

Programmation orientée objet

Fonctionnement du
polymorphisme

Comment le polymorphisme fonctionne-t-il?

Soient les classes suivantes:

```
class A
{
public:
    A();
    virtual void f1();
    virtual void f2();
private:
    int attA_;
};
```

```
class B : public A
{
public:
    void f1() override;
    void f2() override;
    ...
private:
    ...
};

class C : public A
{
public:
    void f1() override;
    ...
private:
    ...
};
```

Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

Comment fait-on pour savoir quelle méthode appeler?

Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 0

Ici il faut appeler A::f1()

Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 0

Ici il faut appeler A::f2()

Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 1

Ici il faut appeler B::f1()

Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 1

Ici il faut appeler B::f2()

Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 2

Ici il faut appeler C::f1()

Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 2

Ici il faut appeler **A::f2()**,
puisque la méthode n'est
pas redéfinie dans **C**.

Comment le polymorphisme fonctionne-t-il?

- Normalement, avec l'héritage, on peut savoir dès la compilation quelle méthode doit être appelée
- Ici, ce n'est pas possible, puisqu'on ne peut pas toujours savoir quel est le type d'objet réellement pointé par le pointeur
- On a vu, dans l'exemple précédent, que le type peut changer lors de l'exécution

Comment le polymorphisme fonctionne-t-il?

- Quand un objet est une instance d'une classe contenant des méthodes virtuelles, cet objet, en plus d'avoir de l'espace alloué pour ses attributs, aura un pointeur à une table appelée *vtable*
- La table *vtable* indique, pour chaque fonction virtuelle, quelle fonction doit être appelée
- Ainsi, dans l'appel `objet1->f1 ()`, on n'a qu'à rechercher « f1 » dans la *vtable*, et exécuter la fonction spécifiée
- Tous les objets d'une même classe pointent vers la même *vtable*

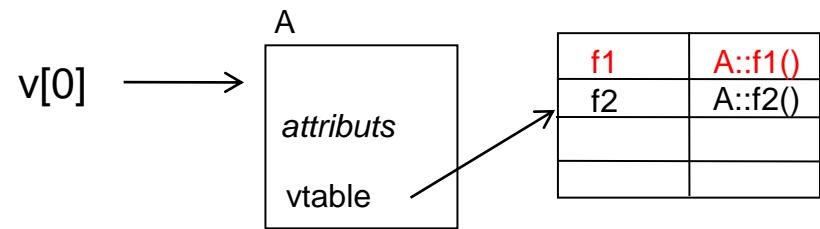
Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```



i = 0

Ici il faut appeler `A::f1()`

Comment le polymorphisme fonctionne-t-il?

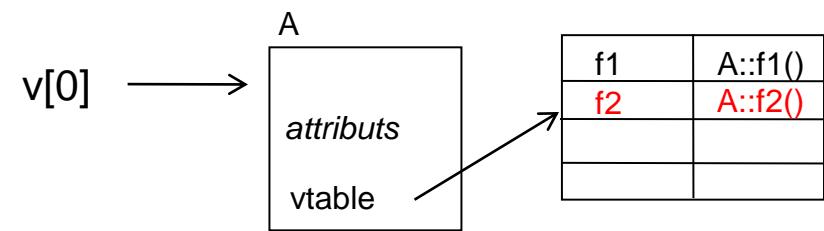
- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 0



Ici il faut appeler A::f2()

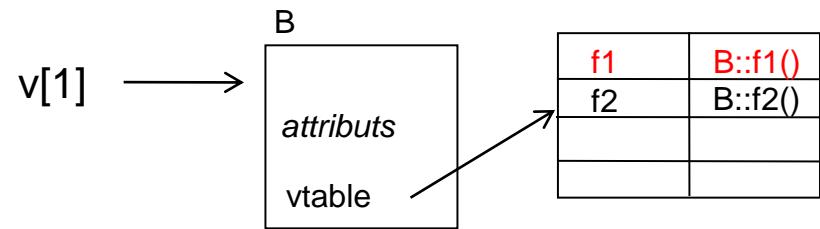
Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```



`i = 1`

Ici il faut appeler `B::f1()`

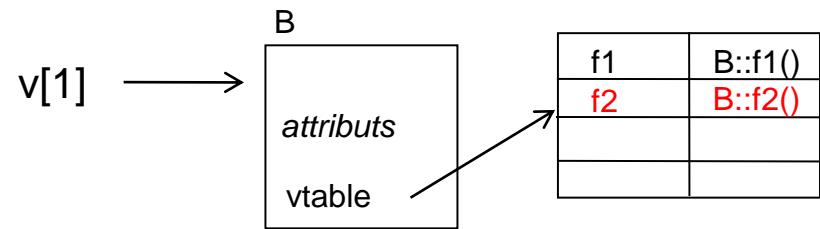
Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```



Ici il faut appeler **B::f2()**

Comment le polymorphisme fonctionne-t-il?

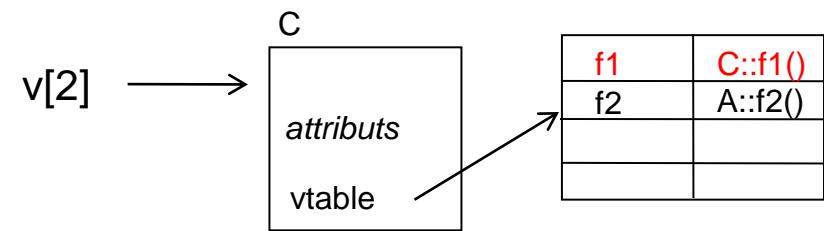
- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 2



Ici il faut appeler C::f1()

Comment le polymorphisme fonctionne-t-il?

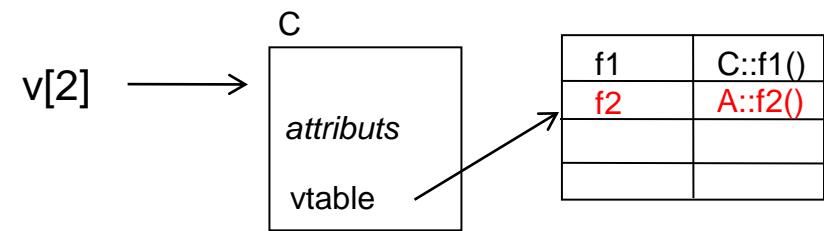
- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i){
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 2



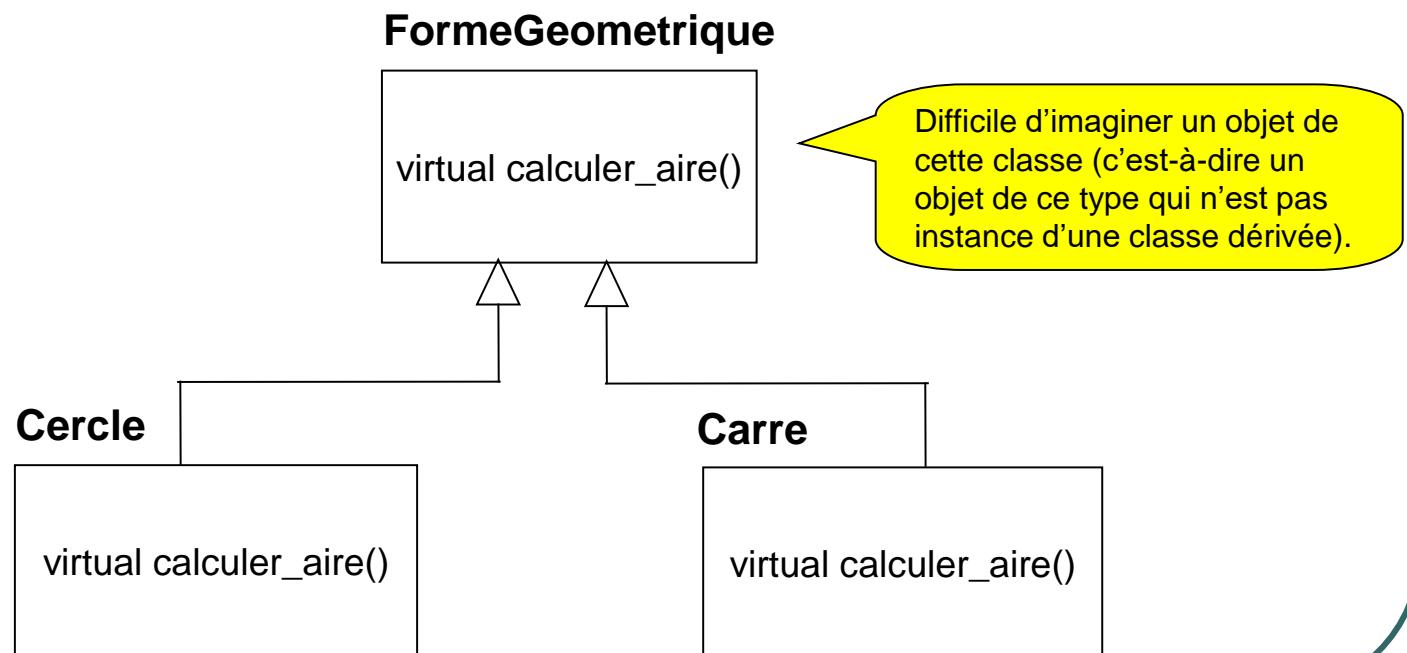
Ici il faut appeler **A::f2()**,
puisque la méthode n'est
pas redéfinie dans **C**

Programmation orientée objet

Classes abstraites

Classes abstraites

- Soit la hiérarchie de classe suivante:



Classes abstraites (suite)

- La classe **FormeGeometrique** est une classe abstraite
- Aucun objet ne peut appartenir directement à cette classe
- Un objet ne peut appartenir qu'à une classe dérivée
- En fait, la classe **FormeGeometrique** ne sert qu'à établir les méthodes qui seront héritées et certains attributs qui seront partagés par toutes les classes dérivées

Classes abstraites (suite)

- En C++, pour définir une classe abstraite, il suffit d'y déclarer **au moins une** fonction virtuelle pure
- Pour déclarer une fonction virtuelle pure, il suffit d'ajouter « = 0 » après la déclaration d'une fonction virtuelle

Voici par exemple comment déclarer virtuelle pure la fonction **addShare()** de la classe

StockExchangeEntity :

```
virtual void addShare() = 0;
```

- Si une classe contient une fonction virtuelle pure, il sera interdit de déclarer un objet de cette classe

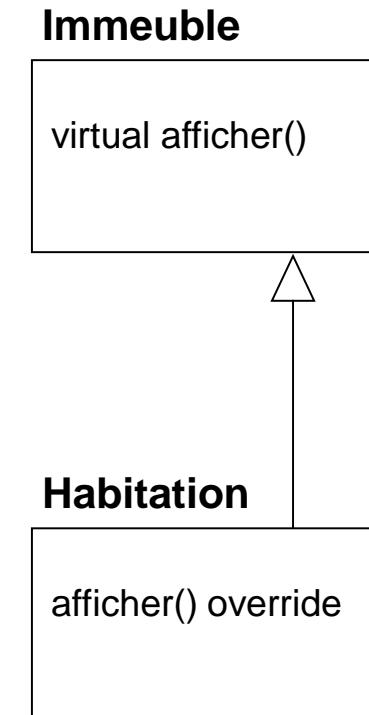
Programmation orientée objet

Types de conversion

Définition des classes Immeuble et Habitation

```
class Immeuble {  
public:  
    virtual void afficher();  
};
```

```
class Habitation : public Immeuble {  
public:  
    void afficher() override;  
};
```



Upcasting

- Un objet de la classe dérivée peut être converti implicitement en un objet de la classe de base.
- Dans ce cas on parle de **upcasting**; on monte dans la hiérarchie de classe
- L'**upcasting** est aussi faisable dans le cas d'un pointeur ou d'une référence
- Ce type de conversion se fait naturellement puisque la classe dérivée est un objet de la classe de base
- Il y a seulement perte de spécificité

Upcasting - Objet

```
void tester(Immeuble unImmeuble) {  
    unImmeuble.afficher();  
}  
  
int main() {  
    Immeuble monImmeuble;  
    Habitation monHabitation;  
    monImmeuble = monHabitation;  
    tester(monHabitation);  
}
```

Upcasting – Référence

```
void tester(Immeuble& unImmeuble) {  
    unImmeuble.afficher();  
}  
  
int main() {  
    Habitation monHabitation;  
    Immeuble& monImmeuble = monHabitation;  
    tester(monHabitation);  
}
```

Upcasting - Pointeur

```
void tester(Immeuble* ptr){  
    ptr->afficher();  
}  
  
int main() {  
    unique_ptr<Habitation> ptrHabitation =  
        make_unique<Habitation>();  
    unique_ptr<Immeuble> ptrImmeuble =  
        make_unique<Habitation>();  
    tester(ptrHabitation.get());  
}
```

Downcasting

- On parle de **downcasting** quand on convertit un objet, un pointeur ou une référence de la classe de base vers un objet, un pointeur ou une référence de la classe dérivée; on descend dans la hiérarchie de classe
- Ce type de conversion est plus délicat à gérer puisqu'un objet de la classe de base ne connaît pas les spécificités de la classe dérivée

Downcasting - Objet

- Le **downcasting** d'un objet n'est pas autorisé par défaut, il faut créer un constructeur de la classe dérivée qui accepte un objet de la classe de base:

```
class Habitation : public Immeuble {  
public:  
    Habitation();  
    Habitation(const Immeuble& immeuble);  
    void afficher() override;  
};
```

Downcasting - Objet

```
void tester(Habitation uneHabitation) {  
    uneHabitation.afficher();  
}  
  
int main() {  
    Immeuble monImmeuble;  
    tester(monImmeuble);  
}
```

Downcasting – Référence & pointeur

- Le **downcasting** d'un pointeur ou d'une référence n'est légal que lorsque la classe réelle du pointeur ou de la référence est celle de la classe dérivée
- Une tentative de **downcasting** qui ne respecte pas cette condition pourrait causer une erreur d'exécution (« undefined behavior ») ou tout simplement produire un comportement inattendu
- Ce type de conversion n'est possible qu'en utilisant les opérateurs de conversion définies en C++

Programmation orientée objet

Classes et objets

Objet

- Un **objet** est une entité pouvant être créée, stockée, manipulée et détruite
- Un objet possède des attributs (propriétés)
- Un objet offre un certain nombre de méthodes (fonctions membres) pour le manipuler
- Tout objet appartient à une **classe** qui représente un type d'objet

Définition d'une classe en C++

```
class NomDeLaClasse{
```

```
public:
```

déclarations de constructeurs

déclarations de méthodes publiques

```
private:
```

déclarations de méthodes privées

```
attributs
```

Les méthodes (ou
fonctions membres)

```
};
```

La visibilité des membres

L'interface

Exemple de définition de classe

```
class Employee
{
public:
    Employee();
    Employee(string name, double salary);
    double getSalary() const;
    string getName() const;
    void setSalary(double salary);

private:
    string name_;
    double salary_; }
```

Interface

État

Remarque: par convention, nous utiliserons toujours un _ pour distinguer les variables qui correspondent aux attributs d'une classe.

