# Programmation C++

## 2ème année Informatique

J. Allali
julien.allali@labri.fr

ENSEIRB

Prog. C++

# Plan

# Plan

# Plan

# Plan

# Plan

# Les classes: déclaration

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

**Exemple**

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

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 *a=new A();
```

ou automatique:

**allocation automatique**

```
A a;
```

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

# 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 accédées 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 methodes privés
public:
attributs et methodes publiques
protected:
attributs et methodes 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 nom 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 *a=new A();
struct A *a=new A();
A *a=new struct A();
struct A *a=new struct A();
```

## allocation automatique:

```
struct A { ... };
A a;
struct A a;
```

# Friend: définition

Le mot clé **friend** permet, dans une class A, 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

# Attributs et méthodes: définition

Une classe est un regroupement d'attributs, propriétés interne 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

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");
        }
};
```

# 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();
```

# 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();
```

# 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:
        void print(); /* méthode publique */
};
```

# Exemple de classe: 2

## Deux fichiers

Test.hpp:

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

Test.cpp:

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

# Exemple de classe: 2

## Deux fichiers

Test.hpp:

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

Test.cpp:

```
#include”Test.hpp”
voidTest::print(){
    printf(“Test\n”);
}
```

Exemple.cpp:

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

# Exemple de classe: 2

## Deux fichiers

Test.hpp:

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

Test.cpp:

```
#include”Test.hpp”
voidTest::print(){
    printf(“Test\n”);
}
```

Exemple.cpp:

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

# Exemple de classe: 2

## Deux fichiers

Test.hpp:

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

Test.cpp:

```
#include"Test.hpp"
voidTest::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.

# 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.

# 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 automatique

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

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

# Plan

# Le mot clé **this**

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

# Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source.
Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de* **const**.)

# Le mot clé **this**

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

# Opérateur de portée

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

Exemple:

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

# Plan

# Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...).

# Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisé pour structurer le code et pour éviter les problèmes de collisions

# Les espaces de nom

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

# Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisé 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 porté.

# 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 porté.

A.hpp

```
namespace tec {
    class A {
    public:
        void 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 porté.

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 porté.

A.hpp

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

A.cpp

```
#include "A.hpp"
void tec::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 porté.

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'espace de nom différents pouvant être défini dans un fichier ni sur le niveau d'imbrication de ces espaces. Cependant la taille des noms de fonctions est limité et les espaces de nom font partis du nom de la fonction.

# 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 porté.

A.hpp

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

A.cpp

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

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 porté

# 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!
}
```

**using** sur un objet

# 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!
}
```

Le **using** n'est actif que sur les instructions qui suivent...

# 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!
}
```

Le **using** dans un espace de nom est local à cet espace

# 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
}
```

Le **using** dans une fonction est local à cette fonction

# 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
}
```

Le **using** peut se faire sur l'ensemble des objets d'un espace

# 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;
}
```

# 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.

# 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

# 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.

# 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 porte est une méthode sans valeur de retour de même nom que la classe.

# 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 porte est une méthode sans valeur de retour de même nom que la classe.

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

# 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 porte 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 attribut;
    public:
        Exemple(){ // Constructeur par défaut
            attribut=0;
        }
        Exemple(double d){ // Constructeur
            attribut=d;
        }
};
```

# Constructeur: exemple 2 fichiers

Exemple.hpp

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

Exemple.cpp

```
#include"Exemple.hpp"
Exemple::Exemple(){
    attribut=0;
}
Exemple::Exemple(double d){
    attributs=d;
}
```

# Constructeur: initialisation des attributs

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

## Initialisation:

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

# Constructeur: initialisation des attributs

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

Initialisation:

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

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

# Constructeur: initialisation des attributs

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

Initialisation:

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

On peut effectuer des opérations

# Constructeur: initialisation des attributs

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

Initialisation:

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

On peut appeler des fonctions

# Constructeur: initialisation des attributs

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

Initialisation:

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

**On peut appeler des methodes!**

# 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:

# Constructeur: et instanciation

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

## Parenthésée

```
A a;
A a(1);
A a('c');
A a(); // Erreur:
       //pas d'intanciation!
```

## Affectation

```
A a;
A a=1;
A a='c';
```

# Constructeur: et instanciation

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

### Parenthésée

```
A a;
A a(1);
A a('c');
A a(); // Erreur:
       //pas d'intanciation!
```

### Affectation

```
A a;
A a=1;
A a='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.

# Constructeur: et instanciation

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

## Exemple

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

# Constructeur: et instanciation

Cette syntaxe est valable pour les types primitifs:

# Constructeur: et instanciation

Cette syntaxe est valable pour les types primitifs:

### Parenthésée

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

### Affectation

```
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):

# 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.
```

# 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 utiliser lors du passage d'un argument à une fonction ou lors d'une levée d'exception.

# Constructeur: et instanciation

Source:

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


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

# Constructeur: et instanciation

Source:

```
void f(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
```

# 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.

# 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);
...
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);

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

# Constructeur: conversion

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

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

void f(B);

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);
...
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.

# 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.

# 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 a;
f(a)
```

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

# 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 a);
};
```

# 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 a);
};
```

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

# 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 a);
};
```

Le problème est que pour instancier la variable locale au constructeur il nous faut le constructeur par recopie!
La solution repose sur l'utilisation des références:

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

# Références

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

# 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).

# 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;
}
```

# Références

Exemple:

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

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]
```

# Références

Exemple:

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

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.

# const: définition

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

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

# const: définition

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

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

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

# const: définition

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

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

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

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

# const: définition

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

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

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

# const: définition

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

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

```
const int *i=0; // int *const i=0;
i=new int[10];
i[0]=5; // erreur!
int const*p=new int[10];
p=NULL; // erreur!
p[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: méthodes

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

# 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 m() ;
};
```

```
void A::m() {
  //  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 a;
const A & b=a;

a.m(); // appel A::m()
b.m(); // appel A::m() 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 a;
const A & b=a;

a.m(); // appel A::m()
b.m(); // appel A::m() const
```

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.

# 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 a;
const A & b=a;

a.m(); // appel A::m()
b.m(); // appel A::m() const
```

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.

# 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 a;
const A & b=a;

a.m(); // appel A::m()
b.m(); // appel A::m() const
```

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.

# Constructeur: recopie (suite)

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

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

# Constructeur: recopie (suite)

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

```
class A{
public:
   A(const A &); // 1
   A(A &);  // 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.

