::endl; // 1
  ...
  f<<1; // 2
  f<<i<<1; // 3
```

La ligne 1 doit être interprétée comme ceci:

```
operator<<(
  operator<<(
    operator<<(std::cout,"un entier")
    ,i)
  ,std::endl)
```

Qu'en déduire sur le type de retour de l'opérateur << ?

Opérateurs: iostream

```
int main(){
  int i;
  std::cout<<"un entier:"<<i<<std::endl; // 1
  ...
  f<<1; // 2
  f<<i<<1; // 3
```

Qu'en déduire sur le type de retour de l'opérateur << ?

le type de retour de l'opérateur << dans ce cas doit être compatible avec le premier argument.

Opérateurs: iostream

```
std::cout<<"un entier:"<<i<<std::endl; // 1
f<<1; // 2
f<<i<<1; // 3
```

Dans le cas 2, est-ce que l'on est en train de décaler un entier ou bien d'afficher 1 dans un flux ?

Dans le cas 3, est-ce:

- f décalé de i puis de 1 ?
- i décalé de 1 affiché dans f ?

$\Rightarrow$ d'une façon générale on essayera de ne pas "détourner" les opérateurs pour une utilisation qui n'est pas en rapport avec le sens premier de cette opérateur

Implanter le décalage à gauche pour une classe Entier qui multiplie l'entier par 2^i a du sens.

Opérateurs: iostream

La lecture depuis un flux se fait de la façon suivante

```
int i,j,k;
char texte[10];
std::cin>>i; // lecture d'un entier
std::cin>>texte; // lecture d'une chaine !!
std::cin>>i>>j>>k; // lecture de 3 entiers
```

Pour la lecture de chaîne de caractère on utilisera un type qui encapsule des mécanismes de réallocation telle que la classe `std::string`

Quel est le type de retour de cet opérateur?

Le type de retour doit compatible avec le premier argument:

```
std::istream &operator>>(std::istream &, ... )
```

Pourquoi les références ne sont pas constantes ?

classe et iostream

Comment adapter votre classe pour pouvoir utiliser l'affichage de la bibliothèque standard:

```
class Point{ int x,y; };
int main(){
  Point p;
  std::cerr<<"debug"<<p;
}
```

Erreur à la compilation:

```
A.cpp: In function 'int main()':
A.cpp:42: error: no match for 'operator<<' in
  'std::operator<< [with _Traits = std::char_traits<char>]
  (((std::basic_ostream<char, std::char_traits<char> >&)(& s
  ((const char*)"debug")) << p'
```

suivi d'une bonne cinquantaine de lignes !

Solution

nous devons écrire l'opérateur `<<` pour la classe `std::ostream` et `Point`:

```
class Point{
  int x,y;
};
std::ostream &operator<<(std::ostream &stream,Point p){
  stream<<p.x<<'/'<<p.y;
  return stream;
}
```

Combien d'erreurs dans ce code ? Au moins 2:

- p devrait être une référence constante
- p.x p.y ne sont pas accessibles

solutions:

- Ajout d'un accesseur dans le code.
- Déclarer l'opérateur comme amie de la classe

Solution

```
class Point{
  int x,y;
public:
  int getx() const {return x;}
  int gety() const {return y;}
};

std::ostream &operator<<(std::ostream &stream,
                         const Entier &p){
  stream<<p.getx()<<'/'<<p.gety();
  return stream;
}
```

Solution

```
class Point{
  int x,y;
public:
  friend std::ostream &operator<<(
              std::ostream &stream,
              const Point &p);
};
std::ostream &operator<<(std::ostream &stream,
                         const Point &p){
  stream<<p.x<<'/'<<p.y;
  return stream;
}
```

Si l'implémentation change (l'attribut privé est renommé), on doit aussi changer le code de l'opérateur :(

$\Rightarrow$ les fonctions amies ne doivent pas être utilisées à la place des accesseurs!

Exemple classique

```
class Point{
  int x,y;
public:
  Point(int x=0,int y=0):x(x),y(y){}
  bool operator==(const Point &p) const {
    return (x==p.x) && (y==p.y);
  }
  bool operator<(const Point &p) const {
    if  (x<p.x) return true;
    if (x==p.y) return y<p.y;
    return false;
  }
  //... <= > >= != ...
  Point operator+(const Point &p)const{
    return Point(x+p.x,y+p.y);
  }
  Point&operator+=(const Point &p){
    x+=p.x; y+=p.y;
    return *this;
  }
  //... - -= * *= ..
};
```

Opérateur: d'affectation

L'opérateur d'affectation correspond au '=' et doit nécessairement être écrit sous la forme d'un membre de classe:

```
class Point{
  int x,y;
public:
  const Point &operator=(int i){
    x=i; y=0;
    return *this;
  }
};
...
Point p;
p=1;
Point f=1; // Erreur: c'est un appel au constructeur!
```

Opérateur: d'affectation

Le compilateur fournit toujours un opérateur d'affectation permettant de copier une instance dans une instance de même type:

```
struct _S a,b;
a=b;
```

L'opérateur d'affectation fournit est de la forme:

```
X &X::operator=(const X&); // 1
X &X::operator=(X&);       // 2
```

Dans les deux cas, la copie est effectuée attribut par attribut dans l'ordre de leur déclaration.

La forme 1 est utilisée si chaque attribut possède un opérateur d'affectation en forme 1.

Si l'on déclare explicitement un opérateur d'affectation en forme 1 ou 2 alors le compilateur n'ajoute plus l'opérateur.

Opérateur: d'affectation

L'opérateur d'affectation fournit par le compilateur ressemble à:

```
class X{
  A a;
  B b;
  C c;
  //....
public:
X &operator=(const X&y){
  a=y.a;
  b=y.b;
  c=y.c;
  //...
  return *this;
 }
};
```

Matrix

Matrix.hpp

```
class Matrix{
  int l,c;
  double *data;
  public:
  Matrix(int l,int c);
  Matrix(const Matrix &x);
  Matrix &operator=(const Matrix &m);
  ~Matrix();
};
```

À partir du moment où une classe effectue de l'allocation dynamique, on écrira systématiquement un constructeur par recopie, un destructeur et un opérateur d'affectation

Matrix.cpp

```
#include "Matrix.hpp"

Matrix::Matrix(int l,int c):l(l),c(c){
  data= new double[l*c];
}
Matrix::Matrix(const Matrix &x):l(x.l),c(x.c){
  data = new double[l*c];
  for(int i=0;i<l*c;++i)
     data[i]=x.data[i];
}
Matrix &Matrix::operator=(const Matrix &m){
  if (this==&m) return *this;
  delete [] data; // libération
  c=m.c; l=m.l; // re-initialisation
  data=new double[c*l];
  for(int i=0;i<l*c;++i)
     data[i]=m.data[i];
}
Matrix::~Matrix(){
  delete [] data;
}
```

Opérateur: foncteur

L'opérateur de fonction `()` est le seul opérateur d'instance (i.e. à part new et delete) qui peut prendre un nombre variable d'arguments. Pour les autres, le nombre d'arguments est fixé par l'arité de l'opérateur.

Une classe qui implémente cet opérateur est appelée foncteur (ou fonctor) car l'on peut alors utiliser une instance comme une fonction:

```
class Greater{
  public:
  bool operator()(int a,int b){return a>b; }
  bool operator()(double a,double b){
    return a>b;
  }
  bool operator()(const char *a,const char *b){
    return strcmp(a,b)>0;
  }
};
```

Opérateur: foncteur

```
class Greater{
  public:
  bool operator()(int a,int b);
  bool operator()(double a,double b);
  bool operator()(const char *a,const char *b);
};
...
Greater greater;
if (greater(1,0)){ ... }
if (greater("aba","abb")){...}
```

On aurait aussi bien pu écrire plusieurs fonctions greater en utilisant la surcharge.

Un autre exemple plus utile (et complexe) est l'implémentation d'une fonction de hachage paramétrable (attributs).
Concept utilisé avec les templates (prog. générique).

Opérateur: de conversion

Il est possible d'écrire des opérateurs prenant en charge la conversion d'une instance vers un autre type:

```
class Entier{
  int _value;
  public:
  operator int()const{
    return _value;
  }
};
```

Le prototype de l'opérateur de conversion est **operator** `type()` **const**, ne possède pas de type de retour (puisque c'est `type`) et ne prend pas d'argument.

L'opérateur de conversion s'écrit obligatoirement sous la forme d'une fonction membre.

Opérateurs: de conversion

L'opérateur de conversion peut être appelé explicitement avec un cast ou bien implicitement:

```
void f(int );
...
Entier e;
int i=e; // conversion implicite
(int )e; // conversion explicite
f(e); // conversion implicite
```

Les opérateurs de conversions sont pris en compte par l'algorithme de résolution.

Opérateur: de conversion

Une utilisation intéressante de l'opérateur de conversion est la conversion en `std::string`:

```
#include<string>
class Entier{
  public:
  operator int() const{ return _value;}
  operator std::string ()const{
    std::stringstream s;
    s<<_value;
    return s.str();
  }
};
...
Entier e;
cout<<(std::string )e;
```

Cette approche est plus logique (que l'implémentation de <<) et permet plus de possiblités.

Opérateurs: autres

L'opérateur `[]` est souvent utilisé. Il prend *un seul argument* et doit obligatoirement être écrit comme fonction membre.