Relation Objet - Classe

- La relation qui lie l'objet et sa classe est la même qu'entre une variable et son type :

`int nbInvités;`

Variable `nbInvités` de type `int`

`string nom;`

Objet `nom` de la classe standard `string`

`Employee unEmploye;`

Objet `unEmploye` de la classe `Employee`

Programmation orientée objet

***Méthodes et
manipulation d'objets***

Méthodes d'une classe

```
class Employee
{
public:
    Employee();
    Employee(string name, double salary); } Constructeurs
    double getSalary() const; } Fonctions d'accès (getters)
    string getName() const; } Fonctions de modification
    void setSalary(double salary); } (setters)
    void setName(string name);

private:
    string name_;
    double salary_;
};
```

The code defines a class `Employee` with the following members:

- Constructors:** `Employee()` and `Employee(string name, double salary)`.
- Fonctions d'accès (getters):** `double getSalary() const` and `string getName() const`.
- Fonctions de modification (setters):** `void setSalary(double salary)` and `void setName(string name)`.

Implémentation des méthodes

- En général, en C++, les méthodes sont implémentées séparément de la définition de classe
- La définition de la classe est faite dans un fichier .h et l'implementation est faite dans un fichier .cpp

Exemple d'implémentation d'une méthode

```
void Employee::setSalary(double salary)
{
    if (salary > salary_)
        salary_ = salary;
}

double Employee::getSalary()
{
    return salary_;
}
```

Paramètres d'une méthode

- Toute méthode a un **paramètre implicite**, qui correspond à l'objet sur lequel elle est appliquée
- Les autres paramètres qui apparaissent dans l'en-tête de la méthode sont les **paramètres explicites**.

Paramètres d'une méthode

- Quand on écrit:
 - `marcel.setSalary(55000);`
 - le compilateur crée en fait une fonction à deux paramètres:
 - `setSalary(marcel, 55000);`
 - Mais tout cela est **transparent pour nous**
- 

Principe d'encapsulation

- On n'a pas accès directement aux attributs d'un objet
- On modifie ou on obtient la valeur d'un attribut toujours par l'intermédiaire d'une méthode
- En résumé, on ne peut manipuler l'état d'un objet que par le biais des méthodes qui sont définies par l'interface de sa classe

Manipulation d'un objet

- Création d'un objet:

Employee

marcel;

Classe de
l'objet

Variable
identifiant l'objet

- Obtention de l'état d'un objet:

```
int salaireMarcel = marcel.getSalary();
```

- Modification de l'état d'un objet:

```
marcel.setSalary(10000);
```

Programmation orientée objet

***Constructeurs et
destructeurs***

Constructeur

- Le rôle principal d'un constructeur:
 - Initialiser les attributs lors de la création d'un objet
- En général, on a un **constructeur par défaut**, qui ne reçoit aucun paramètre, et qui donne aux attributs des valeurs par défaut
- On peut aussi avoir des **constructeurs par paramètres** qui acceptent comme paramètres les valeurs initiales que l'on veut donner aux attributs

Définition de la classe Employee

```
class Employee
{
public:
    Employee();                                Constructeur
                                                par défaut
    Employee(string name, double salary);       Constructeur par
                                                paramètres
    double getSalary() const;
    string getName() const;
    void setSalary(double salary);
    void setName(string name);

private:
    string name_;
    double salary_;
};
```

Constructeur par défaut

- Exemple d'implémentation du constructeur:

```
Employee::Employee(){  
    name_ = "unknown";  
    salary_ = 0.0;  
}
```

Quand un constructeur par défaut est-il appelé? (suite)

- Appel explicite lors de la simple déclaration d'une variable d'objet:

```
Employee marcel = Employee();
```

```
Employee marcel;
```

Forme abrégée
de l'initialisation
par défaut

Il ne faut **pas** mettre de
parenthèses pour
l'initialisation par défaut

Construit un objet de
type Employee en
fournissant des valeurs
par défaut à ses attributs

Quand un constructeur par défaut est-il appelé? (suite)

- Appel implicite lorsqu'on utilise un tableau d'objets:

```
Employee tabDeEmployee[10];
```

Quand un constructeur par défaut est-il appelé? (suite)

- Lorsque l'objet est lui-même un attribut d'un autre objet:

```
class Company
{
public:
    Company ();
    ...
private:
    Employee president_;
    ...
    int nbEmployees_;
};
```

```
Company::Company()
{
    nbEmployees_ = 0.0;
}
```

Constructeur par paramètres

- Exemple d'implémentation du constructeur:

```
Employee::Employee(string name, double salary)
{
    name_ = name;
    salary_ = salary;
}
```

Constructeur par paramètres

- Construction par paramètres d'un objet:

```
Employee marcel = Employee ("marcel", 50000);
```

Construit un objet de type Employee en fournissant des valeurs initiales à ses attributs

```
Employee marcel ("Marcel", 50000);
```

Forme abrégée de l'initialisation par paramètres

Constructeur par paramètres avec valeurs par défaut

- Il est possible de définir un constructeur par paramètres qui admet des valeurs par défaut
- Les valeurs par défaut seront alors explicitées dans la définition de la classe **seulement**
- L'implémentation reste la même que celle du constructeur par paramètres standards
- Ce type de constructeur permet de rassembler différentes surcharges de constructeur en une seule méthode

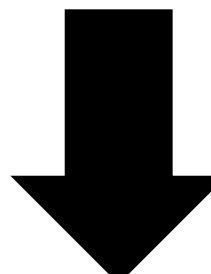
Constructeur par paramètres avec valeurs par défaut

```
class Employee
{
public:
    Employee();
    Employee(string name);
    Employee(string name, double salary);

    ...
};

class Employee
{
public:
    Employee(string name = "unknown", double salary = 0.0);

    ...
};
```



C'est un constructeur par paramètres qui agit aussi comme constructeur par défaut

Destructeur (suite)

- Un destructeur est une méthode qui est appelée lorsqu'un objet est **détruit**
- Le destructeur est défini en lui donnant le même nom que la classe précédé du tilde ~ :

```
Employee::~Employee()  
{  
    // Rien à faire dans ce cas-ci  
}
```

Quand un destructeur est-il appelé?

- Si l'objet est une variable locale à une fonction, il sera **détruit à la sortie de cette fonction**
- Si l'objet est déclaré dans le main(), il sera **détruit à la sortie du programme**
- Si l'objet est dynamique, le destructeur est appelé lorsqu'on exécute **delete** sur cet objet

Programmation orientée objet

Liste d'initialisation

Initialisation des attributs par défaut

```
class Employee
{
public:
    Employee() {}
    Employee(string name, double salary){
        name_ = name;
        salary_ = salary;
    }

private:
    string name_ = "unknown";
    double prime_ = 0.05;
    double salary_ = (1+prime_)*30000.0;
};
```

Problème: Une double initialisation est faite et qui est en lien avec l'ordre de construction d'un objet

Ordre de construction d'un objet

```
class Employee
{
public:
    2 Employee(string name, double salary){
        name_ = name;
        salary_ = salary; 5
    }

private:
    string name_ = "unknown"; 3
    double salary_ = 0.0; 4
};
```

```
int main(){
    Employee bob("Bob", 20000.0); 1
}
```

Les numéros indiquent l'ordre d'exécution des instructions lors de la construction de l'objet **bob**

Liste d'initialisation

```
class Employee
{
public:
    2 Employee(string name, double salary): salary_(salary),
        name_(name) { 5 }

private:
    string name_ = "unknown"; 3
    double salary_ = 0.0; 4
};
```

La liste d'initialisation permet de spécifier quel constructeur sera appelé lors de la construction des attributs et donc d'éviter la double initialisation des attributs

```
int main(){
    Employee bob("Bob",
        20000.0); 1
}
```

Délégation de constructeur

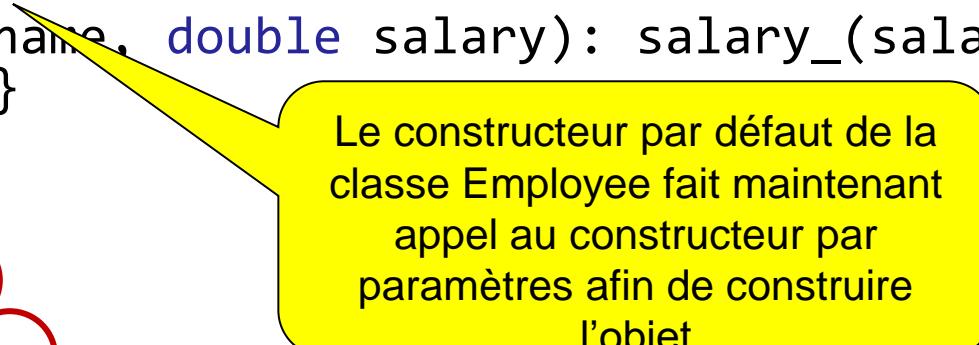
```
class Employee
{
public:
    Employee(): name_(“unknown”), salary_(0.0) {}
    Employee(string name, double salary): salary_(salary),
    name_(name) {}

private:
    string name_;
    double salary_;
};
```

Problème: Il y a une redondance au niveau de l'implémentation des constructeurs qui peut être résolue grâce à la delegation de constructeur

Délégation de constructeur

```
class Employee{  
public:  
    2 Employee(): Employee("unknown", 0.0) {}  
    3 Employee(string name, double salary): salary_(salary),  
        name_(name) { 6 }  
  
private:  
    string name_; 4  
    double salary_; 5  
};  
  
int main(){  
    Employee bob; 1  
}
```



Le constructeur par défaut de la classe Employee fait maintenant appel au constructeur par paramètres afin de construire l'objet

Programmation orientée objet

Allocation dynamique

Allocation automatique et allocation dynamique

- Automatique:

```
Employee e1("John", 15000);
```

```
Employee e1 = Employee ("John",  
15000);
```

- Dynamique:

```
Employee * e1;
```

```
e1 = new Employee ("John", 15000);
```

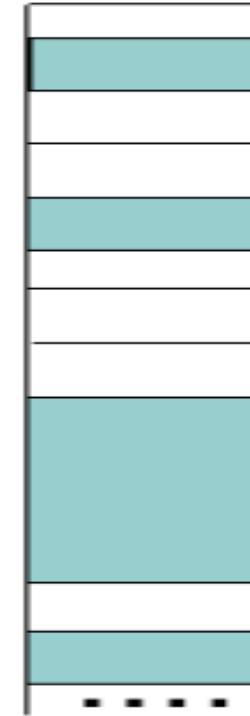
Étapes de l' allocation d'un pointeur

```
Employee* e1 = new Employee ("John", 15000);
```

Étapes de l' allocation d'un pointeur

```
Employee* e1 = new Employee ("John", 15000);
```

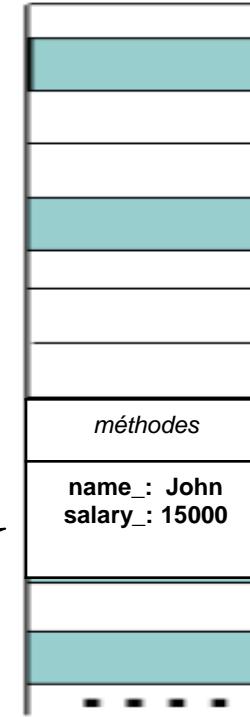
Un espace disponible dans le tas assez grand pour accueillir l'objet est trouvé



Étapes de l' allocation d'un pointeur

```
Employee* e1 = new Employee ("John", 15000);
```

Un objet de type
Employee y est construit

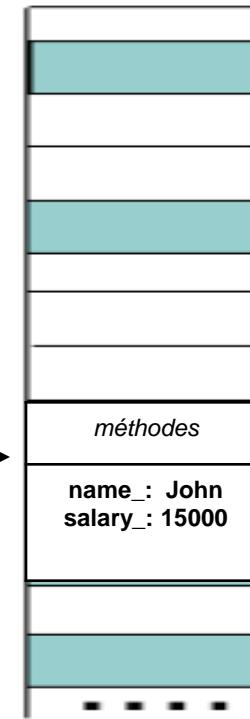


Étapes de l' allocation d'un pointeur

```
Employee* e1 = new Employee ("John", 15000);
```