# Constructeur: recopie (suite)

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

```
class A{
public:
   A(const A &); // 1
   A(A &);  // 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 tout les constructeurs par recopie des attributs sont de type 1, sinon il sera de type 2.

# Constructeur: par recopie

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

# 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 écrire un constructeur par recopie.

# Constructeur: proctected 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

# Constructeur: proctected 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

# Destructeur: définition

Le destructeur est une méthode de la classe 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()
```

# Destructeur: définition

Le destructeur est une méthode de la classe 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.

# Plan

# 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).

# 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.

# 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';
};
```

# 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 (equivalent à une macro).

# Plan

# 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.

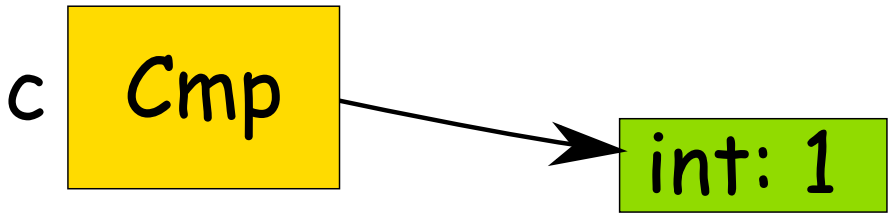
# Une classe compteur

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.