```
class Tab{
  int *data;
  int size;
public:
int operator[](int i) const{
  return data[i];
  }
int &operator[](int i){
  return data[i];
  }
};
```

permet la lecture et l'écriture.

remarque: ce n'est pas équivalent à un 'setter' car on n'a pas de contrôle sur la valeur modifiée

Opérateur []

Attention: l'operateur `[]` ne peut prendre qu'un seul paramètre:

```
Matrix m(5,5);
m[2][3] = 10;
m.operator[](2).operator[](3) = 10;
```

Cela donnerait par exemple:

```
class Matrix{
  int **data;
  //...
  int *operator[](int i){ return data[i];}
  const int *operator[](int i) const{ return data[i];}
  };
```

Le deuxième appel à `[]` est l'opérateur classique des pointeurs. On n'a pas de contrôle sur le dépassement!

Opérateur () vs []

Il *pourrait* être préférable d'utiliser l'operateur `()`:

Cela donnerait par exemple:

```
class Matrix{
  int **data;
  //...
  int operator()(int i,int j) const {
    return data[i][j];
  }
  void operator()(int i,int j, int v) {
    return data[i][j]=v;
  }
};
```

Opérateur: conclusion

- Les opérateurs sont des raccourcis syntaxiques permettant l'appel à des méthodes
- L'opérateur d'affectation = à une fonction très particulière. Il est fournit par le compilateur si l'on en écrit pas un.
- Il est impératif d'implémenter l'opérateur d'affectation si une classe gère des ressources via ses attributs (allocation dynamique).
- Les opérateurs permettent de donner à des instances des comportements similaires aux types primitifs : leur usage prend sens dans le contexte de la *programmation générique*.

Plan

Héritage, polymorphisme

Héritage: intro

Jusqu'à présent, nous avons vu des mechanismes d'encapsulation: le C++ permettant de faire des *struct améliorées*:

- ajout de comportements
- ajout de la visibilité
- méthodes d'initialisation (constructeurs) et de libération (destructeurs)
- contrôle de la copie

Plus l'ajout de nouvelles méthodes d'allocation dynamique (new), des espaces de noms (encapsulation), des références.

Héritage: définition

L'héritage est un mécanisme permettant de construire un type `T` à partir d'un autre type `Base`. Le type `T` se retrouve doté des comportements (méthodes) et propriétés (attributs) du type de Base. Une relation de typage relie ces deux types: `T` est un sous-type de `Base` (ou classe dérivée de `Base`) tandis que `Base` est un super-type de `T`.

En C++, les membres hérités et la relation de typage qu'il existe entre `T` et `Base` ne sont pas necessairement visible de l'extérieur.

La relation est indiquée lors de la déclaration de `T`:

```
class T: public Base{ ...
```

Nous reviendrons sur le sens du **public** plus tard.

Héritage: exemple

Voici un exemple classique d'héritage:

```
class Tuyau{
  double _diametre;
public:
  double debit() const;
  double diametre() const;
};
```

```
class TuyauPer : public Tuyau{
  double _densite;
public:
  double densite() const;
};
```

```
TuyauPer tper;
Tuyau &tuyau=tper; // relation de typage
tper.densite();    // ok
tuyau.densite();   // erreur!
tper.debit();      //ok
```

Héritage: pointeurs et référence

L'utilisation de la relation de typage ne peut se faire qu'avec les pointeurs ou les références.

En effet, pour faire jouer la relation de sous-typage on souhaite manipuler une même instance à travers différentes variables de types différents.

```
TuyauPer tper;
Tuyau t=tper; // Erreur !
Tuyau *p=&tper; // ok
Tuyau &r=tper; // ok
TuyauPer &r2=r; // Erreur !
```

Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un premier paramètre caché (**this** dans la méthode).

Lors d'un appel, la méthode choisie est celle du type de la variable. Si la méthode n'est pas présente directement dans le type, c'est la méthode compatible de son parent le plus proche qui est choisie.

```
class A{
public:
  void f(char ){ puts("A\n");}
};
class B:public A{
public:
  void f(int ){ puts("B\n");}
};
class C:public B{};
```

```
C c;
c.f(1) ; // =>B
c.f('a'); // =>B
c.A::f('A'); // => A
c.B::f('a'); // => B
B &b=c;
b.f('a'); // => B
```

On utilise l'opérateur de portée pour spécifier la méthode à appeler.

Héritage: appel

```
class A{
public:
  void m(){}
};
```

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

```
B b;
A a;
A &r=b;
b.m(); // b est de type B => B::m()
a.m(); // a est de type A => A::m()
r.m(); // r est de type A => A::m()
A *p=&b;
p->m(); // p est de type A* => A::m()
```

Héritage: et mangling

Le “mangling” désigne la façon de générer un nom de symbole à partir d’une fonction ou méthode:

```
$g++ -c /tmp/mang.c
$nm mang.o
         U __gxx_personality_v0
00000000 T _ZN1A1mEv
00000006 T _ZN1B1mEv
```

La première fonction correspond à `A::m()`, la deuxième à `B::m()`

Il faut savoir que cette étape d’encodage n’est pas normalisée et est propre à chaque compilateur. Ainsi, les objets compilés avec un compilateur ne seront sans doute pas exploitable avec un autre compilateur!

En C, il n’y a pas de “mangling”, une fonction génère un symbole de même nom que cette fonction.

Question

Le polymorphisme ne fonctionne qu'avec l'utilisation des pointeurs et des références?

A) vrai

B) faux

Question

A ne possède ni attribut, ni méthode, ni classe de base. Que vaut sizeof(A)?

A) 0

B) 1

C) 4

D) 4 si 32 bits, 8 en 64 bits

Héritage: et constructeur

Lors de la contruction d'une instance de la classe `Fille`, sa classe de base, la classe `Mere` doit elle aussi être initialisée.

Pour initialiser la partie `Mere`, le compilateur ajoute un appel au constructeur par défault:

```
class Mere{
  public:
  Mere(){puts("Mere");}
};
class Fille:public Mere{
  public:
  Fille(){puts("Fille");}
};
```

```
Fille f;
/*
Affiche:

Mere
Fille
*/
```

Comment faire si la classe `Mere` n'a pas de constructeur par défaut?

Héritage: et constructeur

On indique le constructeur à appeler comme ceci:

```
class Mere{
  public:
  Mere(int i){puts("Mere");}
};
class Fille:public Mere{
  public:
  Fille():Mere(1)
  {puts("Fille");}
};
```

```
Fille f;
/*
Affiche:

Mere
Fille
*/
```

L'appel au constructeur de la classe `Mere` doit se faire avant l'initialisation des attributs de la classe.

Dans le cas contraire le compilateur inverse les initialisations.

Héritage: et constructeur

```
class Mere{
  public:
  Mere(int i){
    printf("Mere: %d\n",i);
  }
};
class Fille:public Mere{
  int v;
  public:
  Fille():v(1),Mere(v)
  {puts("Fille");}
};
```

```
Fille f;
/*
Affiche:

Mere: -1479451508
Fille
*/
```

Si l'on compile avec `-Wall`, le compilateur nous signal l'inversion, sinon rien n'est dit.

Héritage: et destructeur

De même que pour le constructeur, les appels au destructeur se font en "cascade". C'est à dire qu'à la fin du destructeur de la classe `Fille`, un appel au destructeur de la classe `Mere` est ajouté:

```
class Mere{
  public:
  Mere(){puts("Mere");}
  ~Mere(){puts("~Mere");}
};
class Fille:public Mere{
  public:
  Fille(){puts("Fille");}
  ~Fille(){puts("~Fille");}
};
```

```
{
Fille f;
}
/*
Affiche:

Mere
Fille
~Fille
~Mere
*/
```

Héritage: et opérateur d'affectation

Nous avons vu que le compilateur ajoute dans les classes un opérateur d'affectation si celui-ci n'est pas écrit.

L'opérateur ajouté par le compilateur fait appel à l'opérateur de la classe `Mere`:

```
class Mere{
  int _v;
  public:
  Mere(int v=0):_v(v){}
  const Mere &operator=(const Mere &){
     printf("Mere:%d\n",_v);
     return *this;
  }
};
class Fille:public Mere{
  Mere m;
  public:
  Fille():Mere(10){}
};
```

```
{
Fille f,g;
f=g;
}
/*
Affiche:

Mere:10
Mere:0
*/
```

Héritage: et opérateur d'affectation

Voici à quoi ressemble l'opérateur d'affectation ajouté par le compilateur:

```
class Fille: public Mere{
  ...
  public:
  const Fille &operator=(const Fille &f){
    Mere::operator=(f);
    this->attribut1=f.attribut1;
    this->attribut2=f.attribut2;
    ...
  }
};
```

Dans le cas où nous voulons écrire notre propre opérateur, c'est à nous d'appeler explicitement l'opérateur d'affectation de la classe `Mere`.

Héritage: et opérateur d'affectation

```
class A{
  public:
  const A&operator=(const A&){
    puts("A=A"); return *this;}
};
class M{
  A a;
};
class F: public M{
  public:
  const F&operator=(const F&f){
    return *this;
  }
};
```