Affectation du pointeur
dans la variable

e1

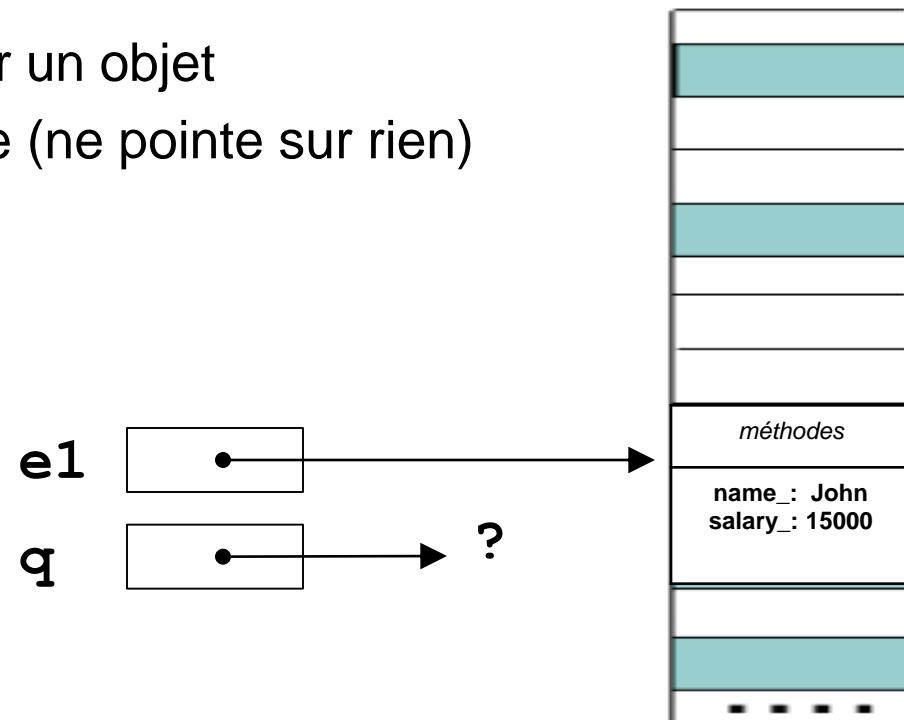


Pointeurs

```
Employee* e1 = new Employee ("John", 15000);
```

```
Employee* q;
```

- Le pointeur **e1** pointe sur un objet
- Le pointeur **q** est invalide (ne pointe sur rien)



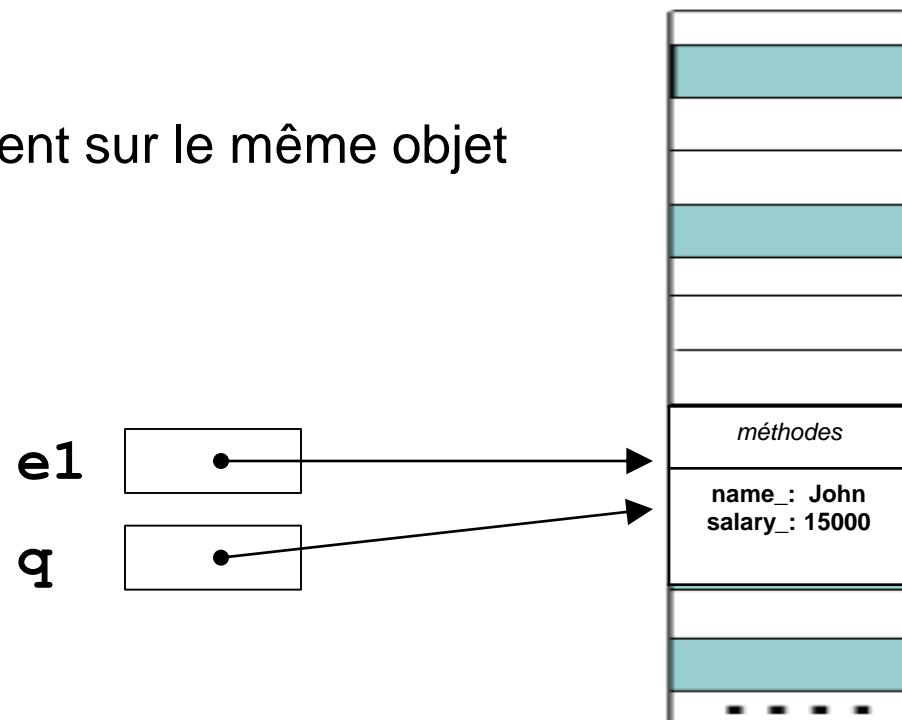
Pointeurs

```
Employee* e1 = new Employee ("John", 15000);
```

```
Employee* q;
```

```
q = e1;
```

- Les deux pointeurs pointent sur le même objet



Initialisation des pointeurs

- Prenez l' habitude de toujours initialiser vos pointeurs:

```
Employee* e1 = nullptr; //C++11
```

Initialisation des pointeurs

- Dans un constructeur aussi, si la classe définit un attribut dynamique:

```
class UneClasse
{
public:
    UneClasse() ;
    ...
private:
    Point* monAttribut_ ;
    ...
} ;
```

```
UneClasse::UneClasse()
{
    monAttribut_ = nullptr;
}
```

Dé-référencement d' un pointeur

- Si **e1** est un pointeur, l' expression ***e1** retourne l' objet pointé par **e1**
- Si on veut appliquer une méthode de l' objet en question:
(*e1) .getSalary ()
- Une autre forme synonyme et plus pratique:
e1->getSalary ()

Désallocation de mémoire

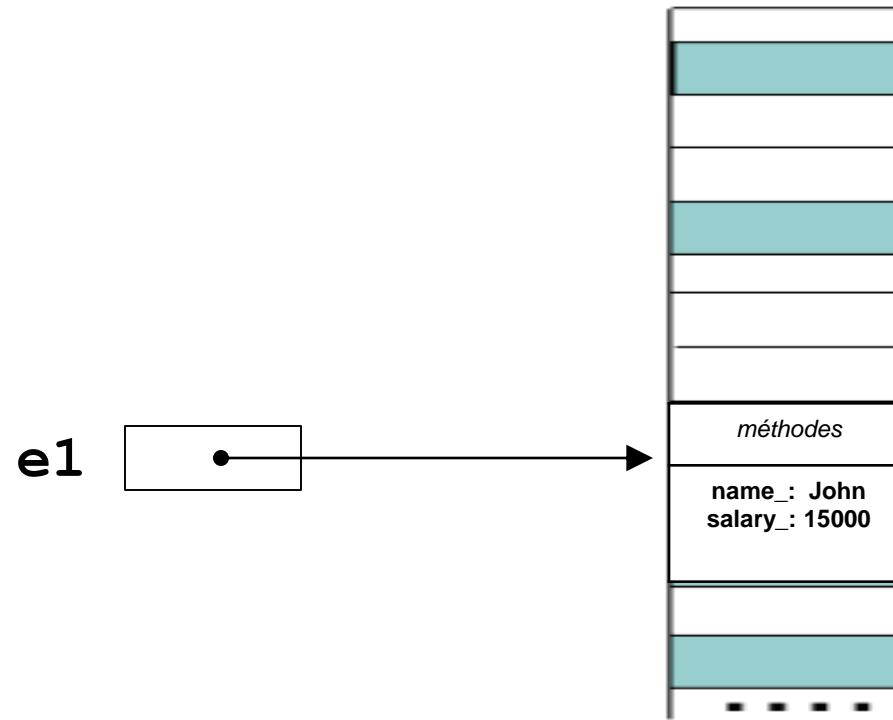
- Pour tout appel à **new** il faut retrouver quelque part un appel à **delete** qui désalloue la mémoire:

```
int main()
{
    Employee* e1 = new Employee
    ("John", 15000);

    ...
    delete e1;
}
```

Étapes de la désallocation

```
delete e1;
```

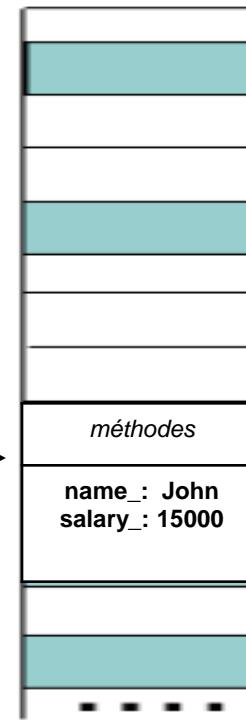


Étapes de la désallocation

`delete e1;`

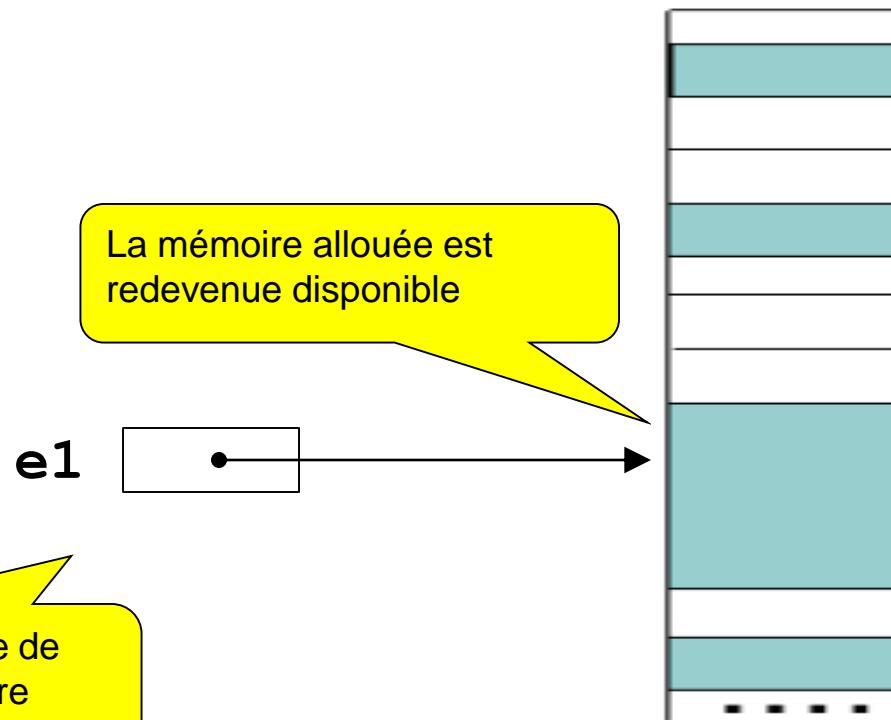
On exécute le destructeur
(dans ce cas-ci, on suppose
qu'il ne fait rien)

`e1`



Étapes de la désallocation

`delete e1;`



Attention, le pointeur continue de pointer sur un espace mémoire invalide

Désallocation

- Ainsi, après l' exécution de **delete**, le pointeur continue de pointer sur le même espace mémoire, devenu invalide
- Pour éviter toute tentative de déréférencer à nouveau ce pointeur, il est bon de le réinitialiser à **nullptr**:

```
delete e1;  
e1 = nullptr;
```

Tableaux et pointeurs

- En C++, l'adresse d'un tableau est en fait un pointeur qui pointe sur le premier élément du tableau:

```
int a[5] = {1,2,3,4,5};  
int* p = a;  
*p = 10;  
cout << a[0];
```

Quelle valeur sera affichée?

Arithmétique des pointeurs

- Si **p** est un pointeur sur un entier, l' expression **p+3** pointe sur le troisième entier en mémoire situé après celui pointé par **p**
- Autre exemple:

```
int a[5] = {1,2,3,4,5};  
int* p = a;  
++p;  
++p;  
cout << *p;
```

Quelle valeur sera affichée?