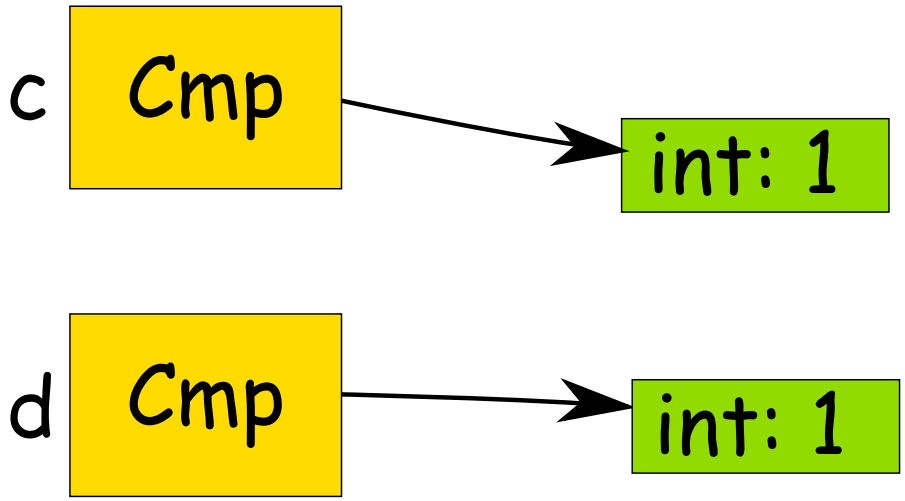
# Compteur: Exemple1

Cmp c;



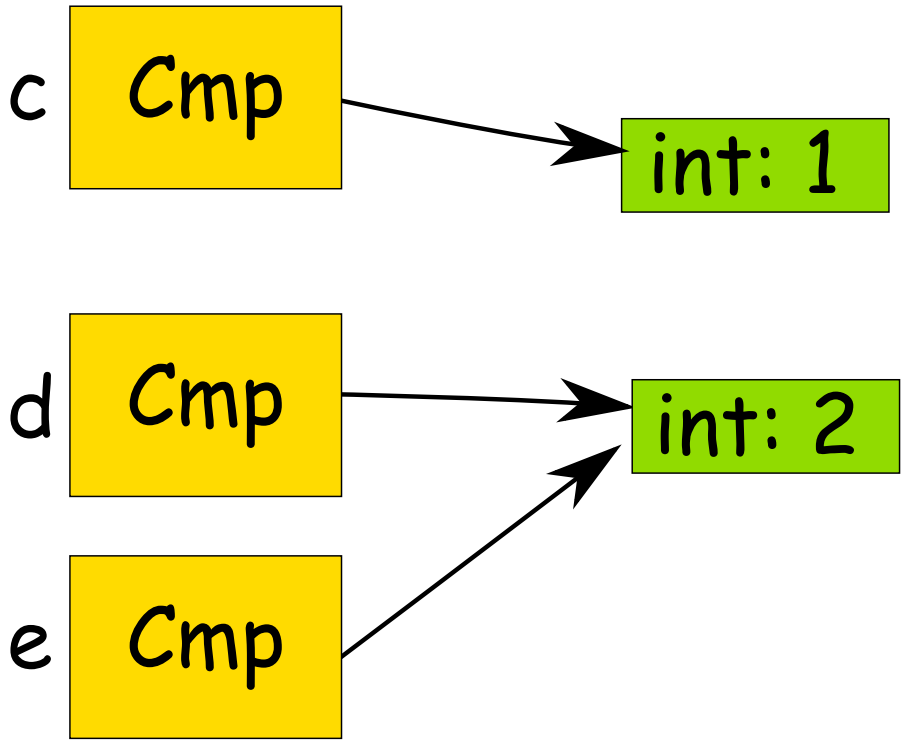
# Compteur: Exemple1

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



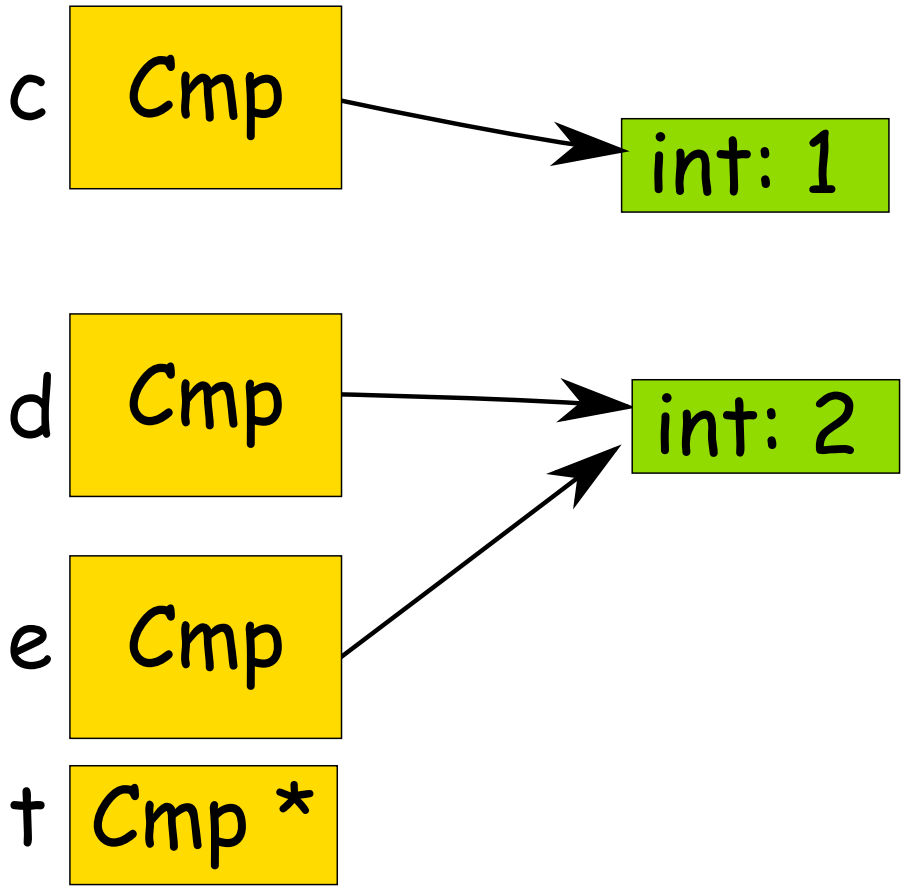
# Compteur: Exemple1

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



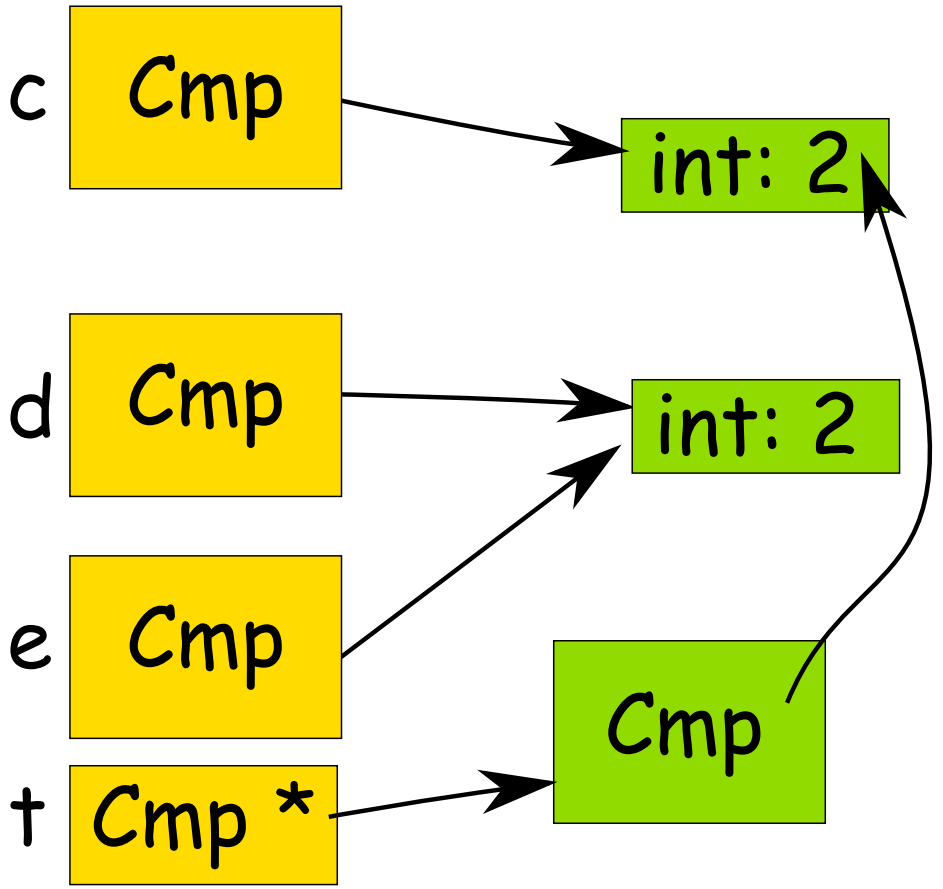
# Compteur: Exemple1

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



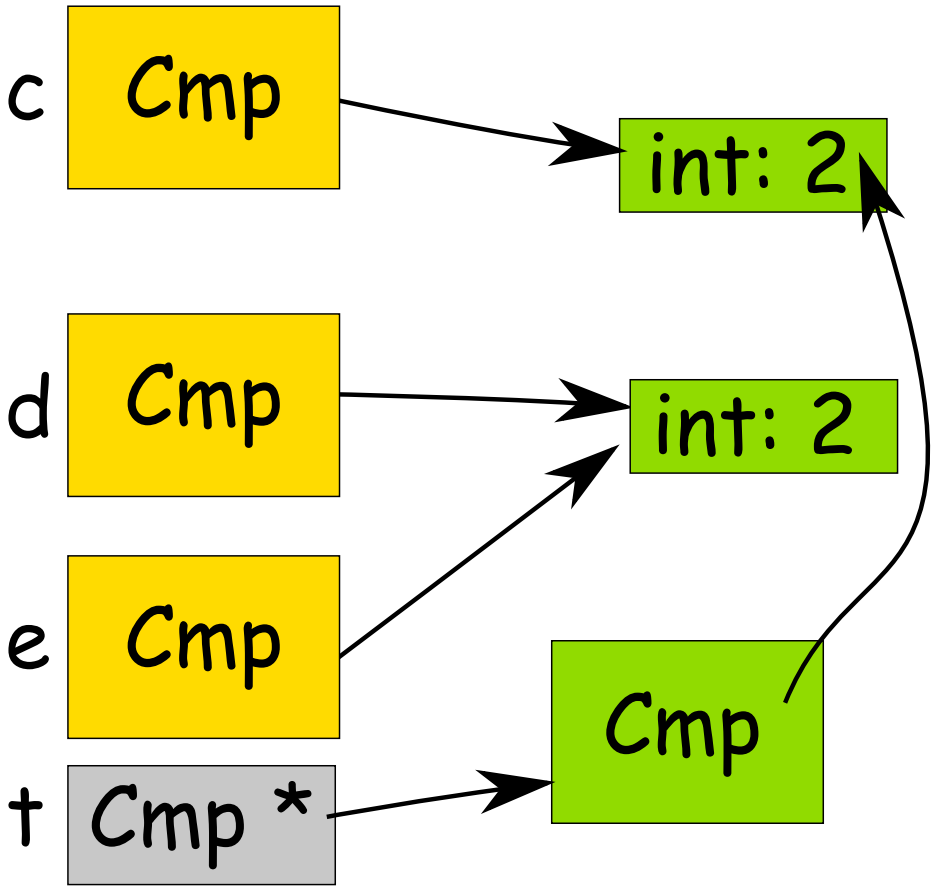
# Compteur: Exemple1

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



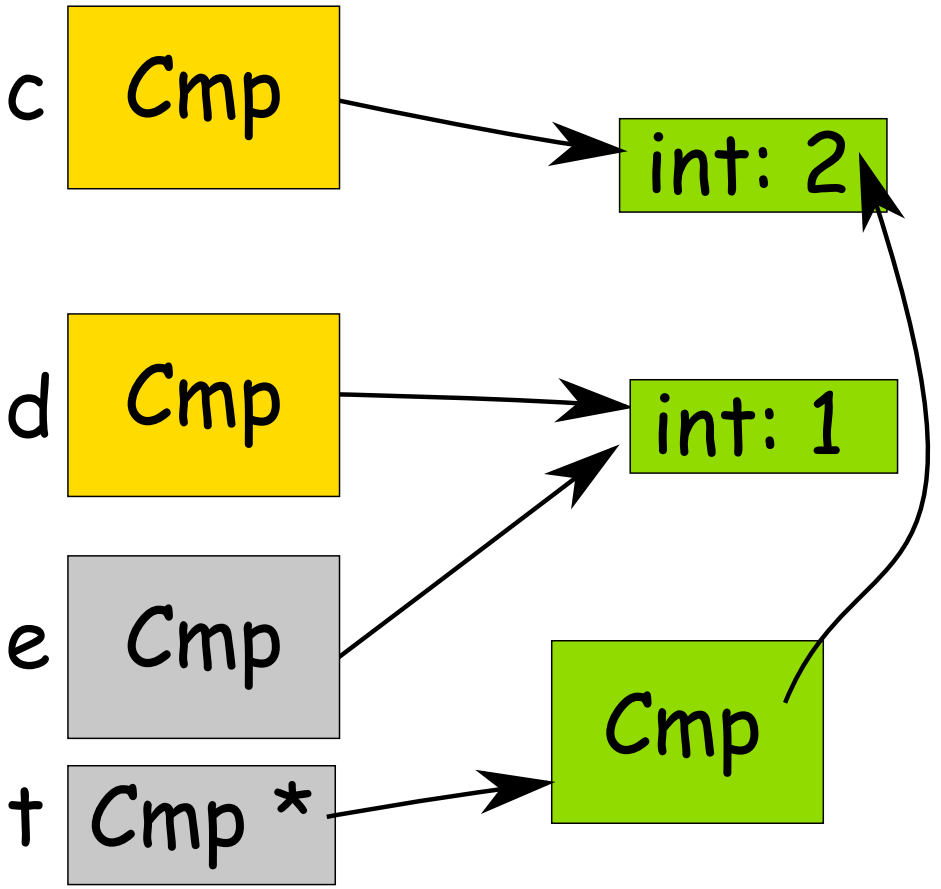
# Compteur: Exemple1

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