```
F f,g;

f=g;
// Pas d'affichage!
```

Héritage: et opérateur d'affectation

```
class A{
  public:
  const A&operator=(const A&){
    puts("A=A"); return *this;}
};
class M{
  A a;
};
class F: public M{
  public:
  const F&operator=(const F&f){
    M::operator=(f);
    return *this;
  }
};
```

```
F f,g;

f=g;
// Affiche A=A
```

On peut faire appel à l'opérateur ajouté par le compilateur!

C++11 et delete

Le C++11 introduit un nouvel usage du mot clé delete:

```
class M{
  public:
  void f() = delete;
};

M m;
m.f();
```

Cela produit le message d'erreur suivant:

```
op_del.cpp:9:6: error: use of deleted
    function 'void M::f()'
9 |   m.f();
```

C++11 et delete

Un usage courant de cette fonctionalité est de supprimer les fonctionalités ajoutées automatiquement par le compilateur:

```
class A{
  int m;
public:
  A(int x) : m(x) {}
  A& operator = (const A &) = delete;
  A(const A&) = delete;
};

A a1(1), a2(2);
a1 = a2;  // Error
A a3(a2); // Error
```

C'est très souvent associé à l'héritage: dès qu'il y a de l'héritage, on va supprimer la copie.

Héritage: public, protected et private

Il existe trois possibilités pour l'héritage, public: **class** `T:` **public** `Base`, protected: **class** `T:` **protected** `Base` et private: **class** `T:` **private** `Base`. L'héritage est privé par défaut dans les classes et publique pour les structures.

Dans le cas de l'héritage **public**, la relation entre `T` et `Base` est visible de tous. Les méthodes **public** de `Base` sont **public** dans `T` et les méthodes **protected** sont **protected** dans `T`.

Dans le cas de l'héritage **protected**, les méthodes **public** et **protected** de `Base` sont **protected** dans `T`.

Dans le cas de l'héritage **private**, les méthodes **public** et **protected** de `Base` sont **private** dans `T`.

Héritage: public, protected et private

On ne peut utiliser un relation type/sous-type que si l'on peut avoir accès à un membre publique de la classe `Base` à travers `T`.

Si l'héritage est publique, tout le monde voit les méthodes publiques de `Base` à travers `T`. Ainsi on pourra toujours voir un `T` comme un `Base`.

Si l'héritage est protégé alors seules les fonctions/classes amies, sous-classes et amies des sous-classes pourront voir un `T` comme un `Base`.

Si l'héritage est privé alors seules les fonctions/classes amies pourront voir un `T` comme un `Base`.

Héritage: public, protected et private

```
class A{
...
};
class B: public A{
...
};
class C: protected A{
...
};
class D: public C{
...
};
class E: private A{
...
};
class F: public E{
...
};
```

```
B b;
A&a=b; // ok

C c;
A *p=&c; // erreur!

void D::f(){
  A &r=*this; // ok
}

void F::f(){
  A &r=*this; // erreur
}

void E::f(){
  A &r=*this; // ok
}
```

Héritage: public, protected et private

L'héritage **protected** ou **private** permet de récuperer une implémentation.

C'est une alternative efficace à la délégation (lien a-un).

Un exemple: on veut implémenter une classe `Pile`. Pour cela on souhaite utiliser la classe `Vector`.

Or, une pile n'est pas un vecteur particulier, on ne souhaite donc pas que la classe `Pile` présente toutes les méthodes de la classe `Vector`.

La solution sans héritage privé consiste à utiliser la délégation: on dispose d'un attribut de type `Vector` que l'on utilise pour implémenter la pile.

Si l'on souhaite que cette "délégation" soit accessible aux sous classes alors on utilise l'héritage protégé sinon on utilise l'héritage privé.

Héritage: et polymorphisme

Jusqu'à présent, l'héritage permet le factorisation de code mais ne permet pas de mettre en oeuvre le polymorphisme.

En effet, pour qu'il y ai polymorphisme il faut que la méthode appelée sur une instance soit celle correspondant au type réel de l'instance et non au type de la variable manipulant cette instance.

Pour mettre en oeuvre le polymorphisme il faut donc utiliser l'instance elle-même pour connaître son type.

Ceci est implanté grace à un attribut caché appelé `vtable`. C'est un pointeur vers une table référençant toutes les méthodes pour lesquelles le polymorphisme doit être mis en oeuvre.

Héritage: et virtual

Tout d'abord il nous faut indiquer quelles sont les méthodes "virtuelles", c'est à dire pour lesquelles on veut du polymorphisme.

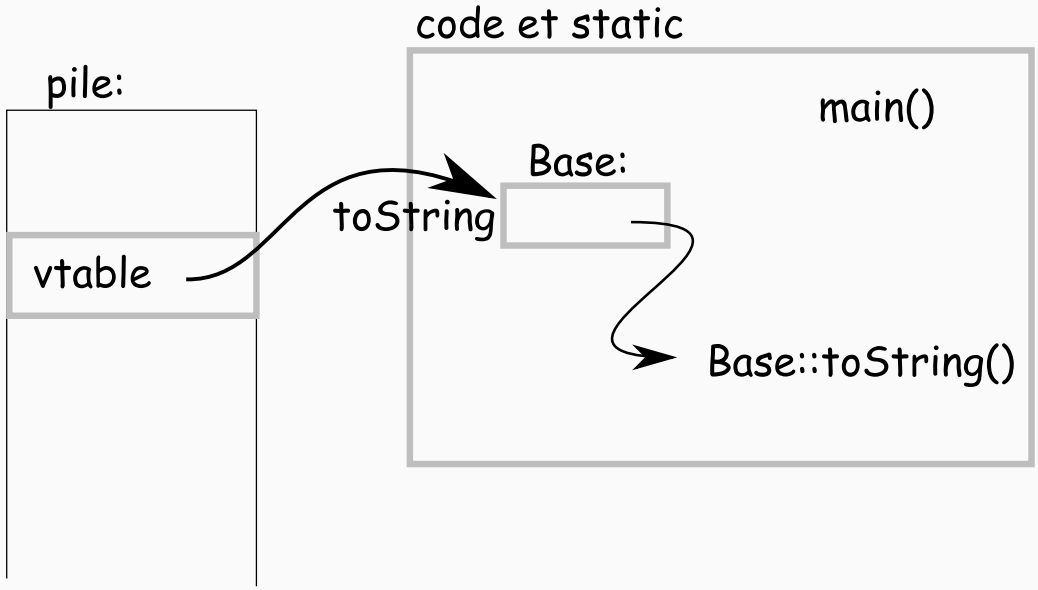
```
class Base {

public:
  virtual std::string toString() const {
    return "Base";
  }
};
```

On notera que la présence de la `vtable` fait "grossir" les instances de la classe: on passe ici de 1 octet à 4 octets (sur un machine 32 bits).

Ceci crée une table associée à la classe Base. Cette table est une table de pointeurs de fonctions: les méthodes virtuelles

Héritage: vtable

```
Base b;
```



Question

Chaque instance d'une classe ayant des méthodes virtuelles possède une table des méthodes virtuelles?

A) vrai

B) faux

Héritage: vtable

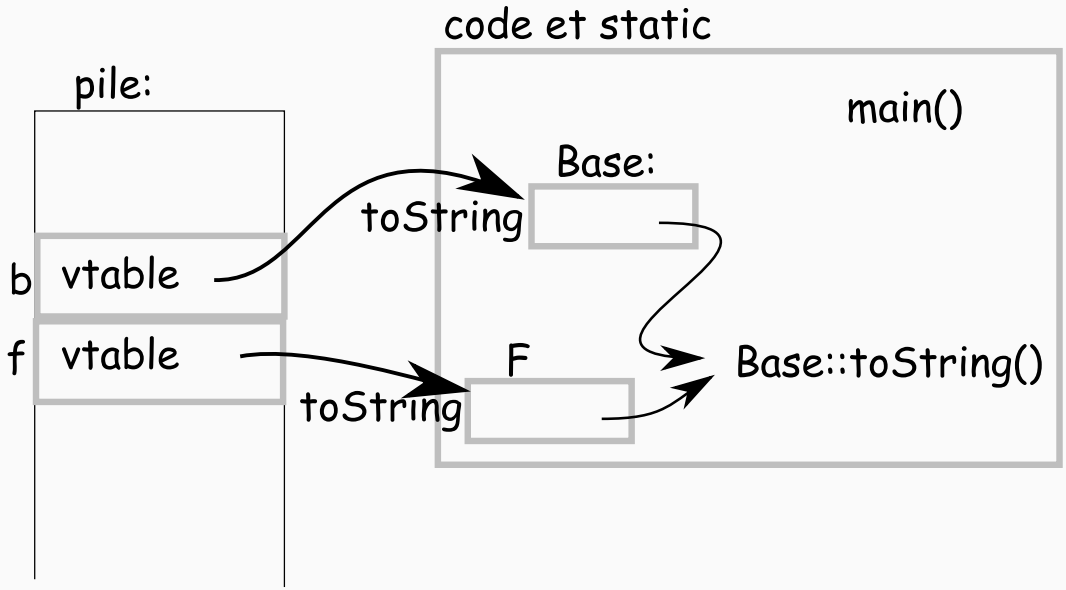
Ajoutons une classe dérivée de `Base`:

```
class F : public Base{
  public:
};
```

Si l'on ne redéfini pas la méthode `toString` alors on obtient le schéma suivant

Héritage: vtable

```
Base b;
F f;
```



Une table des fonctions virtuelles est créée pour la classe `F`. Celle-ci est initialement construite par “recopie” de la table de la classe de base.

Dans les constructeurs, le compilateur ajoute une instruction permettant d’initialiser l’attribut caché `vtable` pour qu’il pointe vers la table correspondant au type de l’instance.

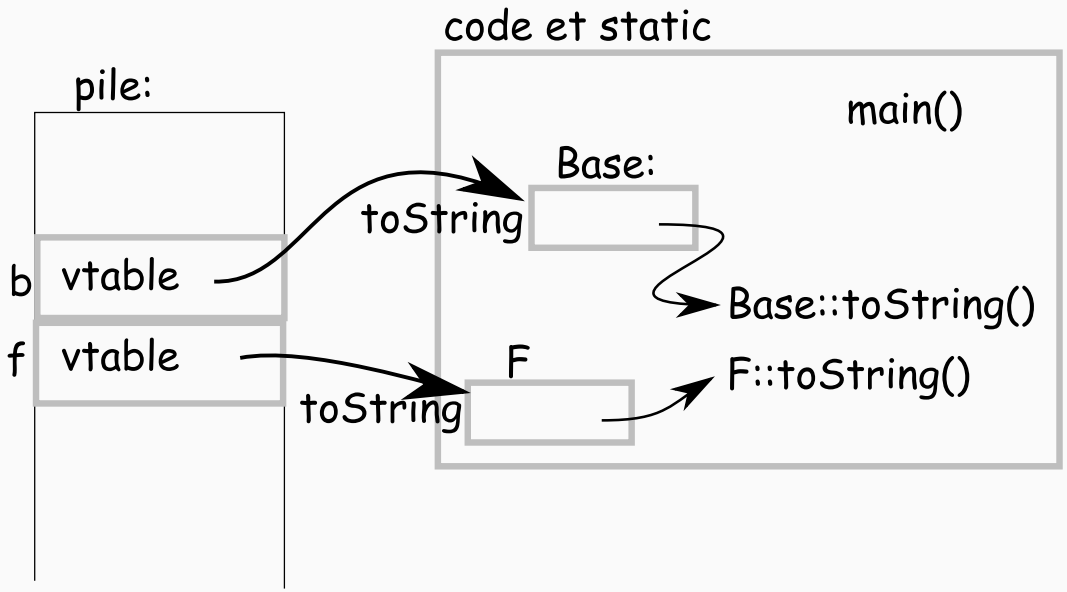
Héritage: vtable

Cas où l'on redéfini la méthode dans la classe `F`:

```
class F : public Base{
  public:
  virtual std::string toString() const{
    return "Fille";
  }
};
```

Héritage: vtable

```
Base b;
F f;
```



Si une méthode est redéfinie: même nom, même arguments (même type) alors l'entrée dans la table correspondante à cette méthode est modifiée.

On voit que ce mécanisme ne modifie en rien le code de la méthode, c'est la façon dont la méthode va être appelée qui va être changé.

Héritage: appel d'une méthode virtuelle

Lors de l'appel à une méthode sur une instance, le compilateur commence par chercher cette méthode dans la classe correspondante au type de la variable sur laquelle la méthode est appelée.

```
Fille f;
Base &r=f;

r.toString();
```

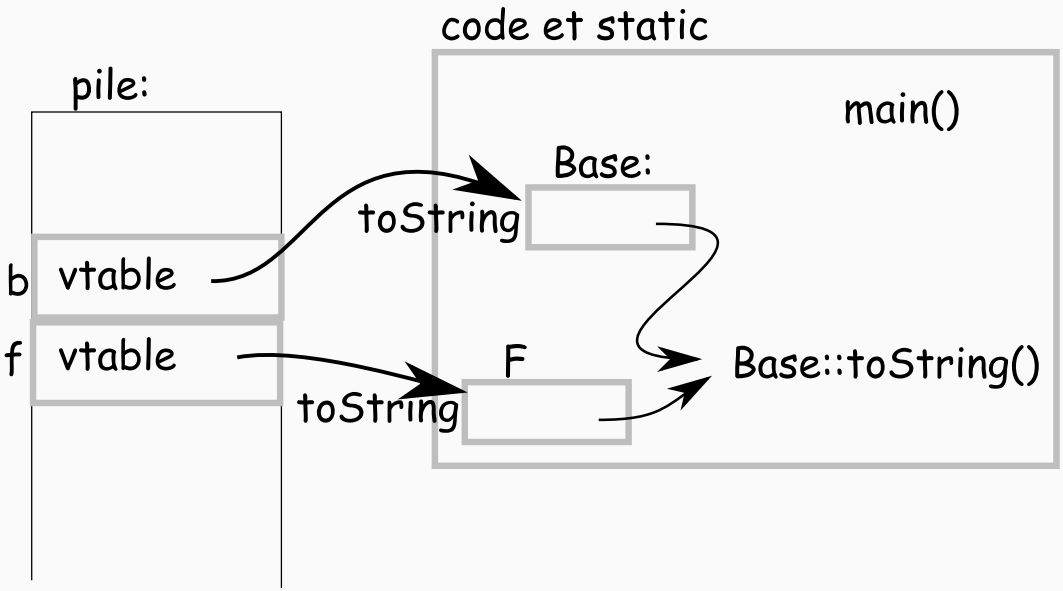
Une fois cette méthode trouvée, il y a deux possibilités: soit la méthode est virtuelle, soit elle ne l'est pas.

Si elle n'est pas virtuelle, alors c'est la méthode du type de la variable qui est appelée, dans l'exemple `Base::toString`.

Si elle est virtuelle, alors c'est la méthode dont l'adresse est dans la `vtable` qui est appelée, dans l'exemple cela dépend si elle a été redéfinie.

Héritage: vtable

```
Fille f;
Base &r=f;


r.toString();
```



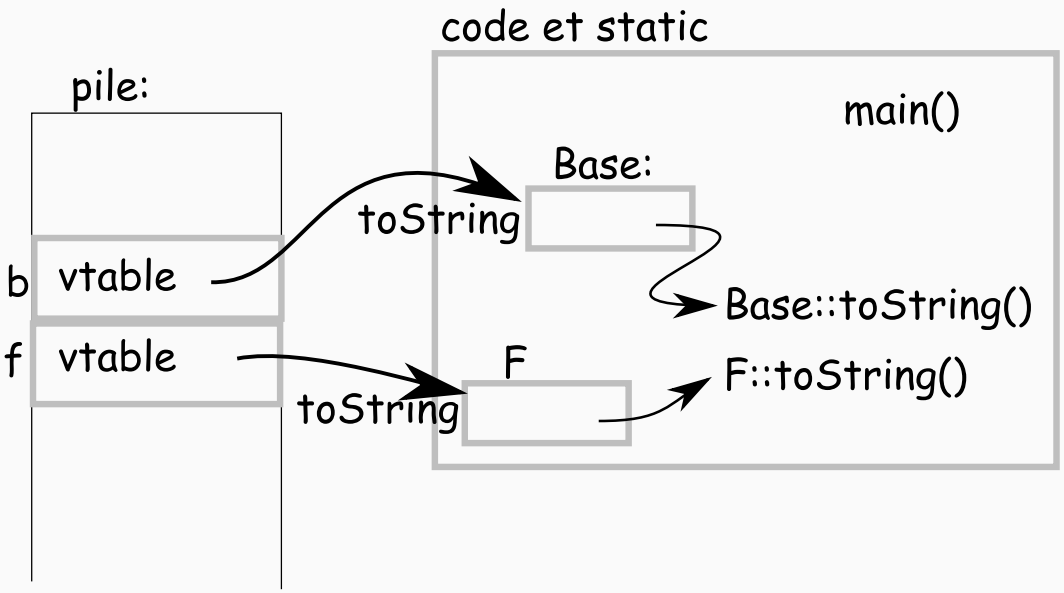
L'appel à la méthode peut être vu comme

```
r.vtable["toString"]();
```

Dans cet exemple, cela correspond à l'appel de `Base::toString()` car la méthode n'a pas été redéfinie.

Héritage: vtable

```
Fille f;
Base &r=f;


r.toString();
```



L'appel à la méthode peut être vu comme

```
r.vtable["toString"]();
```

Dans cet exemple, cela correspond à l'appel de `Fille::toString()` car la méthode a été redéfinie.

Héritage: virtuelle pure

Une méthode est dite “virtuelle pure” si elle n’est pas implémentée dans la classe où elle est déclarée.