Tableau dynamique

- Considérons l' instruction suivante:

```
int n;
```

```
cin >> n;
```

```
Employee* listEmployees = new Employee[n];
```

- L' effet de cette instruction est la création sur le tas d' un tableau dynamique dont la taille sera déterminée lors de l' exécution du programme
- Il ne faudra pas oublier de désallouer mémoire:

```
delete[] listEmployees;
```

Récapitulons

- Tableau d'employés alloué automatiquement:

Employee

listEmployees[6] ;

...

listEmployees[0]

PILE

Employee("",0)

Récapitulons

- Tableau d'employés alloué automatiquement:

```
Employee listEmployees [6] ;
```

```
...
```

```
listEmployees [3] . setSalary (20) ;
```

```
listEmployees [3]
```

PILE

Employee("",0)
Employee("",0)
Employee("",0)
Employee("",20)
Employee("",0)
Employee("",0)

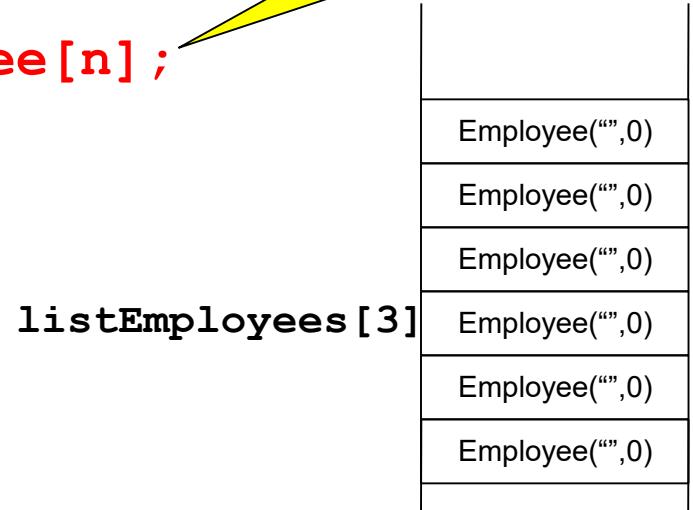
Récapitulons

- Tableau d'employés alloué dynamiquement:

```
int n;  
cin >> n;  
Employee* listEmployees = nullptr;  
...  
listEmployees = new Employee[n];
```

Une séquence de
n objets de la classe
Employee est ajoutée
dans le heap (tas).
Ici n = 6

TAS



Récapitulons

Tableau d'employés alloué dynamiquement:

```
Employee* listEmployees = nullptr;  
...  
listEmployees = new Employee[n];  
...  
listEmployees[3].setSalary(20);
```

listEmployees[3]

TAS

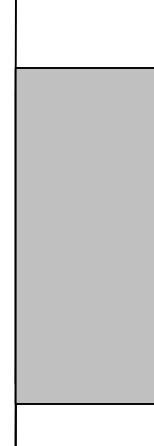
Employee("",0)
Employee("",0)
Employee("",0)
Employee("",20)
Employee("",0)
Employee("",0)

Récapitulons

Tableau d'employés alloué dynamiquement:

```
Employee* listEmployees = nullptr;  
...  
listEmployees = new Employee[n];  
...  
listEmployees[3].setSalary(20);  
...  
delete[] listEmployees;  
listEmployees = nullptr;
```

TAS



Récapitulons

Tableau de pointeurs d'employés alloué automatiquement:

```
Employee* listEmployees[6];  
for ( int i =0 ; i < 6 ; i++)  
    listEmployees[i] = nullptr;
```

On crée un tableau automatique de 6 pointeurs (qui ne sont pas initialisés).

```
listEmployees[0] = new Employee ("Mark",10); PILE
```

TAS

...

```
listEmployees[5] = new Employee ("John",30);
```

listEmployees[0]

Employee
(Mark,10)

...

```
listEmployees[5]->setSalary(20);
```

Employee
(John,30)

...

```
for (int i = 0; i < 6; ++i) {
```

listEmployees[5]

```
    delete listEmployees[i];
```

}

Récapitulons

Tableau de pointeurs d'employés alloué dynamiquement:

```
Employee** listEmployees;
int n;
cin >> n;
...
listEmployees = new Employee*[n];
for ( int i =0 ; i < 6 ; i++)
    listEmployees[i] = nullptr
.....
listEmployees[0] = new Employee("Mark",10);
...
listEmployees[5] = new Employee("John",30);
...
listEmployees[3]->setSalary(20);
...
for (int i = 0; i < 6; ++i) {
    delete listEmployees[i];
}
delete[] listEmployees;
listEmployees = nullptr;
```

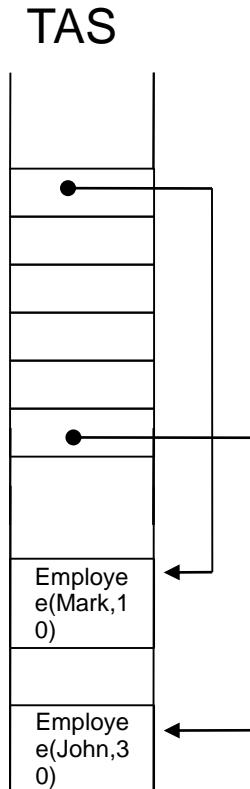
Une séquence de n
pointeurs est ajoutée
dans le heap.
Ici n = 6

listeEmployees[0]

listeEmployees[5]

Employe e(Mark,10)

Employe e(John,30)



Programmation orientée objet

Rappel sur les symboles * et &

Déclaration de variables ou d'objets * et &

- Définition d'un pointeur:

```
int * Ptr = 0;
```

Lors de la définition d'un pointeur, on peut uniquement initialiser la valeur de ce pointeur (et non la valeur vers laquelle il pointe).

Par exemple:

float* ptr= nullptr valide

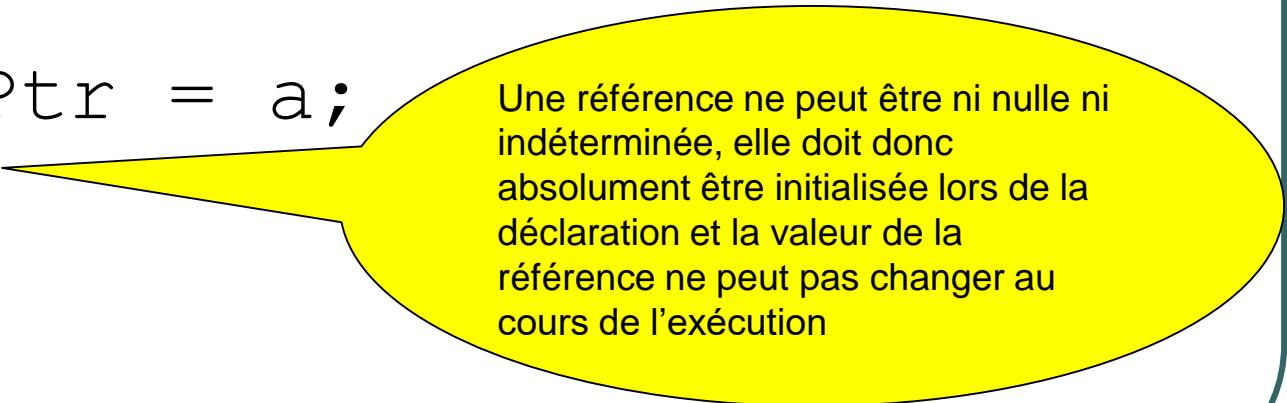
float* ptr= 10.0 non valide

Déclaration de variables ou d'objets &

- Définition d'une référence: permet de manipuler une variable sous un autre nom:

```
int a ;
```

```
int & Ptr = a;
```



Une référence ne peut être ni nulle ni indéterminée, elle doit donc absolument être initialisée lors de la déclaration et la valeur de la référence ne peut pas changer au cours de l'exécution

Section des instructions: opérateur *

Multiplication:

```
res = a * b;
```

Déréférencement d'un pointeur:

```
*Ptr = 10;
```

Section des instructions: opérateur &

- Opération Et binaire:

```
res = a & b;
```

- Adresse d'une variable:

Si on a les déclarations suivantes

```
int a ;
```

```
int * Ptr;
```

```
Ptr = &a;
```

Déclaration

Lorsqu'on déclare plusieurs pointeurs/références du même type, il faut répéter le */&.

Par exemple:

`int* a, * b, * c;` (déclare trois pointeurs)

`int* a, b, c;` (déclare un pointeur et 2 entiers)

`int* u, & v = b , w;` (déclare un pointeur, une référence et un entier)

Programmation orientée objet

Méthodes constantes

Motivation

- Nous avons vu que certaines méthodes ne servent qu'à accéder à des attributs d'un objet
- Pour éviter toute confusion, on aimerait que ces méthodes ne puissent effectuer aucune modification sur les attributs de l'objet

Implémentation d' une fonction d' accès

- Pour assurer que la méthode ne modifie aucun des attributs de l' objet, on ajoute le mot clé **const** après l' en-tête de la fonction:
 - dans la définition de la classe
 - dans l' implémentation de la fonction

Exemple de fonction d' accès

```
string Employee::getName() const  
{  
    return name_;  
}
```

Le compilateur retournera un message d' erreur si la fonction contient une instruction qui tente de modifier la valeur d' un attribut de l' objet **ou si la fonction utilise une méthode qui n' a pas été déclarée constante.**

Exemple de code erroné avec const

```
string Employee::getName() const  
{  
    name_ = "pierre";  
    return name_;  
}
```

On n'a pas le droit de modifier la valeur d'un attribut de l'objet.

Autre exemple de code erroné avec const

```
class Employee
{
public:
    Employee(string name =
              "unknown", double salary
              = 0);
    double getSalary();
    string getName();
    ...
    void print() const;

private:
    string name_;
    double salary_;
};
```

```
string Employee::getName()
{
    return name_;
}

double Employee::getSalary()
{
    return salary_;
}

void Employee::print() const
{
    cout << getName() << endl;
    cout << getSalary() << endl;
```

Erreur!
Pourquoi?

Autre exemple de code erroné avec const

```
class Employee
{
public:
    Employee(string name =
              "unknown", double salary
              = 0);
    double getSalary() const;
    string getName() const;
    ...
    void print() const;

private:
    string name_;
    double salary_;
};
```

```
string Employee::getName() const
{
    return name_;
}

double Employee::getSalary() const
{
    return salary_;
}

void Employee::print() const
{
    cout << getName() << endl;
    cout << getSalary() << endl;
}
```

Encore un exemple de code erroné avec const

```
class Employee
{
    ...
    void print();
    ...
};

class Company
{
public:
    ...
    void print() const;
private:
    ...
    Employee president_;
};

void Company::print() const
{
    ...
    president_.print();
    ...
}
```



Erreur!!
Pourquoi?

Encore un exemple de code erroné avec C++

```
class Employee  
{  
    ...  
    void print();  
    ...  
};
```

Cette méthode n'est pas déclarée constante.

```
class Company  
{  
public:  
    ...  
    void print() const;  
private:  
    ...  
    Employee president_;  
};
```

Cette méthode n'a pas le droit de modifier un attribut.

```
void Company::print() const  
{  
    ...  
    president_.print();  
    ...  
}
```

Puisque la méthode `print` n'est pas déclarée constante, elle pourrait donc modifier l'attribut `president` (même si elle ne le fait pas en pratique). Le compilateur refusera donc de compiler ce code.

Encore un exemple de code erroné avec const

```
class Employee
{
    ...
    void print() const;
    ...
};

void Company::print() const
{
    ...
    president_.print();
    ...
}
```

```
class Company
{
public:
    ...
    void print() const;
private:
    ...
    Employee president_;
};
```

Programmation orientée objet

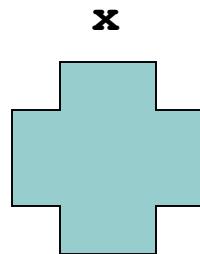
Passage de paramètres

Deux méthodes de passage de paramètre

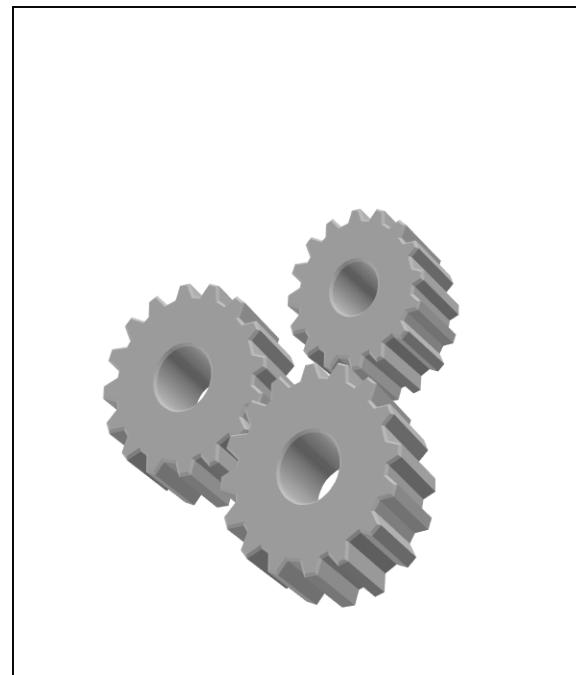
- ***Passage par valeur***: on fait une copie du paramètre et c' est cette copie qui sera utilisée à l' intérieur de la fonction
- ***Passage par référence***: on passe une référence à une entité, c' est-à-dire que l' entité passée en paramètre est manipulée directement, mais sous un autre nom

Passage par valeur

FONCTION

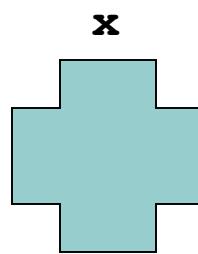


Un objet est
passé en
paramètre.