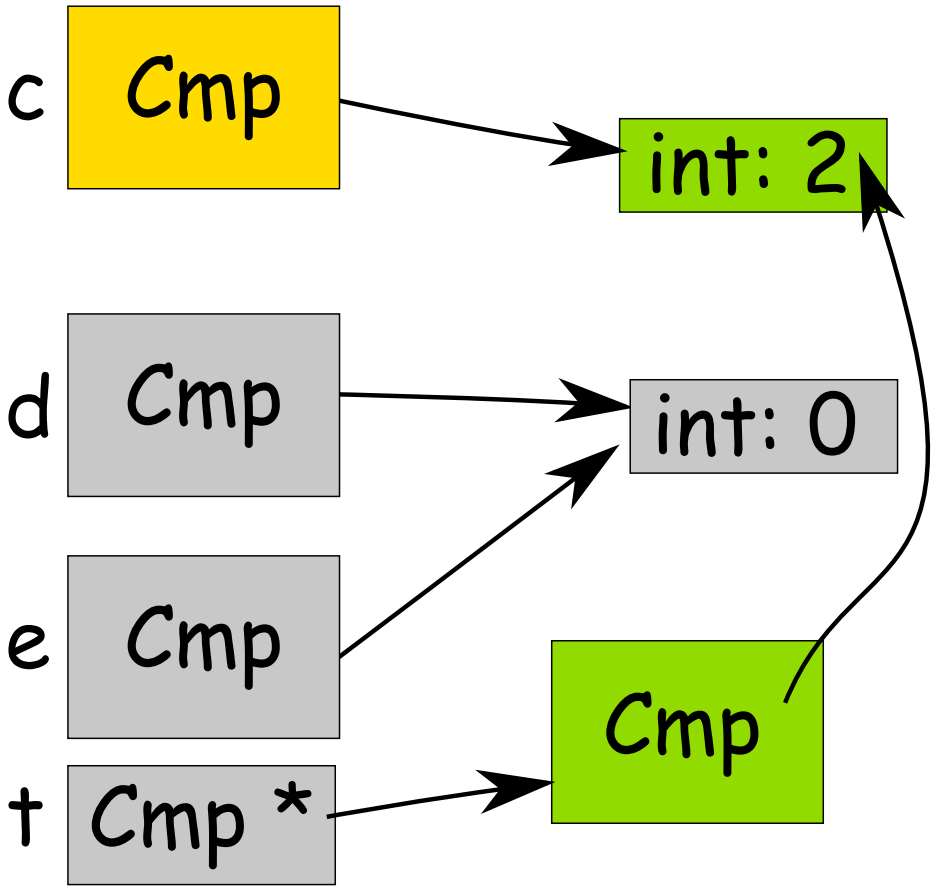
# Compteur: Exemple1

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



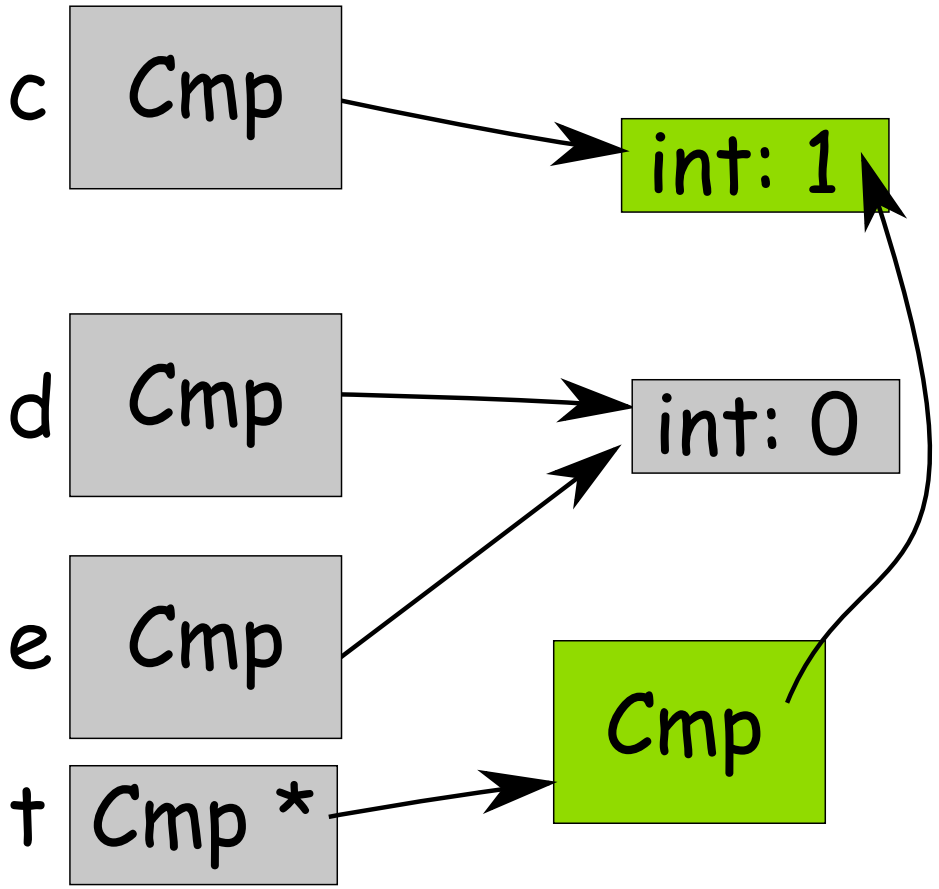
# Compteur: Exemple1

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



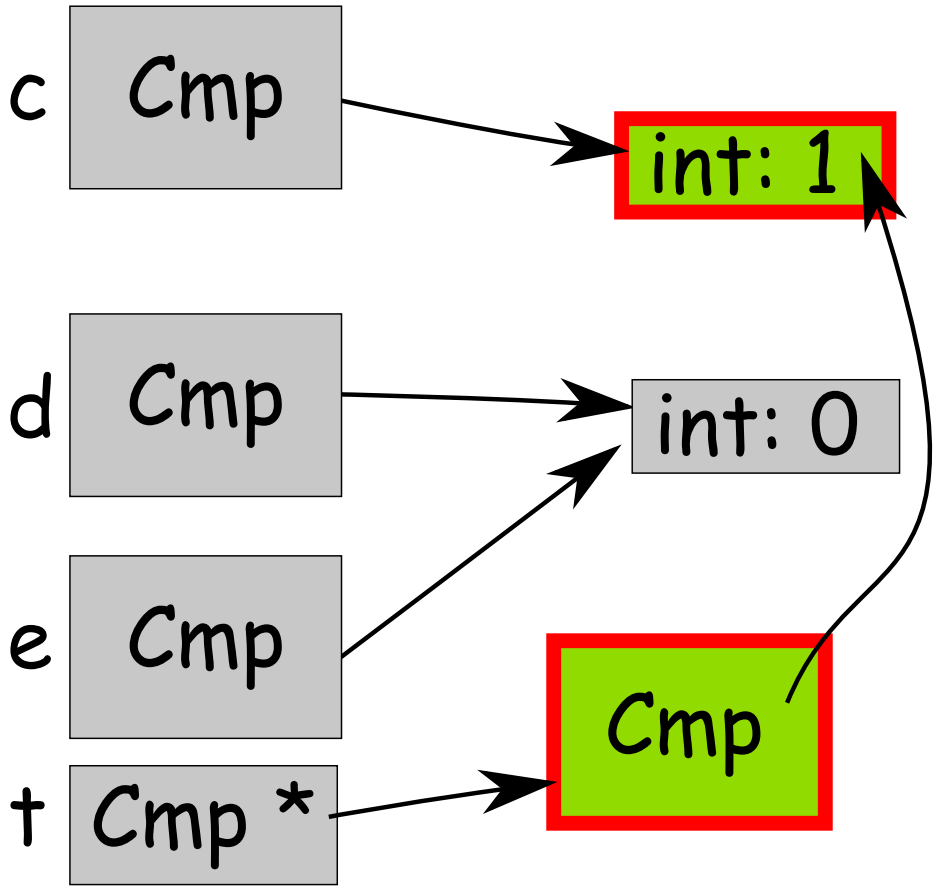
# Compteur: Exemple1

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



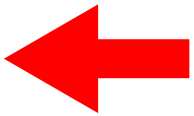
# Compteur: Exemple1

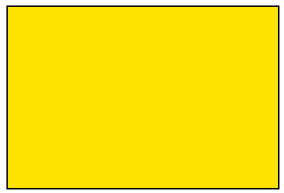
```
Cmp c;
Cmp d;
Cmp e=d;
Cmp *t;
t=new Cmp(c);
...
}
```



# Compteur: Exemple2

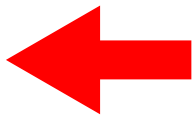
```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```