```
classe Base{
  public:
  virtual std::string toString() const =0;
};
```

Le `=0` est bien compréhensible: cela correspond à mettre à 0 le pointeur dans la table des fonctions virtuelles!

Une classe n’est instanciable que si sa table des fonctions virtuelles ne comporte pas de 0.

Dans cette exemple, `Base` n’est donc pas instanciable et toute sous classe de `Base` ne sera instanciable que si elle redéfinie la méthode `toString`.

Héritage: virtuelle pure

Soit le code suivant:

```
class Base{
  public:
  virtual void m(const Base &)=0;
};

class Fille : public Base{
  public:
  virtual void m(const Fille &){}
}
```

Question: Quelle affirmation est vraie?

A) `Base` est instanciable.

B) `Fille` est instanciable.

C) `Base` et `Fille` sont instanciables.

D) aucune n'est instanciable.

Héritage: et redéfinition

La redéfinition d'une méthode consiste à écrire dans une classe dérivée une méthode qui

- porte le même nom
- a le même nombre et type d'arguments
- a comme type de retour un sous-type du type de retour de la méthode redéfinie ou le même type.

Question

L'appel à une méthode non virtuelle sur une instance NULL génère systématiquement un SEGFAULT

A) vrai

B) faux

Héritage: et valeur par défaut

En C++, les arguments des fonctions et méthodes peuvent prendre des valeurs par défaut pour les arguments:

```
void f(int =0){}

f(); // Appel f(0);
```

Les valeurs par défaut doivent être précisées dans la déclaration, en effet elle indique au compilateur comment compléter l'appel avec ces valeurs si nécessaire.

Les valeurs par défaut doivent être données du dernier argument vers le premier argument afin d'éviter toute ambiguité.

Héritage: et valeur par défaut

```
class M{
public:
  virtual ~M(){}
  virtual void h(int =0)=0;
};
class F: public M{
  public:
  virtual void h(int i=1){}
};
```

```
int main()
{
  F x;
  M &y=x;

  y.h(); // appel de F::m(0);
  x.h(); // appel de F::m(1);
}
```

Bien que l'appel soit virtuel, c'est le type de la variable qui est utilisé pour connaître les valeurs par défaut des arguments!

Héritage: et valeur par défaut

```
class M{
public:
  virtual ~M(){}
  virtual void h(int =0)=0;
};
class F: public M{
  public:
  virtual void h(int i){}
};
```

```
int main()
{
  F x;
  M &y=x;

  y.h(); // appel de F::m(0);
  x.h(); // Erreur !
}
```

D'une manière générale on evitera de mélanger redéfinition, surcharge et valeur par défaut: il faut toujours (c'est aussi valable pour les opérateurs, exceptions...) que le comportement attendu soit "limpide".

Héritage: et using

Lorsque l'on mélange redéfinition et surchage il peut arriver des choses "bizarre":

```
class M{
  public:
  void h(const M&);
};
class F: public M{
  public:
  void h(const F&);
};
```

```
F x,y;
M &p=x,&q=y;
x.h(y); // 1. ok
p.h(q); // 2. ok
p.h(x); // 3. ok
x.h(p); // 4. erreur!
```

Question: quelle méthode est appelée par `p.h(x);`?

A) M::h

B) F::h

Pour le cas 4, comme `F` hérite de `M` on pourrait s'attendre à ce que la méthode `M::m` soit appelée.

Algorithme de résolution

Lors de l'appel d'une méthode `m` sur une variable `v` de type `V` (ptr ou ref) pointant sur une instance de type `I`, la méthode est cherchée de la manière suivante:

- Existe-t-il des méthodes de nom `m` dans la classe `V`?⇒ using
- Oui (partie 1):
  - Si une méthode est compatible: elle est choisie
  - Si plusieurs méthodes sont compatibles: ambiguité
  - Si aucune méthode n'est compatible: erreur.
- Non (partie 2):
  - On cherche `m` dans la hiérarchie de `V`. Lorsqu'une méthode est trouvée, on applique la partie 1.
- Une méthode a été trouvée (sinon erreur).
- Si elle est virtuelle:
  - On effectue l'appel en utilisant la `vtable`
  - Si la méthode a été redéfinie dans `I` alors c'est `I::m` qui est appelée.
  - Sinon c'est la méthode du plus proche parent.

Algorithme de résolution: remarque

Si une variable `v` de type `T` n'est ni une référence ni un pointeur, alors la résolution se fait à la compilation.

Même si la méthode appelée est virtuelle, il n'y aura pas utilisation de la `vtable`.

Appel avec opérateur de portée

Lorsque l'on utilise l'opérateur de portée, l'appel de méthode se fait en inspectant la classe indiquée dans l'appel et sans tenir compte du **virtual**.

```
struct A{
  virtual void m(){}
};
struct B: A{
};
struct C: B{
  virtual void m(){}
};
```

```
C z;
A &x=z;
B &y=z;
x.m();
z.B::m();
y.m();
y.B::m();
```

Les appels utilisant l'opérateur de portée ne passent pas par la `vtable`, la résolution se faisant à la compilation.

Héritage: et using

On peut utiliser le mot clé **using** pour inclure des méthodes parent dans les méthodes à inspecter:

```
class M{
  public:
  void h(const M&);
};
class F: public M{
  public:
  using M::h;
  void h(const F&);
};
```

```
F x,y;
x.h(y); // 1. ok
M &p=x,&q=y;
p.h(q); // 2. ok
p.h(x); // 3. ok
x.h(p); // 4. ok
```

Le **using** `M::h;` dans la classe `F` indique au compilateur d'inclure la ou les méthodes de nom `h` dans la classe `M` dans l'algorithme de résolution.

Question

À ce stade du cours, l'utilisation de using dans une classe fille par rapport à une méthode mère suppose l'usage de la surcharge

A) vrai

B) faux

Héritage: classe abstraite

On appel "classe abstraite" une classe ayant au moins une méthode virtuelle pure.

En C++, il n'existe pas de définition d'interface (dans la norme), cependant on pourra appeler interface une classe ne disposant que de méthodes virtuelles pures (et un destructeur virtuel).

Les classes abstraites et les interfaces ne sont pas instanciables.

Une classe abstraite ou interface ne peut pas être utilisée comme type de retour par recopie ou comme argument par valeur (recopie).

On ne peut utiliser que des pointeurs et références sur ces types.

Héritage: et interface

Exemple d'une interface:

```
class Usager{
  public:
  virtual ~Usager(){}
  virtual std::string nom() const=0;
  virtual void monterDans(Transport &)=0;
};
```

Soit I une interface, laquelle de ces syntaxes est invalide?

A) `I *i;`

B) `I &i=j;`

C) `I i;`

D) `I &*i=p;`

Héritage: et classe abstraite

Exemple d'une classe abstraite:

```
class PassagerAbstrait{
  protected:
    std::string _nom;
    int _etat;
    int _destination;
    ...
    virtual choixPlaceMontee(Bus &b)=0;
    ...
  public:
    virtual ~PassagerAbstrait(){}
    void estDebout() const {
      return _etat==DEBOUT;
    }
};
```

Héritage: et destructeur

Si une classe va être dérivée alors elle **doit** avoir un destructeur virtuel (implémenté):

```
class A{
};
class B{
  std::string s;
  public:
  B():s("hello"){}
};
...
{
  B b;
} // ok: le destructeur de B appelle le
// destructeur de la classe std::string sur s
```

Héritage: et destructeur

Si une classe va être dérivée alors elle **doit** avoir un destructeur virtuel (implémenté):