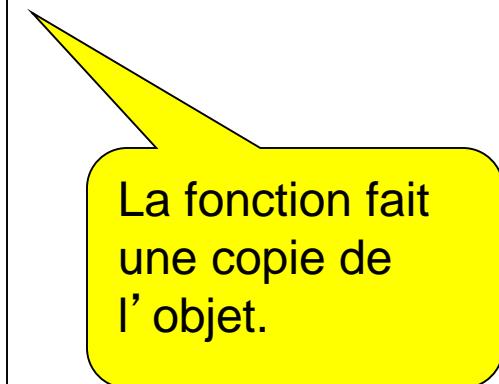
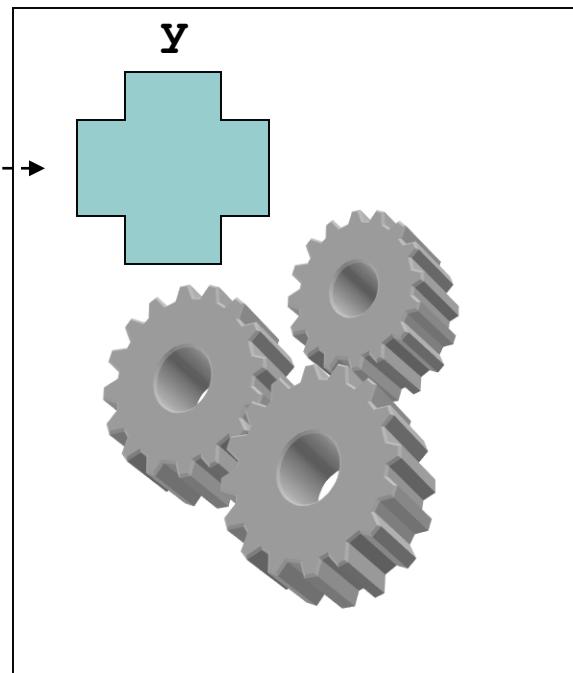


Passage par valeur

FONCTION

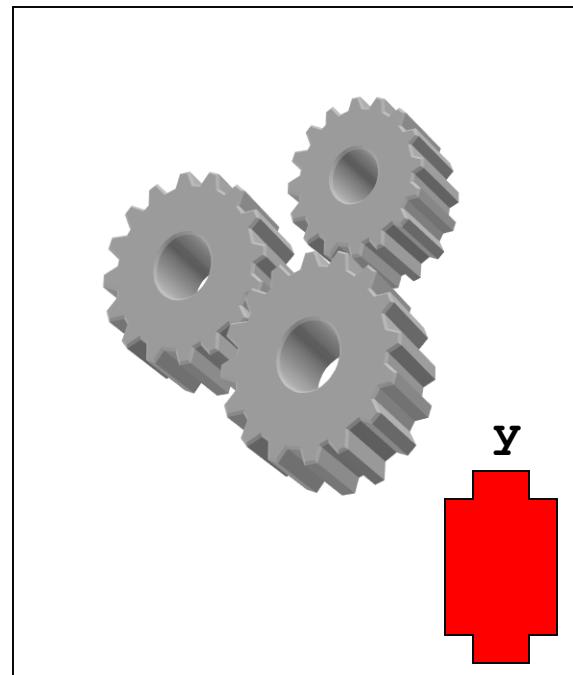
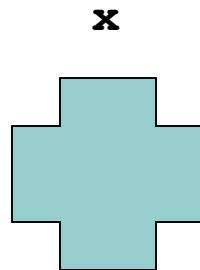


→



Passage par valeur

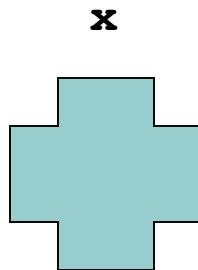
FONCTION



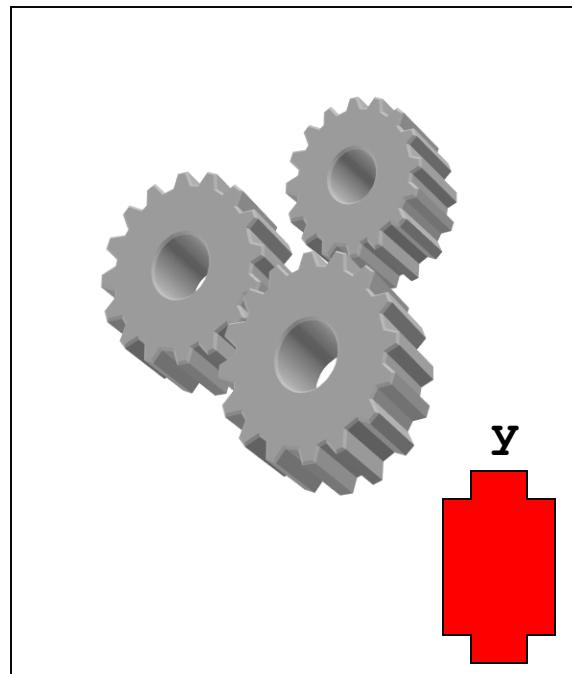
La fonction manipule l' objet et peut aussi changer son état.

Passage par valeur

FONCTION

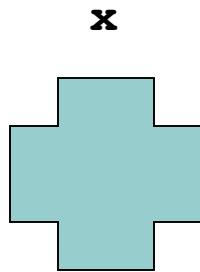


L'objet initial est demeuré inchangé.

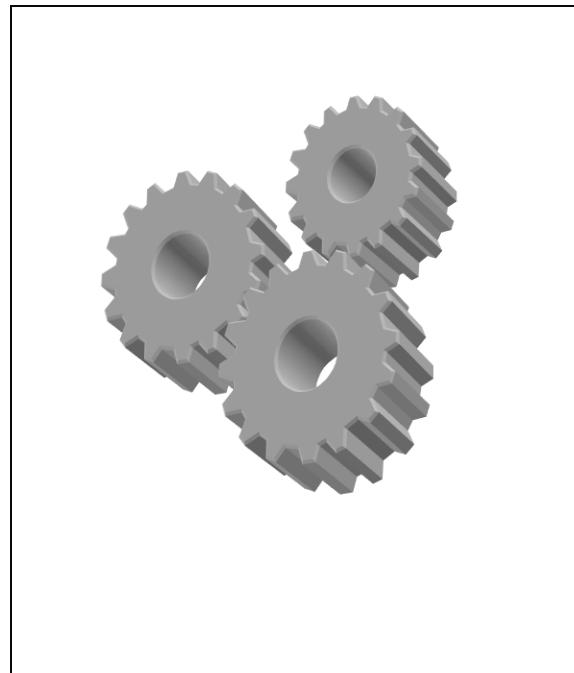


Passage par référence

FONCTION

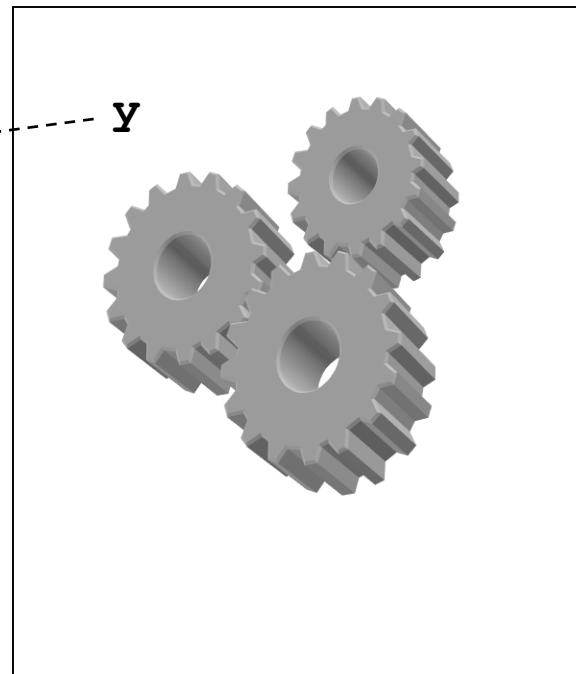
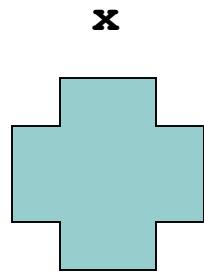


Un objet est
passé en
paramètre.



Passage par référence

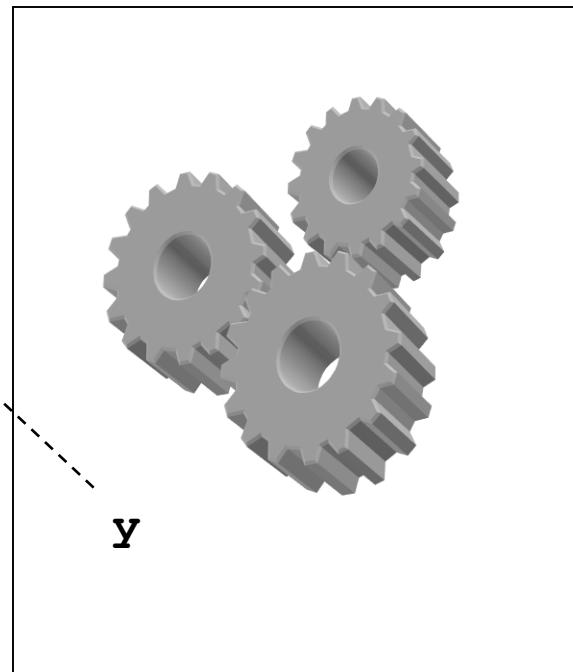
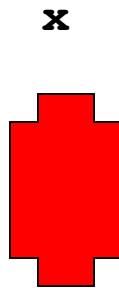
FONCTION



La fonction utilise un autre nom pour le même objet.

Passage par référence

FONCTION



La fonction manipule l' objet et peut aussi changer son état.

Passage par référence

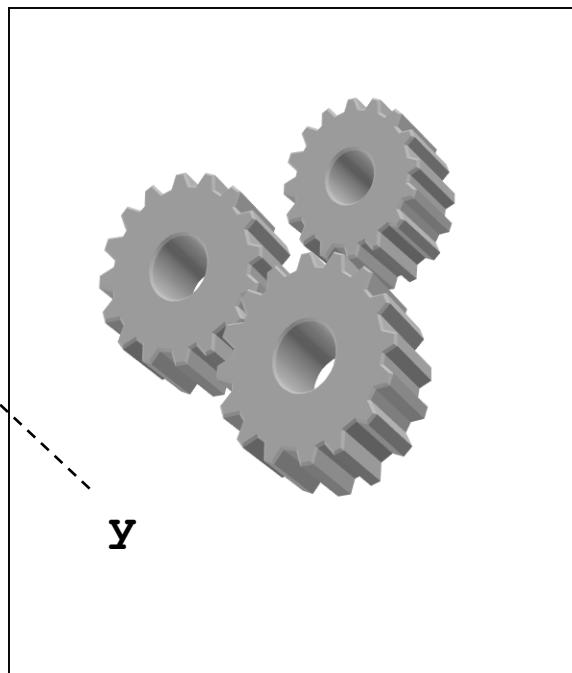
FONCTION

x



L' objet initial peut avoir changé.

y



Exemple problématique

```
void increase(Employee employe, double percentage)
{
    double newSalary= employe.getSalary() *
                    (1 + percentage/100);
    employe.setSalary(newSalary);
}

int main()
{
    Employee michel("Michel",100);
    increase(michel,5);
    cout << michel.getSalary();
    ...
}
```

Désolé, mais
le salaire n'a
pas changé!

Employe*& ?

Exemple corrigé

```
void increase(Employee& employe, double percentage)
{
    double newSalary= employe.getSalary() *
                    (1 + percentage/100);
    employe.setSalary(newSalary);
}

int main()
{
    Employee michel("Michel",100);
    increase(michel,5);
    cout << michel.getSalary();
    ...
}
```

employe est une référence au même objet que celui contenu dans la variable michel.

Le salaire aura finalement été augmenté!

Référence constante

- Souvent, on passe un objet par référence non pas parce qu'on veut le modifier, mais plutôt parce qu'on veut **éviter une copie qui est coûteuse**
- Pour éviter que cet objet soit modifié, on utilisera alors une référence constante

Référence constante (exemple)

```
void printCompany(const Company& c)
{
    cout << "Company " << c.getName();
    if (c.hasEmployees()) {
        cout << " has " << c.getNumberEmployees() << " employee(s) : "
            << endl;
        for (int i = 0; i < c.getNumberEmployees(); i++) {
            cout << " - Employee " << (i+1) << ":" <<
            c.getEmployeeByPos(i).getName() << endl;
        }
    } else {
        cout << " doesn't have any employee"
    }
}
```

Le compilateur permettra seulement l'utilisation de méthodes qui ont été déclarées const.

Programmation orientée objet

Création et manipulation de
vecteurs

Vecteur

- Un vecteur est une **collection séquentielle d' items du même type**
- On l' utilise comme si c' était un tableau
- *Particularité intéressante:* si on insère un nouvel item dans un vecteur déjà plein, sa **taille sera automatiquement augmentée** afin de pouvoir recevoir ce nouvel item

Vecteur (suite)

- La classe **vector** fait partie de la STL (Standard Template Library) de C++
[cppreference - vector](#)
- Pour utiliser les vecteurs, il faut donc inclure la classe dans le programme:

```
#include <vector>
```

Création d'un vecteur

- La classe **vector** est une classe **générique**, il faut donc toujours spécifier le type des éléments qu' il contiendra lorsqu'on instancie un vecteur:

```
int main() {  
    vector<int> vectInt;  
    vector<Employee> vecEmploye;  
}
```

Taille et capacité

- Il est important de distinguer la taille et la capacité d' un vecteur
- Un vecteur est un *conteneur séquentiel* qui maintient à l' interne un **tableau**
- *Taille*: nombre d' éléments effectivement **contenus** dans le tableau interne. Elle est accessible en utilisant la méthode **size()**
- *Capacité*: taille du tableau **interne**. Elle est accessible en utilisant la méthode **capacity()**

Insertion et retrait d'éléments

- Étant donné qu'un vecteur est basé sur un tableau interne, on le remplit et le vide **par la fin** de manière efficace:
 - **push_back()**: Ajoute un élément à la fin du vecteur
 - **pop_back()**: Retire le dernier élément du vecteur

Insertion et retrait d'éléments (suite)

- Les fonctions **insert()** et **erase()** permettent d'insérer et retirer des éléments n'importe où dans le vecteur
- Il faut **éviter de faire appel à ces méthodes**, à cause de leur coût en temps d'exécution puisqu'elles font le décalage complet des éléments du tableau qui suivent l'emplacement d'insertion ou de retrait
- Si vous devez fréquemment les appeler, c'est que **l'utilisation d'un vecteur n'est pas appropriée**