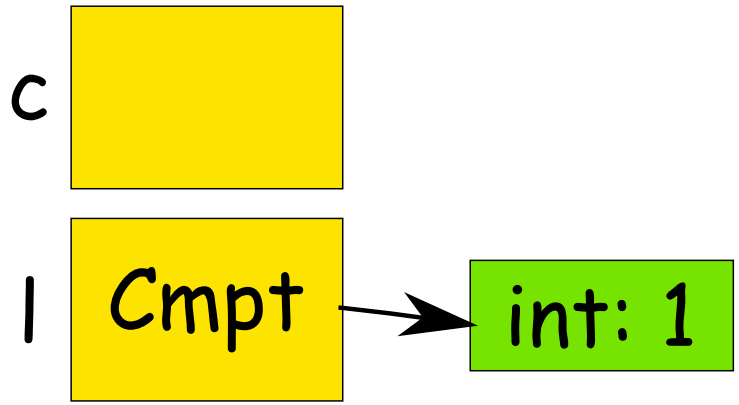


c 

# Compteur: Exemple2

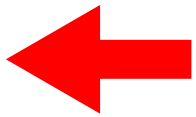
```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```

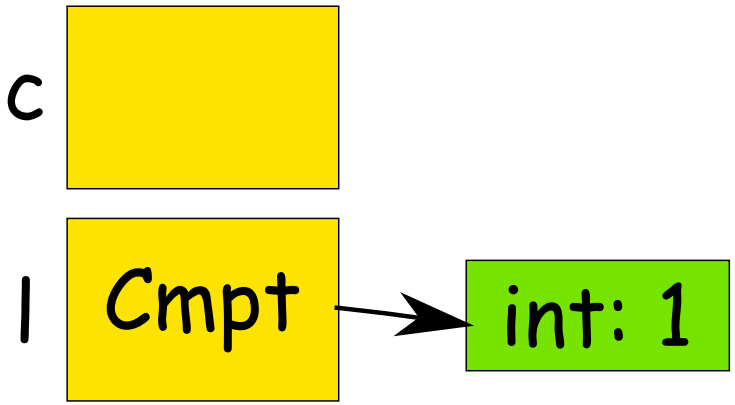




# Compteur: Exemple2

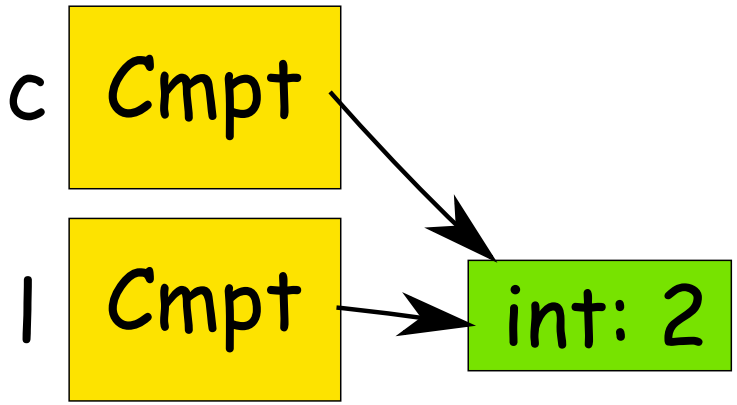
```
Cmpt f(){
Cmpt l;
return l;
}

Cmpt c=f();
```





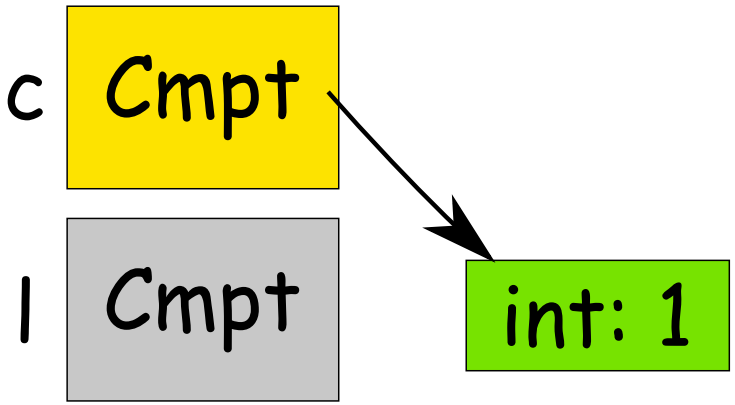
# Compteur: Exemple2

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

Cmpt c=f();
```



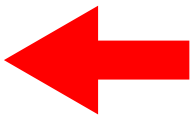
# Compteur: Exemple2

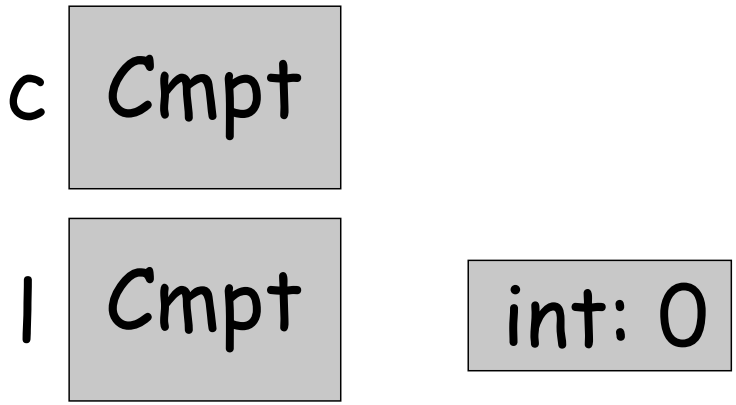
```
Cmpt f(){
Cmpt l;
return l;
}

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
}
```

# Problème:

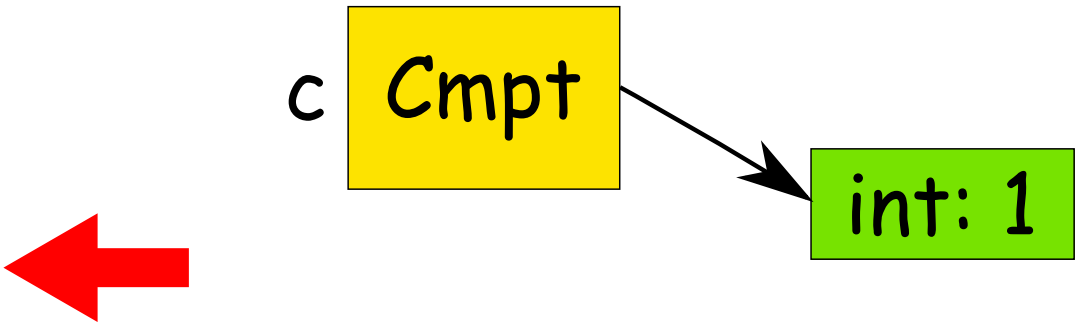
Examinons le source suivant:

```
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!

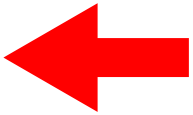
# Problème:

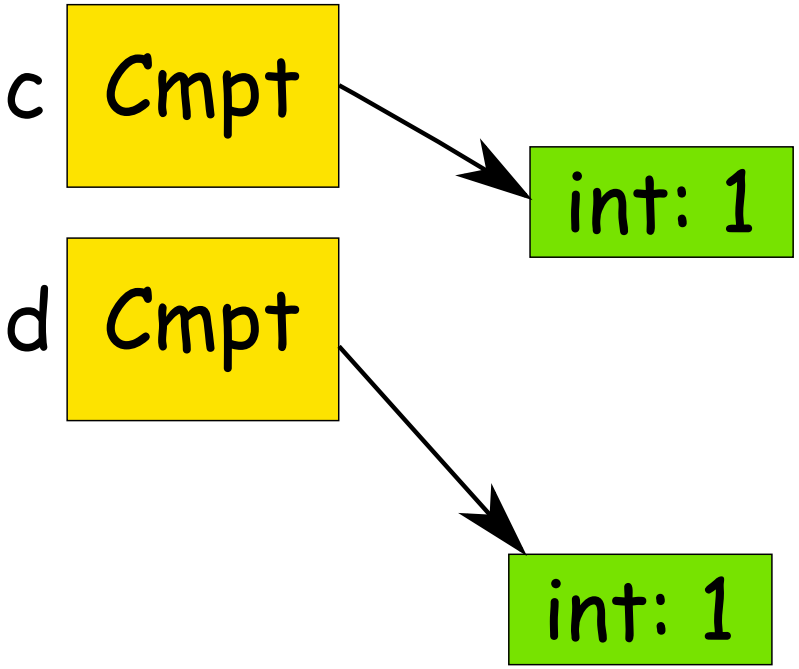
```
int main(){
  Cmpt c;
  Cmpt d;