```
class A{
};
class B{
  std::string s;
  public:
  B():s("hello"){}
};
...
A *a = new B;
delete a; // Fuite mémoire, le destucteur de
// std::string n'est pas appelé.
```

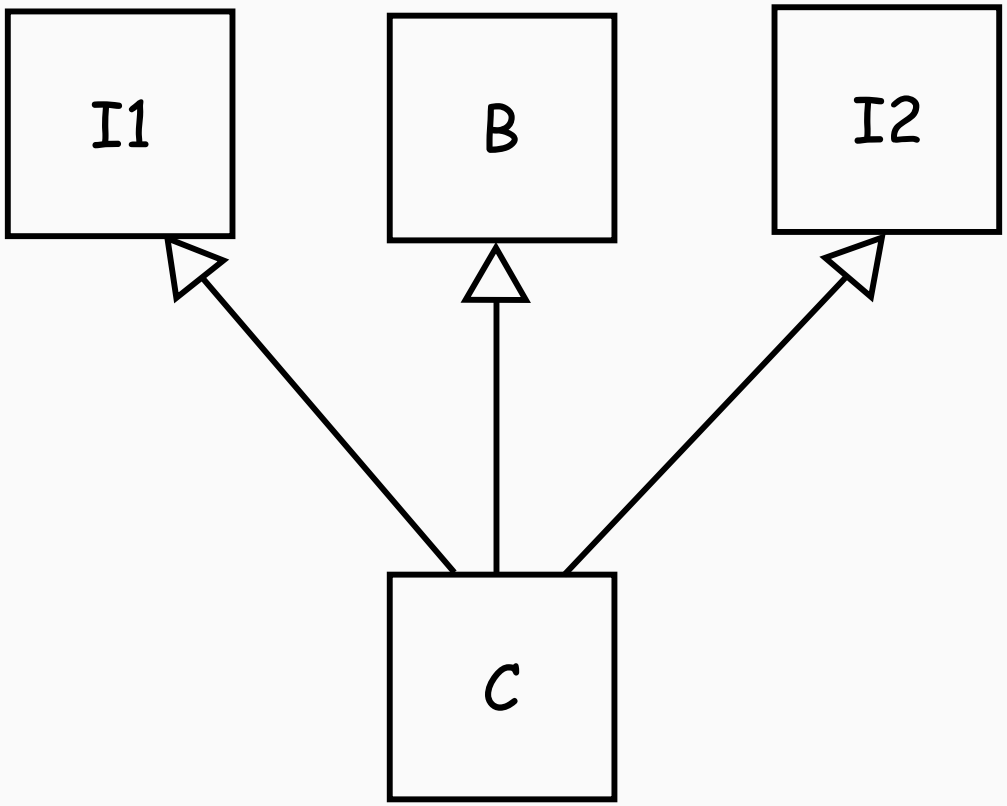
Héritage multiple

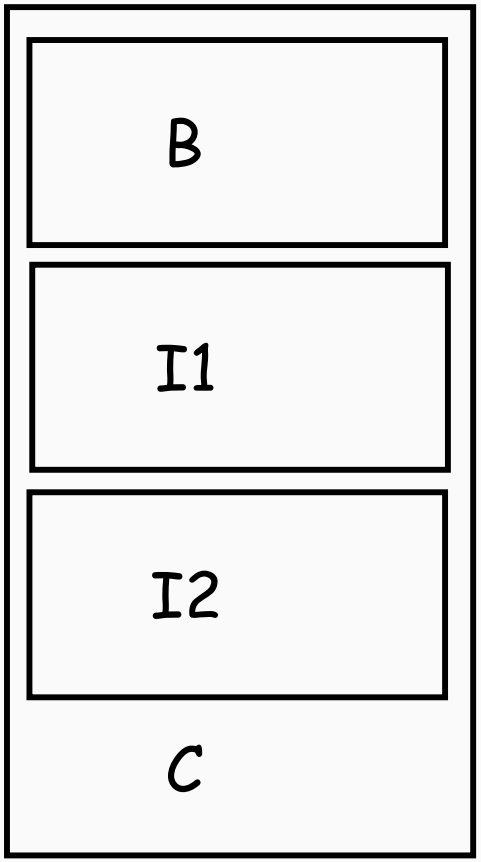
En C++, une classe peut avoir plusieurs classes de base, que celles-ci soient abstraites ou non:

```
class I1{ ... };
class I2{ ... };
class B{ ...};
class C: public B, public I1, public I2 {
  // implémentation de toutes le méthodes
  // virtuelles pures.
};
```

La classe `C` hérite ici de trois classes. Ainsi, on pourra se représenter `C` en mémoire de la façon suivante:

Hérite multiple: exemple





Héritage multiple

L'opérateur de portée et le **using** sont très utils pour lever les ambiguités:

```
class I1{
  public:
  void m();
};
class I2{
  public:
  void m();
};

class F: public I1, public I2{
  ....
  m(); // Erreur!
  I1::m(); // ok
  ...
};
```

Héritage multiple

L'opérateur de portée et le **using** sont très utils pour lever les ambiguités:

```
class I1{
  public:
  void m();
};
class I2{
  public:
  void m();
};

class F: public I1, public I2{
  ....
  using I2::m;
  m(); // ok
  I1::m(); // ok
  ...
};
```

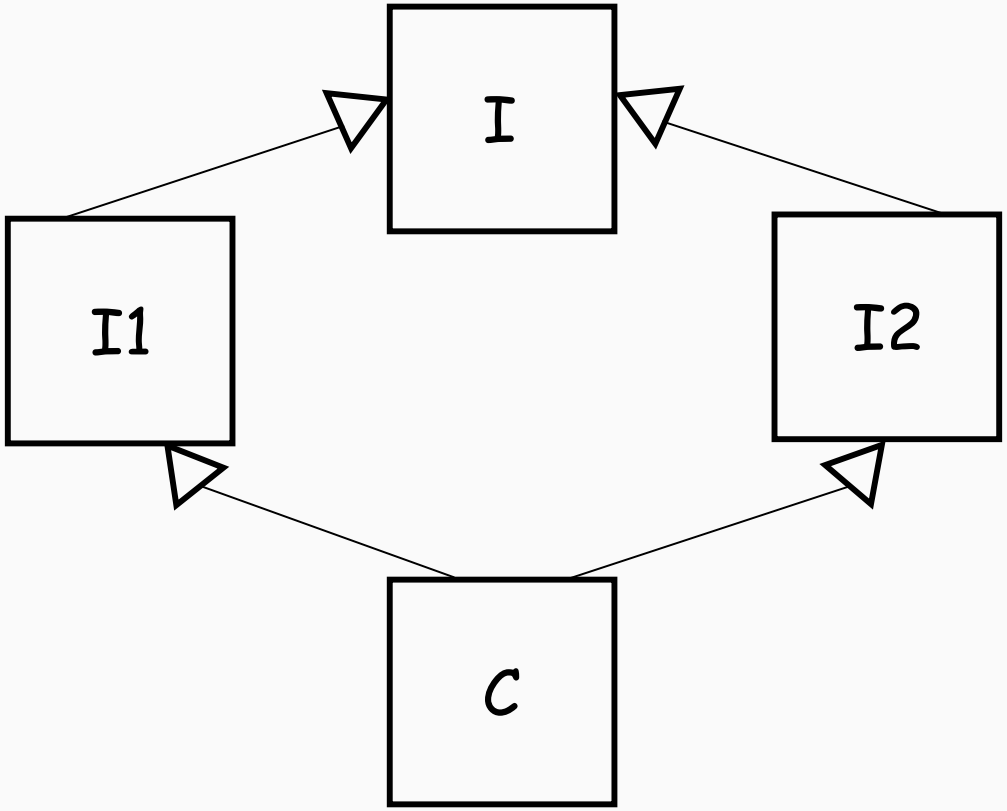
Héritage multiple virtuel

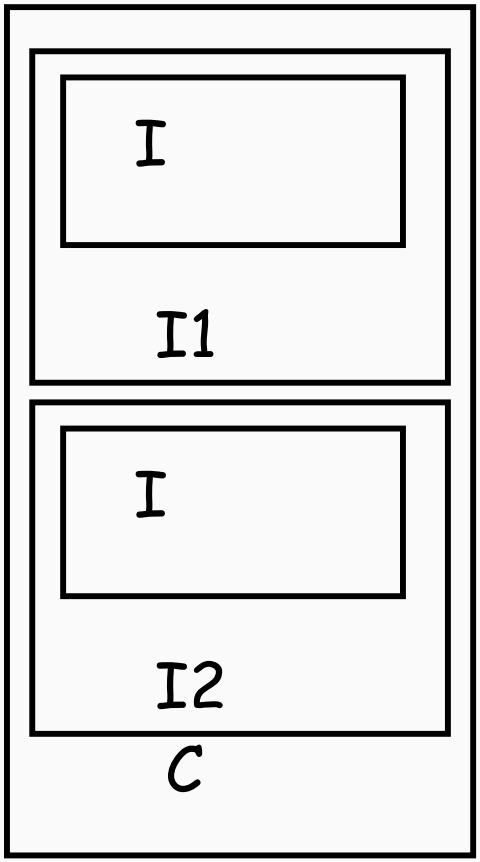
Il peut arriver qu'une classe possède plusieurs fois une même classe comme classe de base:

```
class I{ public: int _value; };

class I1: public I{};

class I2: public I{};

class C: public I1, public I2{};
```

Dans ce cas, C dispose de deux instances de la classe I:

Hérite multiple: exemple





Héritage multiple: ambiguïté

Ainsi, une instance de la classe `C` à deux attributs `_value`:

```
C c;
c._value ; // Erreur!
```

L'opérateur de portée permet de lever cette ambiguïté:

```
C c;
c.I1::_value ; // Ok
c.I1::I::_value ; // Ok
c.I2::_value ; // Ok
```

Héritage multiple: virtuel

Dans le cas d'un héritage en losange comme celui que nous venons de voir, les classes `I1` et `I2` pourraient vouloir partager leur classe de base `I`.

Pour cela, on utilise l'héritage virtuel.

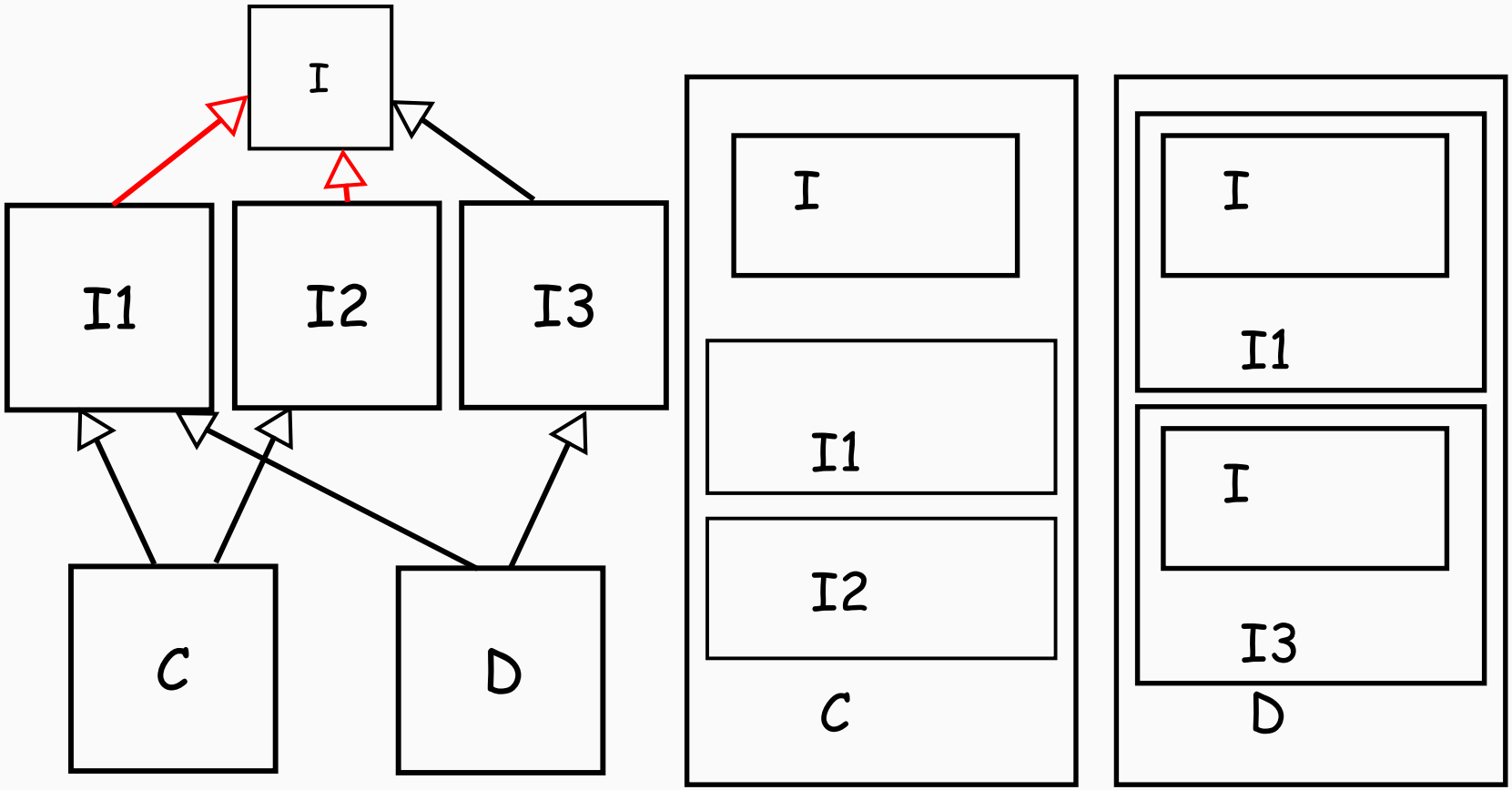
L'héritage virtuel permet d'indiquer qu'une classe est prête à partager une de ces classes de base avec une autre classe qui aurait fait de même:

```
class I{};
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};
```

Une instance de la classe `C` ne contient qu'un seul `I`

Une instance de la classe `D` en possède deux

Hérite multiple: virtuel



Hérite multiple: virtuel

L'héritage virtuel permet de lever des ambiguïtés:

```
class I{ public: virtual void m()=0; };
class I1 : public virtual I{};
class I2 : public virtual I{};
class I3 : public I {};
class C : public I1, public I2{};
class D : public I1, public I3{};

C c;
c.m(); // ok
D d;

d.m(); // Erreur!
```

On utilisera systématiquement l'héritage virtuel lors de l'héritage d'une interface.

Dans le cas ci-dessus, `I` doit bien entendu posséder un destructeur virtuel.

Hérite multiple: virtuel

D'une façon générale, on n'utilisera pas l'héritage virtuel si la classe de base possède des propriétés (attributs) pouvant changer.

Si l'héritage virtuel concerne une classes de base *ayant des attributs*, alors un *attribut caché* est ajouté par le compilateur (un peu comme la `vtable`). Il y a donc un coût mémoire.

On pourra s'autoriser l'héritage virtuel si la classe de base est une interface, ou bien une classe ne possédant que des méthodes virtuelles pures, un constructeur par défaut et des attributs dont la valeur est fixée lors de la construction.

Plan

Template

Template: présentation

En plus du support de la programmation objet, le `C++` ajoute au `C` la programmation générique.

La programmation générique consiste à écrire du code (fonction ou classe) indépendamment de certains types. Ceux-ci seront fixés plus tard lors de l'utilisation de ce code.

En `C++` c'est le mot clé **template** qui permet la définition de la généricité

On préfixera le code générique d'une intruction **template**<**class** `T`> où `T` devient un type variable, paramètre du code.

Template: présentation

On pourra écrire deux types de code générique: des classes génériques et des fonctions génériques.

Voici un exemple de fonction générique:

```
template<class T>
void swap(T &a, T&b){
  T t=a;
  a=b;
  b=t;
}
```

Cette fonction effectue l'echange de valeur entre `a` et `b` quel que soit leur type.

On voit cependant que `a` et `b` doivent être de même type

L'utilisation de cette fonction `swap` se fait comme suit:

```
template<class T>
void swap(T &a, T&b){...}

int a=0;
int b=1;
char c='a';
```

```
swap<int>(a,b); // 1
swap(a,b); // 2
swap(a,c); // 3
```

Dans le cas 1, nous indiquons explicitement le type pour `T`, le compilateur remplace alors `T` par **int** et trouve la fonction `swap<int>(int &,int &)` qui convient à l'appel. Par la même occasion, il compile cette fonction.

Dans le cas 2, le compilateur cherche la fonction `swap` et essaye de déduire un type convenable pour `T` à partir des arguments (ceci n'est pas toujours possible). Il construit `swap<int>` et compile cette fonction.

Dans le cas 3, il n'arrive pas à trouver une fonction `swap` avec le bon prototype et génére une erreur.

Template: et compilation

Un problème se pose pour la compilation d'un code template.

En effet, celui-ci étant à trous, il ne peut être compiler qu'une fois les types de paramètres connus.

Pour cela, on ne peut plus mettre seulement le prototype dans le `.hpp`, nous metterons aussi le code de la fonction:

en une fois:

```
template<class T>
void swap(T &a, T& b){
  T t=a;
  a=b;
  b=t;
}
```

en deux fois:

```
template<class T>
void swap(T &, T& );

...

template<class T>
void swap(T &a, T& b){
  T t=a;a=b;b=t;
}
```

Template: classe

Voici un exemple de classe paramétrée (ou générique):

```
template<class T>
class Entier{
  T _value;
  public:
  Entier(const T &t):_value(t){}
};
```

Dans le cas d'une classe, on écrira donc le code directement dans le fichier `.hpp`

On peut tout à fait séparer les déclarations de l'implémentation:

```
template<class T>
class Entier{
  T _value;
  public:
  Entier(const T &t);
};
```

```
template<class T>
Entier<T>::Entier(const T &t):_value(t)
{}
```

Template: classe

Lors de l'utilisation d'une classe générique, nous devons indiquer explicitement le type à utiliser pour les paramètres:

```
Entier a; // Erreur!
Entier<int> b; // ok
Entier<long> c; // ok
```

On parlera de type complet pour désigner une classe générique dont l'ensemble des paramètres sont spécifiés.

Il est important de retenir qu'il n'existe aucune relation de typage entre deux types complets correspondant à une même classe générique.

Template: valeur par défaut

On peut spécifier des valeurs par défaut pour les paramètres d'une classe générique (pas des fonctions). Celles-ci doivent être fournies de la droite vers la gauche:

```
template<class T=int>
class Entier{...

template<class D, class H=int>
class HashTable{...

template<class T=int,class C>  // Erreur!
int compare(T *, T *)

Entier e; // ok :)
```

Template: et type primitif

Un code peut aussi être paramétré par des valeurs (constantes) de types primitifs:

```
template<int taille>
class Matrix{
  int _data[taille][taille];
  public:
};
```

Dans ce cas, `Matrix<4>` et `Matrix<3>` sont deux types différents. Cet exemple pourrait être utilisé en infographie.

Template: spécialisation

Il est possible de modifier l'implémentation pour certaines valeurs de paramètre:

```
template<class T>
void affiche(const T &t){
  std::cout<<t<<std::endl;
}
```

Dans le cas des entiers, on utilise printf:

```
template<>
void affiche<int>(const int &t){
  printf("%d\n",t);
}
```

Template: spécialisation

Dans le cas où les paramètres sont des types primifs, on peut spécialiser selon les valeurs:

```
template<int n>
void affiche(){
  printf("%d\n",n);
}

template<>
void affiche<0>(){
  printf("zero\n");
}

...
  affiche(); // erreur !
  affiche<1>(); // affiche: 1
  affiche<0>(); // affiche: zero
```

Template: spécialisation

Dans le cas des classes, la spécialisation peut aussi se faire sur les pointeurs:

```
template<class T>
class Tableau{
  T *_data;
  ...
};

template<class T>
class Tableau<T *>{
  T **_data; // Attention ici !
  public:
  Tableau(int i){
    ...
    _data[i]=NULL;
    ...
  }
};
```

Template: spécialisation partielle

Il est possible de ne spécialiser que certains paramètres d'un code générique:

```
template<class T,class H>
class HashTable{
};

template<class T,class H>
class HashTable<T *,H>{
};

template<class T>
class HashTable<T *,int>{
};
```

Important: il n'y a aucune contrainte entre ces différentes implémentations car elle ne correspondent pas au même type

Template: contraintes

Lorsque l'on utilise un type dans un code générique, il faut faire très attention aux propriétés que l'on impose sur ce type:

```
template<class T>
void swap(T &a,T&b){
  T t=a;
  a=b;
  b=t;
}
```

```
template<class T>
void swap(T &a,T&b){
  T t;
  t=a;
  a=b;
  b=t;
}
```

On veillera à toujours indiquer clairement les méthodes attendues pour une type paramètre.

Template: STL et BOOST

Il existe deux grandes bibliothèques utilisant les templates, la Standard Template Library et BOOST.

La STL contient:

- des conteneurs (tableaux, map, ...),
- des iterateurs (parcours),
- des algorithmes (comparaison, recopie, recherche, tris...),
- des allocateurs mémoire,
- des foncteurs

La bibliothèque BOOST contient beaucoup, beaucoup de choses (conteneurs, graphes, maths, entrées/sorties, fichiers...).

Template: STL

Pour comprendre et utiliser la STL, il faut savoir qu'il est possible de déclarer un alias de type dans une classe:

```
template<class T>
class Entier{
  public:
  typedef T type;
};

Entier<int>::type i;
```

Ceci fait que la variable `i` est de type **int**

Template: STL

Parmis les conteneurs les plus connus, il y a les vecteurs, ceux-ci répondent au concept de “Conteneur à Accès Quelconque” et “Sequence avec insertion en fin”. Ces concepts définissent un certain nombre des propriétés: compléxité, nom de méthodes, type interne...

```
#include <vector>

std::vector<int> v;
v.push_back(0);
v.push_back(1);
v[0]=2;
for(std::vector<int>::iterator i=v.begin();
          i!=v.end();
          i++)
   std::cout<<"␣"<<*i;
```

Cet exemple utilise le type interne `iterator` qui correspond au type implémentant le concept d’itérateur pour la classe vecteur.

Template: STL

Les itérateurs ont été conçus pour fonctionner comme les pointeurs: on utilise `++` et `--` pour se déplacer, `*` et `->` pour utiliser la valeur représentée par l'itérateur.

Voici un exemple permettant de trier un vecteur:

```
#include <vector>
vector<int> v;
v.push_back(0); v.push_back(5);  v.push_back(2);
std::sort(v.begin(),v.end());
```

L'algorithme `std::sort` effectue un trie des données contenues par la séquence des deux itérateurs.

Ainsi, cet algorithme pourra être utiliser avec de nombreux autre conteneurs de la STL qui disposent des itérateurs et même des tableaux:

```
int t[3] = {0, 5, 2};
std::sort(t,t+3);
```

STL

La STL repose sur la notion de concept.

Un concept est un peu comme une "interface" mais qui n'a d'existence que sur le papier.

Une concept est un ensemble de méthodes, types internes, propriétés que doit posséder une classe répondant au concept.

Exemple de concept: "container"

- `X::value_type` type contenu
- `X::iterator` type à utiliser pour parcourir ("input iterator")
- `X::const_iterator` type pour parcourir sans modifier
- `X::reference` type de la référence vers le type contenu
- `X::const_reference` référence constante
- `X::pointer` pointeur vers le type contenu
- `X::difference_type` différence entre deux iterateurs
- `X::size_type`

STL

La STL propose un ensemble de classes paramétrées répondant à un ou plusieurs concepts :

- Séquences : `vector`, `deque`, `list`, `slist`, `bit_vector`
- Conteneurs associatifs :`set`, `map`, `multiset`, `multimap`, `hash_set`, `hash_map`, `hash_multiset`, `hash_multimap`, `hash`
- Les chaînes de caractères : `Character Traits`, `char_traits`, `basic_string`

Ainsi qu'un ensemble d'algorithmes :

- `for_each`, `find`, `find_if`, `adjacent_find`, `find_first_of`, `count`, `count_if`, `mismatch`, `equal`, `search`, `search_n`, `find_end`
- remplacement dans une séquence ...
- tris, mélange, rotation, renversement, ...
- recherche d'élément, min, max ...

Template: et metaprogrammation

La metaprogrammation consiste à faire s'executer un programme par le compilateur au moment de la compilation et donc de façon statique.

Un exemple classique de metaprogrammation est un calcul mathématique simple:

```
template<int a, int b>
struct Addition{
  static const int resultat=a+b;
};

std::cout<< Addition<2,3>::resultat <<std::endl;
```

Template: et metaprogrammation

et un peu plus compliqué:

```
template<int a, int b>
struct Puissance{
  static const int resultat=a*Puissance<a,b-1>::resultat;
};

template<int a>
struct Puissance<a,0>{
  static const int resultat=1;
};

std::cout<< Puissance<2,3>::resultat <<std::endl;
```

Template: et metaprogrammation

La metaprogrammation permet de faire des choses très puissantes telles que la composition de types. Cependant, elle reste souvent réservée à l'implémentation de bibliothèques complexe et est difficile à maîtriser.

Elle repose sur les concepts suivant:

- les templates,
- la spécialisation,
- les types internes,
- les attributs statiques et constants

Template: et metaprogrammation

On peut ainsi écrire un “language” sur les types, avec des branchements conditionnels:

```
template<bool C, class T, class E>
struct IfThenElse{
  typedef T result;
};

template<class T,class E>
struct IfThenElse<false,T,E>{
  typedef E result;
};

IfThenElse<true,int,char>::result i;
```

Template: et metaprogrammation

Comparaison de types:

```
template<class T,class U>
struct Equal{
  static bool result=false;
};
template<class T>
struct Equal<T,T>{
  static bool result=true;
}

bool b=Equal<int,size_t>::result;

IfThenElse<Equal<int,ssize_t>::result,
           int,
           long>::result i;
```

Template: et metaprogrammation

Ce type de metaprogrammation sert pour l'optimisation:

```
template<class T>
struct IsPointer{
  static bool result=false;
};
template<class T>
struct IsPointer<T *>{
  static bool result=true;
};
template<class T>
class MaClasse{
  typedef IfThenElse<IsPointer<T>::result,list<T>,vector<T>> icontainer;
  icontainer _data;
  ...
};
```

Template: et metaprogrammation

On peut aussi utiliser les fonctions templates pour inférer des types et “cacher” les templates

```
struct Interface {
  virtual ~Interface(){}
  ....
};

template<class T>
class MaClasse : public Interface{
  typedef IfThenElse<IsPointer<T>::result,list<T>,vector<T>> icontainer;
  icontainer _data;
  ...
  public:
  MaClasse(const T &t){
    ....
  }
  ...
};

template<class U>
static MaClasse<U> *getClasse(const U &value){
  return new MaClasse<U>(value);
}

int main(){
  Interface *i=getClasse(0);
}
```

Template: et metaprogrammation

La métaprogrammation permet de faire des API simplifiées:

```
struct RandomContainer{};
struct ReversibleContainer{};

template<class T,class Type>
struct Container{
  typedef vector<T> result;
}

template<class T>
struct Container<T,ReversibleContainer>{
  typedef list<T> result;
}

Container<int,RandomContainer>::result container;
```

Plan

Synthèse

Synhèse

Le langage C++ dispose de nombreux concepts:

- l'encapsulation (visibilité)
- les références
- les opérateurs
- l'héritage simple et multiple
- les fonctions vituelles
- la programmation générique

Egalement, il laisse au programmeur la gestion de la mémoire.

Synthèse

Programmer efficacement en C++, c'est répondre à deux exigences:

- Ecrire du code lisible, dont le fonctionnement est clair.
- Viser la performance: en temps et en mémoire.

Ecrire du code lisible

Ecrire du code lisible, dont le fonctionnement est clair implique:

- Ne pas utiliser les opérateurs n'importe comment.
- Ne pas détourner les opérateurs au delà de leurs sens premier.
- Limiter l'usage de la programmation générique à des parties de code isolée.
- Pas de surchage avec l'héritage
- Pas de valeur par défaut avec l'héritage

Il n'est pas vraiment possible d'interdire la copie (par construction ou par affectation) aussi il faut la gérer au mieux (utilisation de la visibilité).

Ecrire du code lisible

D'une manière générale, on réservera le polymorphisme aux classes structurantes (architecture logicielle):

- Destructeur virtuel
- méthodes toutes virtuelles
- Heritage virtuel sur les *interfaces*
- Limiter l'usage de l'héritage multiple.
- pas d'opérateurs
- pas de surcharge

Ces classes servent pour la structuration générale du logiciel: en pratique, le nombre d'instances est réduit et l'utilisation (appel de méthodes) des instances est modéré.

Viser la performance

Pour les parties *calculatoires* (pixels d'une image, graphes, etc...) on utilise en priorité:

- Des classes sans méthodes virtuelles (si beaucoup d'instances)
- Pas d'héritage
- Les opérateurs si l'on veut profiter de code générique
- La programmation générique

Ce code doit être *isolé* du reste de l'application (api, bibliothèque).