Retrait d' un élément au milieu

- Si l'ordre des éléments n'a pas d'importance, l'opération n'est pas trop coûteuse:

```
int main() {  
    vector<int> vectInt;  
  
    for (unsigned int i = 0; i < 10; i++)  
        vectInt.push_back(i);  
  
    vectInt[5] = vectInt.back(); ①  
    vectInt.pop_back();  
}
```

②

Accès aux éléments

- Étant donné que les éléments sont stockés de manière contigue, on peut accéder de manière efficace aux éléments si on connaît leur index avec:
 - L'opérateur d'accès aléatoire [] comme pour un tableau
 - La méthode **at()**
- Le premier et le dernier élément sont également accessibles en utilisant les méthodes **front()** et **back()**

Manipulation d'un vecteur

```
int main() {
    vector<int> vectInt;
    vectInt.push_back(0);
    vectInt.push_back(1); } Ajout d'éléments
    cout << vectInt.size() << endl; // 2
    cout << vectInt[0] << endl; // 0
    cout << vectInt.at(0) << endl; // 0
    cout << vectInt.front() << endl; // 0
    cout << vectInt.back() << endl; // 1 } Accès aux
                                    éléments
    vectInt.pop_back(); } Retrait d'éléments
    vectInt.pop_back();
    cout << vectInt.size() << endl; // 0
}
```

Itération sur les éléments

- On peut itérer sur l'intervalle complet du vecteur en utilisant une boucle for

```
int main() {
    vector<int> vectInt;

    for(unsigned int i = 0; i < 10; i++)
        vectInt.push_back(i);

    //Affiche 0 1 2 3 4 5 6 7 8 9
    for (unsigned int i : vectInt)
        cout << i << " ";

    cout << endl;
}
```

Programmation orientée objet

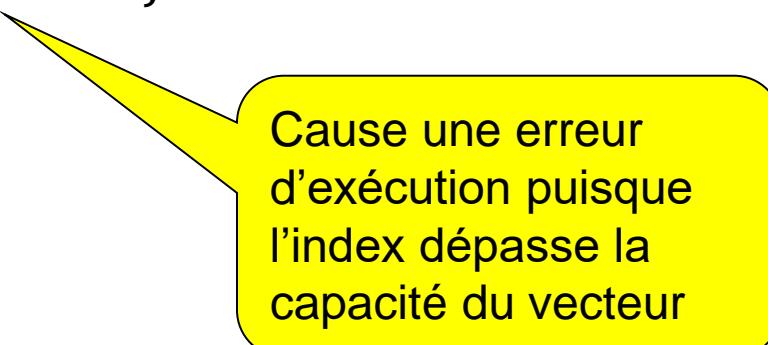
Capacité d'un vecteur

Capacité

- Par défaut, la capacité d' un vecteur est 0
- La capacité d'un vecteur est augmentée automatiquement lorsqu'il n'y a plus assez de place dans le vecteur lorsqu'on tente d'ajouter un nouvel élément
- On peut **initialiser la taille** d' un vecteur lors de sa déclaration (dans ce cas, il sera rempli par des valeurs par défaut ou en appelant le **constructeur par défaut** dans le cas d' un vecteur d' objets)
- Si on sait qu' un indice est inférieur à la capacité d' un vecteur, on peut modifier la valeur à cette position comme dans un tableau (**soyez prudent!**)

Capacité (suite)

```
int main() {  
    vector<int> unTableau(10);  
    vector<Employee> listEmployees(10);  
  
    unTableau[6] = 64;  
    listEmployees[3] = Employee("John", 15000);  
  
    unTableau[10] = 10;  
}
```



Cause une erreur
d'exécution puisque
l'index dépasse la
capacité du vecteur

Augmentation de la capacité

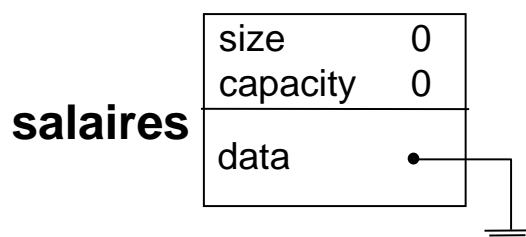
- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```



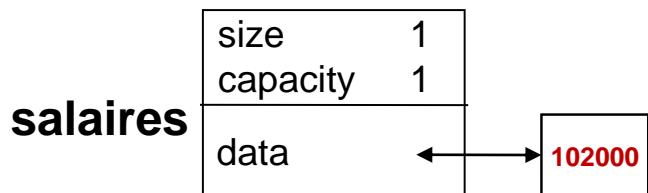
Le vecteur est créé
avec capacité nulle.

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

On alloue un tableau de taille 1 pour contenir le nouvel élément.

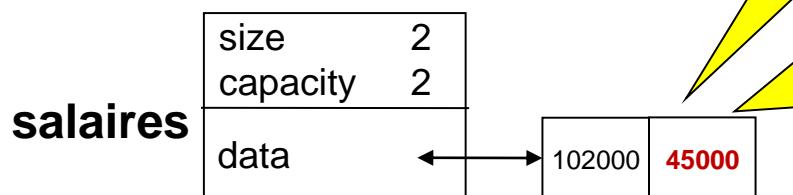


Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

Comme le tableau est plein, on augmente sa taille pour pouvoir ajouter le nouvel élément.

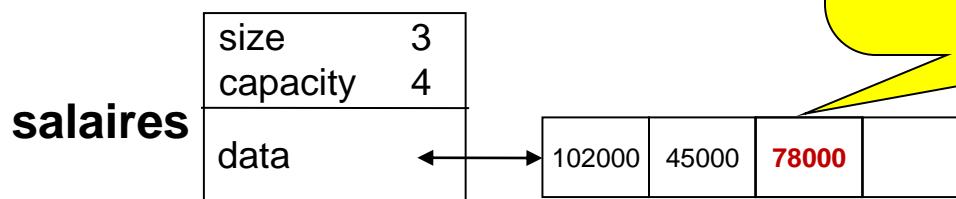


(1) allouer l'espace double sur le tas, (2) copier tout vers cette espace, et (3) détruire l'espace originale

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

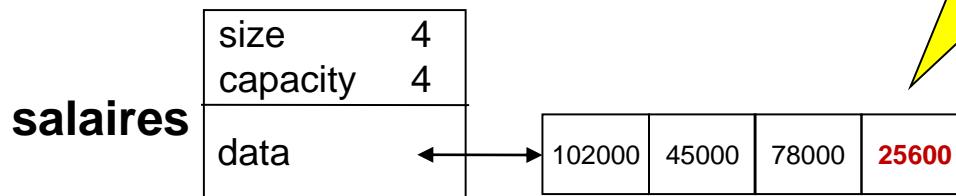


Encore une fois, comme le tableau est plein, on augmente sa taille pour pouvoir ajouter le nouvel élément.

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```



On ajoute le nouvel élément.

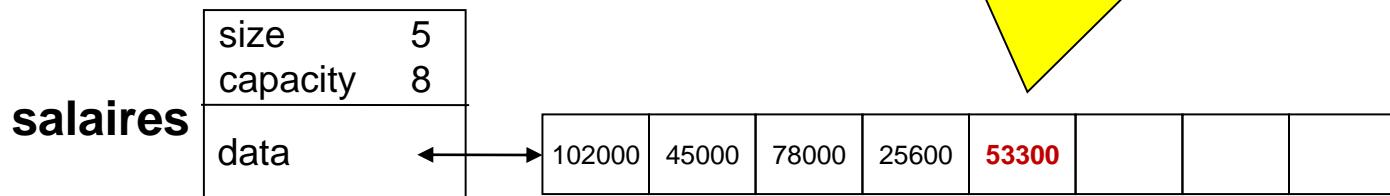
Remarquez qu' on n'a pas atteint la capacité maximale du tableau.

Augmentation de la capacité (suite)

- Soit par exemple le programme suivant:

```
int main() {  
    vector< double > salaires;  
    salaires.push_back(102000.0);  
    salaires.push_back(45000.0);  
    salaires.push_back(78000.0);  
    salaires.push_back(25600.0);  
    salaires.push_back(53300.0);  
}
```

Encore une fois, comme le tableau est plein, on augmente sa taille pour pouvoir ajouter le nouvel élément.

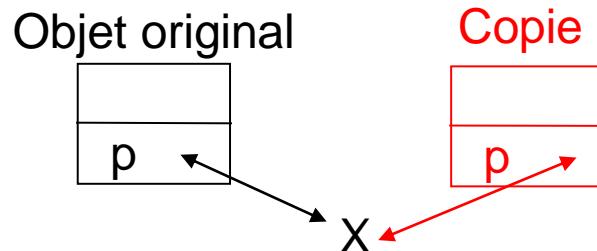


Programmation orientée objet

Copie et référencement de
vecteurs

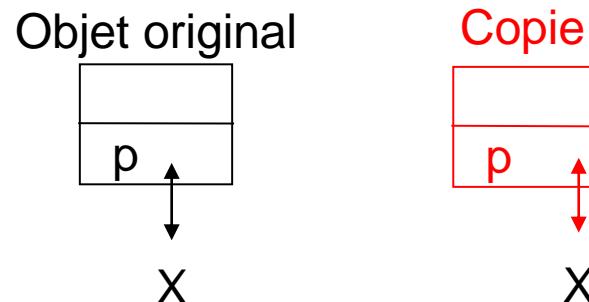
Shallow copy

- Jusqu'à maintenant, nous avons toujours considéré que les objets sont copiés de manière superficielle ("**shallow copy**")
- Lorsqu'on fait une copie superficielle, l'objet est copié tel quel. Si un objet contient un pointeur, celui-ci sera partagé entre l'objet original et la copie



Deep copy

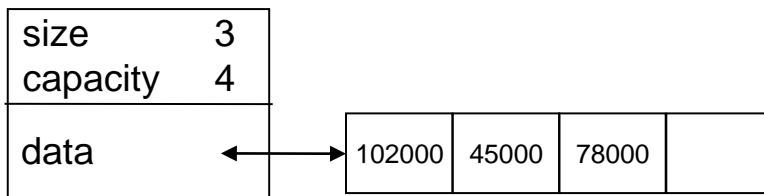
- Une copie en profondeur copie le contenu du pointeur vers un nouvel emplacement mémoire
- Lorsqu'une copie en profondeur est faite, il n'y a aucun partage de mémoire entre l'objet original et la copie



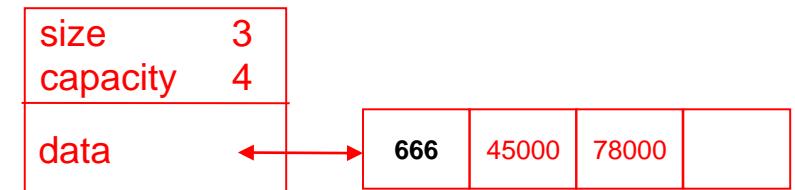
Copie d'un vecteur

- Lorsqu'un vecteur est copié, il s'agit toujours d'une copie en profondeur
- En effet, lorsqu'une copie est réalisée, ce n'est pas le pointeur qu'on copie, mais tout le tableau dynamique
- Ainsi, toute modification à la copie n'a aucun impact sur le vecteur original

Vecteur original



Copie



Passage de vecteur en paramètre

- Lorsqu' on passe un vecteur par valeur à une fonction, une copie éventuellement coûteuse est réalisée
- Il est donc **généralement plus approprié de passer un vecteur par référence**
- Si on veut éviter qu' il soit modifié, on passe une référence constante:

```
void f(const vector<int>& unVecteur)
{
    ...
}
```

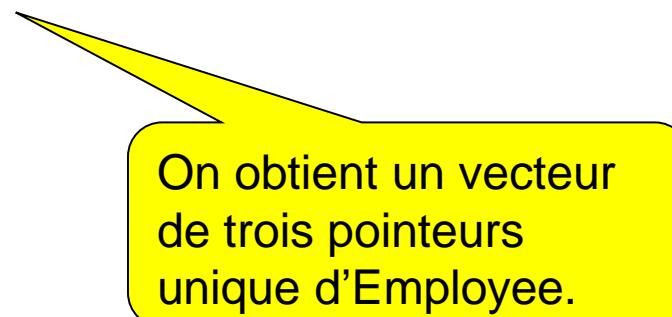
Remarques sur les vecteurs d'objets

- Si on a un vecteur d'objets, l'appel à **push_back()** fera une copie de l'objet dans le vecteur, ce qui peut devenir coûteux
- C'est pourquoi on a souvent tendance, en C++, à **utiliser des vecteurs de pointeurs sur des objets**

Remarques sur les vecteurs d'objets (suite)

- Exemple:

```
int main() {  
    vector<unique_ptr<Employee>> employes;  
    employes.push_back(make_unique<Employee>("John", 15000));  
    employes.push_back(make_unique<Employee>("Mark", 16000));  
    employes.push_back(make_unique<Employee>("Jenny", 12000));  
}
```



On obtient un vecteur de trois pointeurs unique d'Employee.

Programmation orientée objet

Pointeurs intelligents

Motivation

- Un pointeur brut est un pointeur dont le type est directement T^* , où T peut être un int, un string, un Employee, ...
- Les pointeurs bruts sont une source majeure de problèmes.
- Les problèmes se trouvent majoritairement au niveau de:
 - la possession de mémoire
 - Qui doit désallouer la mémoire dynamique?
 - Quand désallouer la mémoire d'un pointeur partagé?
 - la désallocation des pointeurs
 - Risque de fuite de mémoire
 - Risque d'erreur d'exécution(« undefined behavior »)

Bonnes pratiques en C++

- Pour ces raisons: (C++ core guidelines)
 - Jamais transférer la possession de la mémoire à l'aide d'un pointeur ou référence brut (l.11)
 - Un pointeur ou référence brut ne devrait jamais posséder une ressource mémoire (R.3, R.4)

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

Qu'est-ce qu'un pointeur intelligent?