  d=c;
}
```



# Problème:

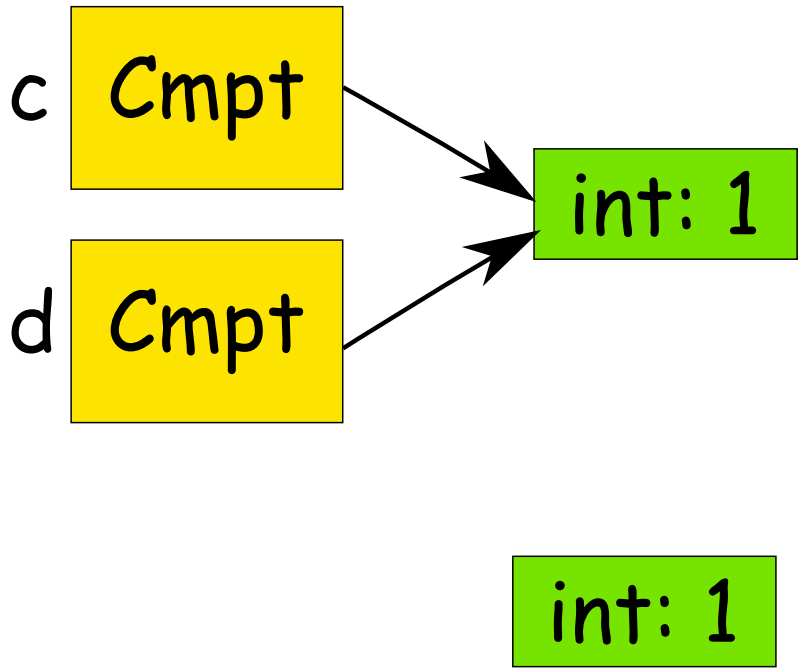
```
int main(){
  Cmpt c;
  Cmpt d;

  d=c;
}
```





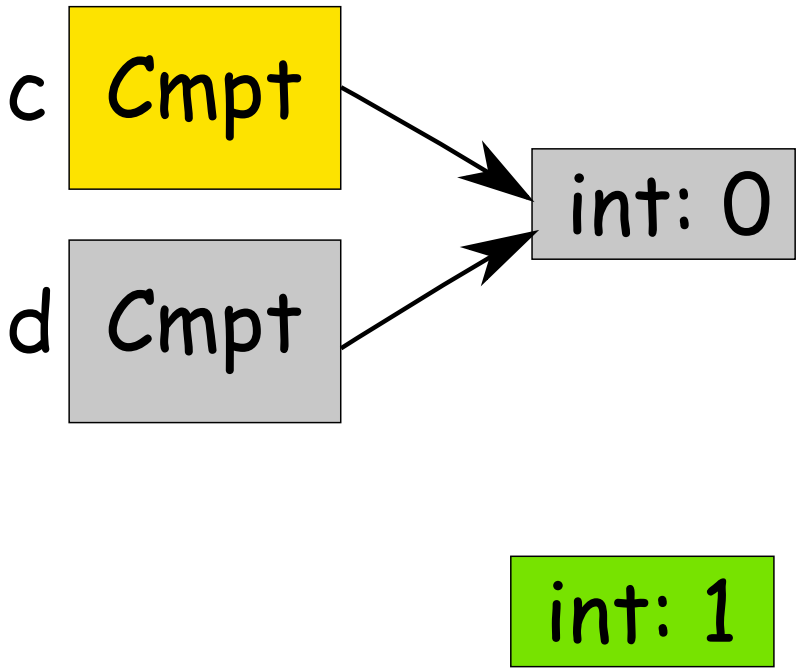
# Problème:

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

  d=c;
}
```



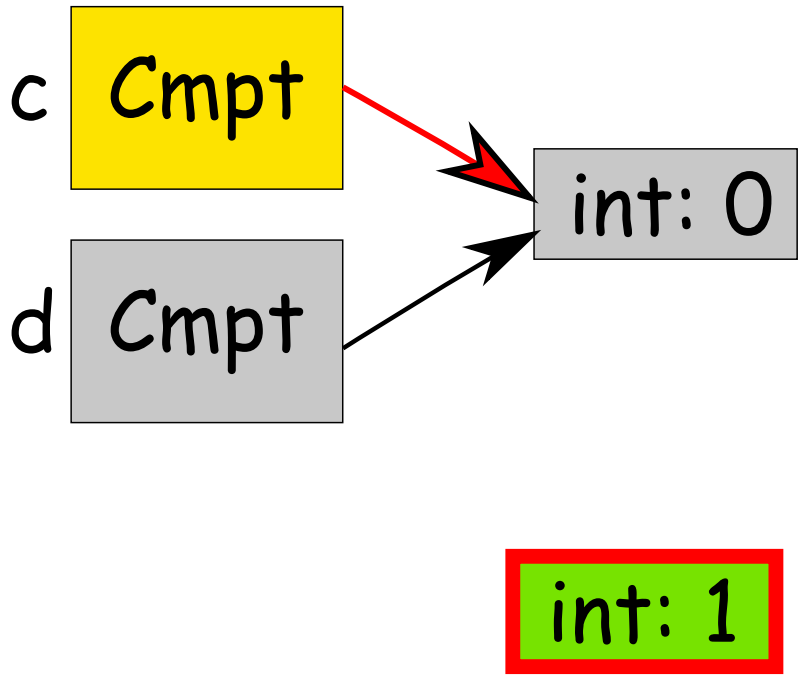
# Problème:

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

  d=c;
}
```



# Problème:

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

  d=c;
}
```



# Plan

# Opérateurs: définition

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

# Opérateurs: définition

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

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

# 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`

# 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`.

# 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:

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

# Opérateurs: liste

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

## Unaire:

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

## Binaire:

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

<     <=     >     >=

!=    ==     &&    ||

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

->    ->*
```

# Opérateurs: liste

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

## Unaire:

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

## spéciaux:

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

## Binaire:

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

<     <=     >     >=

!=    ==     &&    ||

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

->    ->*
```

# Opérateur: 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.

# Opérateur: 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 operateur permettant de faire l'addition d'un `A` avec un `B`,

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

ne sera pas résolu pas cet opérateur.

# Opérateur: 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 operateur 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 $\alpha$ s'écrira sous le nom d'une fonction ou méthode non statique de nom `operator` $\alpha$.