- Un pointeur intelligent est une classe de la librairie standard qui encapsule la notion de pointeur:
 - S'occupe d'allouer et désallouer la mémoire dynamique
 - Les opérateurs de déréférencement (*, ->), d'accès ([] et de comparaison sont accessibles par un pointeur intelligent de la même manière qu'un pointeur brut
- Pour pouvoir utiliser les pointeurs intelligents, on doit inclure leur librairie:

```
#include <memory>
```

Classes de pointeurs intelligents

- Propriétaire unique: `unique_ptr`
 - Un objet de type `unique_ptr` est le seul à pointer vers l'espace mémoire qu'il a créé
 - Un objet de type `unique_ptr` décide de la durée de vie de l'espace mémoire vers lequel il pointe
 - Plusieurs propriétaires: `shared_ptr`
 - Un objet de type `shared_ptr` peut partager la mémoire qu'il a créée avec d'autres objets du même type
 - Lorsqu'il n'y a plus d'objets de type `shared_ptr` qui pointe vers l'espace mémoire, l'espace est automatiquement désallouée
- cppreference – [shared_ptr](#) et [unique_ptr](#)

Création de pointeurs intelligents

```
int main() {  
    unique_ptr<Employee> emp_ptr_unique =  
        make_unique<Employee>();  
    shared_ptr<Employee> emp_ptr_shared =  
        make_shared<Employee>("Bob");  
    unique_ptr<Employee[]> emp_tableau =  
        make_unique<Employee[]>(1);  
}
```

À la sortie de la fonction, les variables locales sont détruites. L'espace mémoire allouée dynamiquement par les pointeurs intelligents est désallouée par ceux-ci lors de leur destruction.

Manipulation des pointeurs intelligents

```
int main() {
    unique_ptr<Employee> emp_ptr_unique =
        make_unique<Employee>("Bob");
    emp_ptr_unique->setName("Bobette");
    cout << emp_ptr_unique->getName() << endl; // "Bobette"
    Employee emp;
    *emp_ptr_unique = emp;
    cout << emp_ptr_unique->getName() << endl; // "unknown"
    unique_ptr<Employee[]> emp_tableau =
        make_unique<Employee[]>(1);
    cout << emp_tableau[0].getName() << endl; // "unknown"
}
```

Accès au pointeur brut

```
int main() {
    unique_ptr<Employee> emp_ptr_unique =
        make_unique<Employee>("Bob");
    Employee* emp_ptr = emp_ptr_unique.get();
}
```

unique_ptr

- Une erreur de compilation se produit lorsqu'on tente de copier un pointeur intelligent unique.
- Par contre, on peut transférer la possession de la mémoire dynamique en utilisant la fonction std::move

```
int main() {  
    unique_ptr<Employee> emp_ptr_1 =  
        make_unique<Employee>("Bob");  
    unique_ptr<Employee> emp_ptr_2 = move(emp_ptr_1);  
}
```

Pourquoi?

Transfert de possession de
emp_ptr_1 à emp_ptr_2.
emp_ptr_1 devient
nullptr.

Exemple de possession de mémoire

• Avec les pointeurs bruts

```
class MaClasse {  
public:  
    void posseder(Item* item) {  
        delete item_;  
        item_ = item;  
    }  
    ~MaClasse() { delete item_; }  
private:  
    Item* item_ = nullptr;  
};
```

Pas évident de savoir que l'appelant donne sa possession de la mémoire.

Doit libérer manuellement, sinon fuite

Doit initialiser sinon « undefined behavior » lors du premier delete.

Manque-t-il un delete après?

```
int main() {  
    MaClasse a;  
    Item* item = new Item;  
    a.posseder(item);  
    a.posseder(new Item);  
}
```

Exemple de possession de mémoire

- Avec pointeurs intelligents uniques

```
class MaClasse {  
public:  
    void posseder(unique_ptr<Item> item){  
        item_ = move(item);  
    }  
private:  
    unique_ptr<Item> item_;  
};
```

Appelant doit donner sa possession

Déallocation automatique de l'ancien item_, et item_ conserve l'espace mémoire du paramètre item

Déallocation automatique à la destruction

```
int main() {  
    MaClasse a;  
    unique_ptr<Item> item = make_unique<Item>();  
    a.posseder(move(item));  
    a.posseder(make_unique<Item>());  
}
```

Transfert explicite; item est nullptr après

move non nécessaire car objet temporaire

shared_ptr

- Possession de la mémoire est partagée entre différents pointeurs intelligents
 - Compteur de pointeurs intelligents qui possèdent la mémoire dynamique qui est obtenu à l'aide de la fonction `use_count()`
 - **Mémoire dynamique libérée lorsque le dernier pointeur intelligent est détruit** (compte à 0)
- Un transfert de possession de mémoire est possible en utilisant la fonction `std::move()` entre deux `shared_ptr`

```
int main() {  
    shared_ptr<Item> item_1 = make_shared<Item>();  
    shared_ptr<Item> item_2 = item_1;  
    shared_ptr<Item> item_3 = move(item_1);  
}
```

Partage la possession (+1)

Transfert la possession → rien à compter;
item_1 devient nullptr

shared_ptr – Partage de la possession

```
int main() {
    shared_ptr<Item> item_1 = make_shared<Item>();
    cout << item_1.use_count() << endl; //1
    shared_ptr<Item> item_2 = item_1;
    cout << item_2.use_count() << endl; //2
    shared_ptr<Item> item_3 = move(item_1);
    cout << item_3.use_count() << endl; //2
}
```

Bonnes pratiques en C++ (suite)

- On devrait: (C++ core guidelines)
 - Utiliser unique_ptr et shared_ptr pour représenter la possession mémoire. (R.20)
 - Préférer unique_ptr à shared_ptr sauf si besoin de partager la possession. (R.21)
 - Prendre en paramètre à une fonction un pointeur intelligent si et seulement si pour changer la durée de vie de l'espace mémoire. (R.30)
 - Utiliser une simple référence ($T\&$) ou pointeur (T^*) si la fonction n'a pas à influencer la durée de vie.
 - Utiliser T^* si peut être nullptr; sinon préférer $T\&$.

Bonnes pratiques – Exemple

```
void parametreParReference(Item& item) {
    item.ajouterCaractere("A");
}

void parametreParPointeur(Item* item) {
    if (item != nullptr)
        item->ajouterCaractere("B");
}

int main() {
    unique_ptr<Item> item = make_unique<Item>();
    parametreParReference(*item);
    parametreParPointeur(item.get());
}
```

unique_ptr ou shared_ptr ne change rien au reste du code de cet exemple

Résumé

- Un pointeur intelligent est une classe générique.
- La mémoire dynamique allouée est détruite quand le pointeur intelligent qui possède cette mémoire n'y pointe plus (détruit ou réaffecté).
- Cette mémoire dynamique peut ou ne pas être partagée.
- move() pour le transfert de possession de mémoire à un autre pointeur intelligent.

Programmation orientée objet

Composition

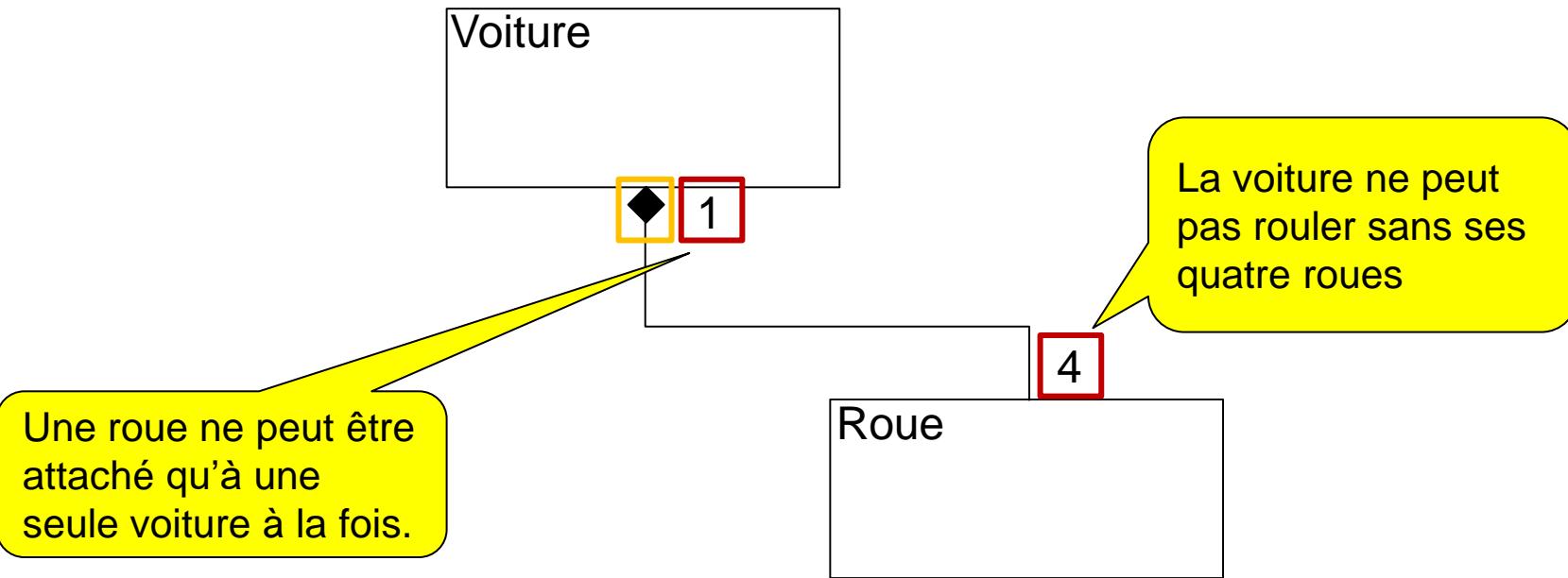
Composition

- La composition est une relation de possession (relation “**a un**” ou “**est composé de**”) entre un objet composite (objet englobant) et sa ou ses composantes (objet(s) englobé(s))
- Par exemple:
 - Une voiture a quatre roues
 - Un ordinateur a une carte-mère
 - Un train a des sièges
 - Un humain a un cœur

Composition

- Il s'agit d'une relation forte, l'objet englobant gère la mémoire de ses composantes
- Lorsque l'objet englobant est détruit, ses composantes sont automatiquement détruites, leur **durée de vie est dépendante** de celle de l'objet englobant
- Un objet englobé n'appartient qu'à un seul objet composite à la fois, l'objet composite est propriétaire unique de ses composantes

Composition par valeurs



Composition par valeurs

- Il y a composition par valeurs lorsque l'objet englobant a absolument besoin de l'objet englobé pour exister
- Si un objet A est un attribut de l' objet B, le **constructeur de l' objet A sera appelé avant celui de l' objet B.**
- Si vous réfléchissez bien, ceci est logique: pour construire une voiture, il faut d' abord construire ses composantes, comme le moteur et les roues.

Composition par valeurs – Exemple avec Voiture

```
class Voiture {  
public:  
    Voiture(): roues_{ make_unique<Roue[]>(4) } {}  
  
private:  
    unique_ptr<Roue[]> roues_;  
};  
  
int main() {  
    Voiture voiture;  
}
```

Les quatres roues sont construites par défaut avant même que l'objet voiture soit construit

La classe voiture est responsable d'allouer la mémoire pour les roues

Lorsque l'objet voiture est détruit, les quatre roues sont aussi détruites

Composition par valeurs – Exemple avec Company

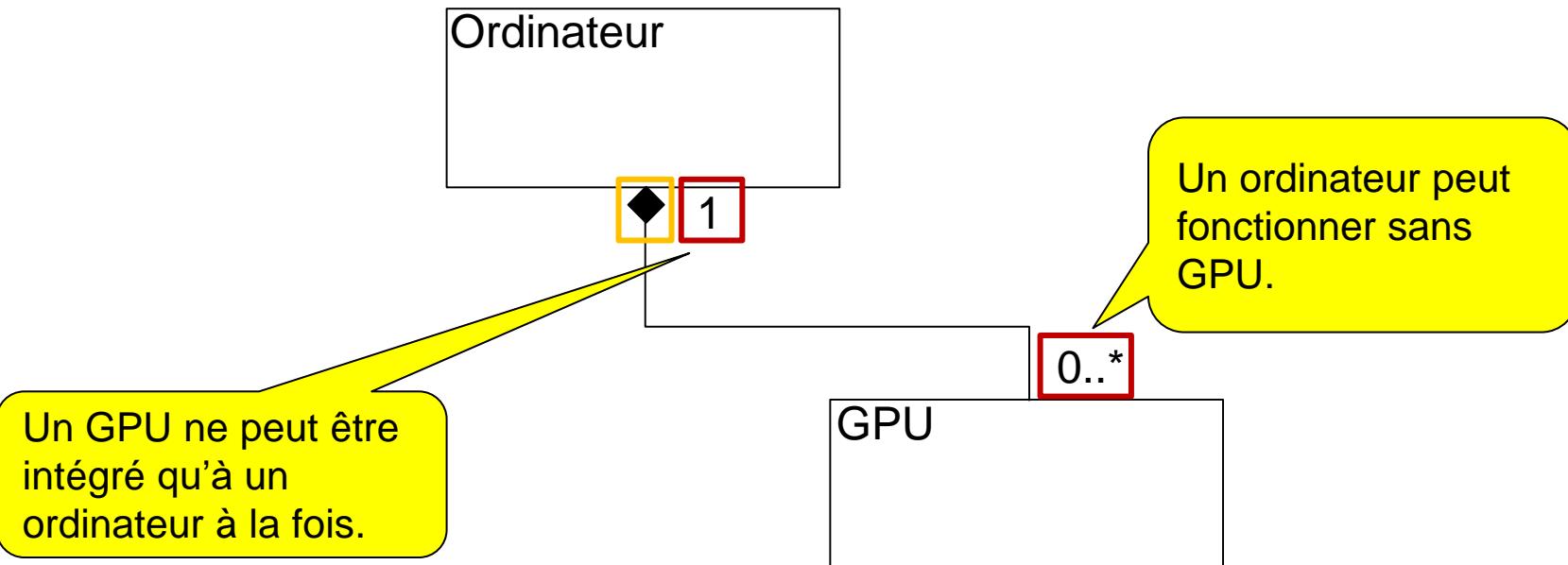
```
class Company {  
public:  
    Company() {}  
    Company(string nomPresident): president_(nomPresident) {}  
  
private:  
    Employee president_;  
};  
  
int main() {  
    Company comp;  
    Company poly("Bob");  
}
```