# Opérateurs: syntaxe

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

Le type de retour de cette fonction est libre.

# Opérateurs: syntaxe

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

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.

# Opérateurs: syntaxe

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

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 appeler les opérateurs comme des fonctions où des méthodes:

fonction:

```
class Dix{
};

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

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

# Opérateurs: syntaxe

On peut appeler les opérateurs comme des fonctions où des méthodes:

méthode:

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

int main(){
  Dix d;
  int douze=d.operator+(2);
  2.operator+(d); //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 a,b;
  a+b*a; // Erreur !
  (a+b)*a; // ok
}
```

# Opérateurs: unaires

On écrira les opérateurs unaires dans les classes:

```
class Entier{
  int v;
  public:

  Entier operator-() const {
    return Entier(-v);
  }
};
```

# Opérateurs: unaires

On écrira les opérateurs unaires dans les classes:

```
class Entier{
  int v;
  public:

  Entier operator-() const {
    return Entier(-v);
  }
};
```

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 Entier{
  int v;
  public:
  Entier operator+(const Entier &e) const{
    return Entier(v+e.v);
  }
};
```

# Opérateurs: binaires

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

```
class Entier{
  int v;
  ...
};
Double operator+(const Entier &e,const Double &d){
  return Double(e.value()+d.value());
}
Double operator+(const Double &d,const Entier &e){
  return e+d;
}
```

# Opérateurs: binaires

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

```
class Entier{
  int v;
  ...
};
Double operator+(const Entier &e,const Double &d){
  return Double(e.value()+d.value());
}
Double operator+(const Double &d,const Entier &e){
  return e+d;
}
```

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.

# 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.

# 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

# Opérateurs: iostream

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

- `ios` : bases pour les entrées/sorties

# 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

# 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

# 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

# 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

# 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

# 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

# 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`

# 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
```

# 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)
```

# 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 << ?

# 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 ?

# 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 $2^i$ à du sens.

# Opérateur: 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`

# Opérateur: 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?

# Opérateur: 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 &, ... )
```

# Opérateur: 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 Entier{ int v; };
int main(){
  Entier e;
  std::cerr<<"debug"<<e;
}
```

# classe et iostream

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

```
class Entier{ int v; };
int main(){
  Entier e;
  std::cerr<<"debug"<<e;
}
```

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> >&)(&
  ((const char*)"debug")) << e'
```

suivi d'une bonne cinquantaine de lignes !

# Solution

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

```
class Entier{
  int v;
};

std::ostream &operator<<(std::ostream &stream,Entier e){
  stream<<e.v;
  return stream;
}
```

Combien d'erreurs dans ce code ?

# Solution

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

```
class Entier{
  int v;
};

std::ostream &operator<<(std::ostream &stream,Entier e){
  stream<<e.v;
  return stream;
}
```

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

- e doit être une référence constante
- e.v n'est pas accessible

# Solution

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

```
class Entier{
  int v;
};

std::ostream &operator<<(std::ostream &stream,Entier e){
  stream<<e.v;
  return stream;
}
```

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

- e doit être une référence constante
- e.v n'est pas accessible

solutions:

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

# Solution

```
class Entier{
  int v;
public:
  int value() const {return v;}
};

std::ostream &operator<<(std::ostream &stream,
                         const Entier &e){
  stream<<e.value();
  return stream;
}
```

# Solution

```
class Entier{
  int v;
public:
  friend std::ostream &operator<<(
              std::ostream &stream,
              const Entier &e);
};
std::ostream &operator<<(std::ostream &stream,
                         const Entier &e){
  stream<<e.v;
  return stream;
}
```

# Solution

```
class Entier{
  int v;
public:
  friend std::ostream &operator<<(
              std::ostream &stream,
              const Entier &e);
};
std::ostream &operator<<(std::ostream &stream,
                         const Entier &e){
  stream<<e.v;
  return stream;
}
```

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

# Solution

```
class Entier{
  int v;
public:
  friend std::ostream &operator<<(
              std::ostream &stream,
              const Entier &e);
};
std::ostream &operator<<(std::ostream &stream,
                         const Entier &e){
  stream<<e.v;
  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 en faveur des accesseurs !

# Opérateur: d'affectation

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

```
class Entier{
  int v;
public:
  const Entier &operator=(int i){ v=i ;}
};
...
Entier e;
e=1;
Entier f=1; // Erreur !
```

# 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.

# Compteur

Revenons sur notre exemple de classe Compteur et ajoutons un opérateur d'affectation...

*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*

# 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.

# 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:

# 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")){...}
```

# 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.*

# 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).

# 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;
  }
};
```

# 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.

# 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.

# Opérateurs: de conversion

Une utilisation intéressante de l'utilisation 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.

# Opérateurs: autres

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

L'opérateur **new** est utilisé pour faire du placement d'objet en mémoire. Il permet d'écrire ces propres allocateurs ou encore de faire du "debug-age". Le nombre d'argument est variable mais le premier est obligatoirement de type `size_t`, le type de retour est obligatoirement **void**`*`.

# Plan

# Héritage: définition

L'héritage est un mécanisme permettant de construire un type `T` à par d'un autre type `Base`. Le type `T` se retrouve doté de 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`.

# Héritage: définition

L'héritage est un mécanisme permettant de construire un type `T` à par d'un autre type `Base`. Le type `T` se retrouve doté de 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'exterieur.

# Héritage: définition

L'héritage est un mécanisme permettant de construire un type `T` à par d'un autre type `Base`. Le type `T` se retrouve doté de 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'exterieur.

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;
};
```

# 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.

# 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.

# 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 permier paramètre caché (**this** dans la méthode).

# Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un permier 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.

# Héritage: appel de méthodes

En C++ l'appel de méthode doit être vu comme un appel de fonction avec passage d'un permier 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 porter pour désigner 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()`

# 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!

# 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.

# 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.
Le “mangling” permet la mise en oeuvre de la surcharge, des espaces de nom, des méthodes de classes...

# 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.

# 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`.

# 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`.

# 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 **protected**, les méthodes **public** et **protected** de `Base` sont **private** dans `T`.

# Héritage: public, protected et private

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

# Héritage: public, protected et private

On ne peut faire 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`.

# Héritage: public, protected et private

On ne peut faire 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`.

# Héritage: public, protected et private

On ne peut faire 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.

# 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).

# 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 exemle: 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`.

# 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 exemle: 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 implanter la pile.

# 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 exemle: 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 implanter la pile.

Si l'on souhaite que cet "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.

# 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.

# 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.

# 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 {}
};
```

# 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 {}
};
```

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).

# 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 {}
};
```

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