L'objet `president_` est construit en utilisant le constructeur par défaut de `Employee`

L'objet `president_` est construit en utilisant le constructeur par paramètres de `Employee`

Lorsque les objets de la classe `Company` sont détruits, leur président est aussi détruit

Composition par pointeurs



Composition par pointeurs

- Il y a composition par pointeurs lorsque l'objet englobant **n'a pas besoin** de l'objet englobé pour exister
- Cette relation est généralement implémentée en utilisant des **pointeurs intelligents uniques**
- L'objet englobant est donc le propriétaire unique de ses composantes et est responsable de l'allocation et de la désallocation de leur mémoire

Composition par pointeurs – Exemple avec Ordinateur

```
class Ordinateur {  
public:  
    void ajouterGPU(string fabriquant) {  
        gpus_.push_back(make_unique<Gpu>(fabriquant));  
    }  
private:  
    vector<unique_ptr<Gpu>> gpus_;  
};  
int main() {  
    Ordinateur ordi;  
    ordi.ajouterGPU("Nvidia");  
}
```

L'ordinateur alloue l'espace mémoire et crée le GPU en utilisant le constructeur par paramètres

L'ordinateur est créé sans GPU.

Lorsque l'objet ordi est détruit, tous ses GPUs sont aussi détruits

Programmation orientée objet

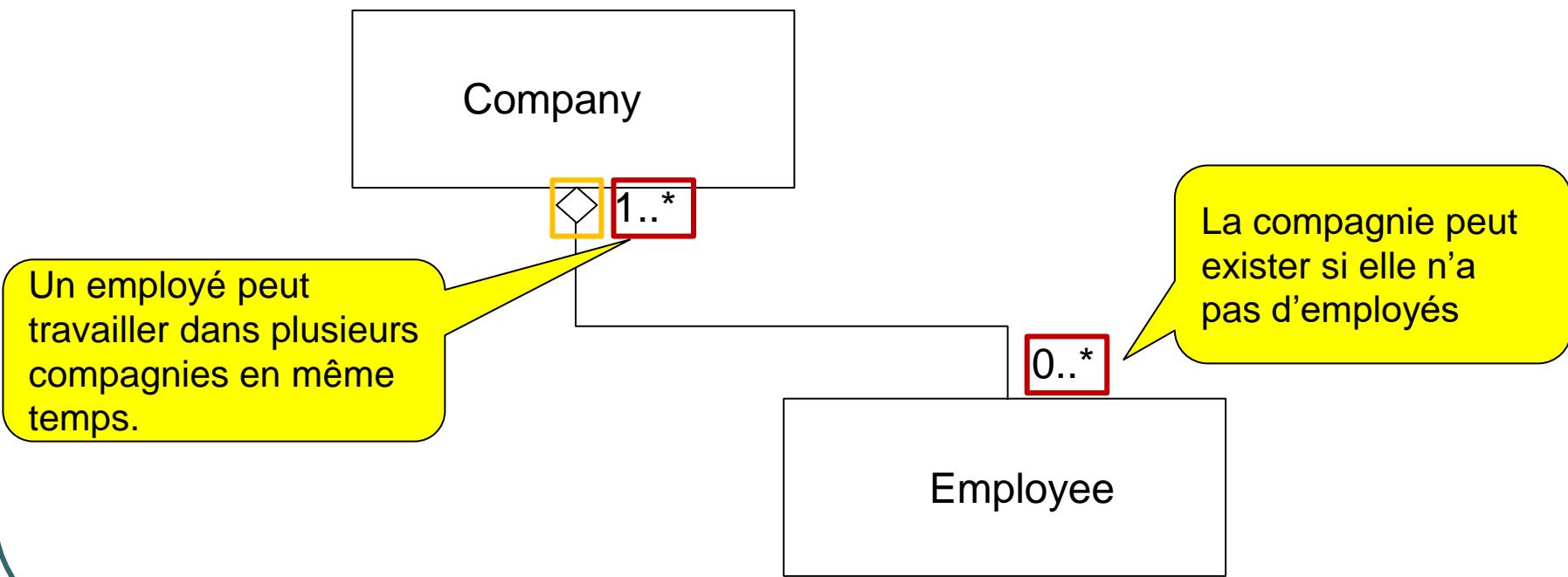
Agrégation

Agrégation

- L'agrégation consiste essentiellement en une utilisation (relation “**utilise un**”) d'un objet comme faisant partie d'un autre objet
- L'objet englobé par agrégation n'est pas détruit lorsqu'il n'est plus utilisé par un agrégat, sa **durée de vie est indépendante** de celle de l'objet englobant
- Un objet peut être utilisé par plusieurs agrégats en même temps, il peut donc y avoir partage de mémoire entre les objets englobants

Agrégation par pointeurs

- Un objet de la classe Company **utilise** des objets de la classe Employee **au besoin**



Agrégation par pointeurs

- Il y a agrégation par pointeurs lorsque l'objet englobant n'a pas besoin de l'objet englobé pour exister.
- Cette relation est généralement implémentée en utilisant des **pointeurs intelligents partagés**
- Il n'y a aucune allocation et désallocation de mémoire qui se fait lors d'une agrégation par pointeurs

Agrégation par pointeurs – Exemple implémentation

```
class Company {
public:
    Company(): president_(nullptr) {}
    Company(const shared_ptr<Employee>& emp):
        president_(emp) {}

    void ajouterEmployee(const shared_ptr<Employee>& emp){
        employees_.push_back(emp);
    }

private:
    vector<shared_ptr<Employee>> employees_;
    shared_ptr<Employee> president_;
};
```

Agrégation par pointeurs – Exemple implémentation

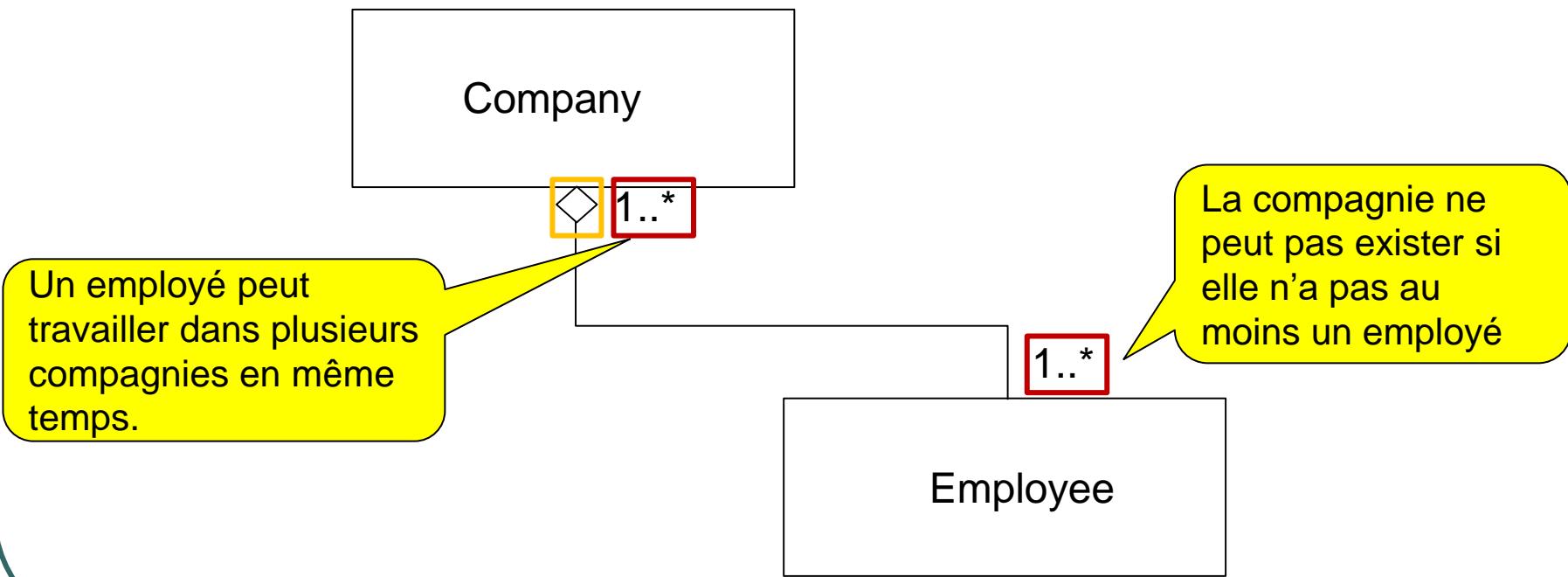
```
int main() {  
    shared_ptr<Employee> bob = make_shared<Employee>("Bob", 10000);  
    Company poly;  
  
    poly.ajouterEmployee(bob);  
}
```

Lorsqu'on ajoute bob à poly, son compteur de référence est incrémenté

bob est créé par le main et donc sa durée de vie est indépendante des compagnies qui l'utilisent

Agrégation par référence

- Un objet de la classe Company **utilise au moins un** objet de la classe Employee



Agrégation par référence

- Il y a agrégation par référence lorsque l'objet englobant **a absolument besoin** de l'objet englobé pour exister.
- La durée de vie de l'objet englobé reste indépendante de celle de l'objet englobant, mais elle doit être plus grande que celle de l'objet englobant

Agrégation par référence

- Lorsqu'on fait une agrégation par référence,
la référence doit absolument être initialisée avant même que l'objet englobant soit construit
- La référence doit donc absolument être **initialisée dans la liste d'initialisation** sinon il y aura erreur de compilation

Agrégation par référence

```
class Company {  
public:  
    Company(Employee& emp): president_(emp) {}  
  
private:  
    Employee& president_;  
};  
  
int main() {  
    Employee bob("Bob", 10000);  
    Company poly(bob);  
}
```

La classe Company
ne peut pas avoir de
constructeur par
défaut

L'attribut interne
sera une référence à
cet objet.

Comme l'attribut interne de l'objet **poly** réfère
au même objet, toute modification sur bob
affectera l'attribut interne et vice versa

Programmation orientée objet

Composition vs. Agrégation

Raphaël Beamonte, 2014

Composition ou agrégation?

```
class MaClasse {  
    public:  
        MaClasse();  
        ~MaClasse();  
        ...  
    private:  
        Point attribut_  
}
```

Composition ou agrégation?

```
class MaClasse {  
public:  
    MaClasse();  
    ~MaClasse();  
    ...  
private:  
    Point attribut__;  
}
```

Composition

Composition ou agrégation?

```
class MaClasse {  
    public:  
        MaClasse();  
        ~MaClasse();  
        ...  
    private:  
        Point &attribut_;  
}
```

Composition ou agrégation?

```
class MaClasse {  
public:  
    MaClasse();  
    ~MaClasse();  
    ...  
private:  
    Point &attribut_;  
}
```

Agrégation
par référence

Composition ou agrégation?

```
class MaClasse {  
public:  
    MaClasse();  
    ~MaClasse();  
    ...  
private:  
    Point *attribut_;  
}
```

Composition ou agrégation?

```
class MaClasse {  
public:  
    MaClasse();  
    ~MaClasse();  
    ...  
private:  
    Point *attribut_;  
}
```

Agrégation
par pointeur?

Composition ou agrégation?

```
class MaClasse {  
public:  
    MaClasse();  
    ~MaClasse();  
    ...  
private:  
    Point *attribut_;  
}
```

```
MaClasse::MaClasse() {  
    attribut_ = new Point[2];  
}  
  
MaClasse::~MaClasse() {  
    delete [] attribut_;  
}
```

Composition!

Composition ou agrégation?

```
class MaClasse {  
public:  
    MaClasse();  
    ~MaClasse();  
    ...  
private:  
    Point *attribut_;  
}
```

```
    MaClasse::setAttribut(  
        Point *attribut) {  
    attribut_ = attribut;  
}  
  
    MaClasse::~MaClasse() {  
}
```

Agrégation
par pointeur!

Composition ou agrégation?

Attribut	Objet	Référence	Pointeur
Type de relation	Composition	Agrégation par référence	Agrégation par pointeur <u>OU</u> Composition
Que regarder?	Définition	Définition	Définition <u>ET</u> Implémentation

Programmation orientée objet

Introduction à la surcharge des
opérateurs et fonctions

Surcharge de fonctions

- Il y a surcharge de fonction lorsqu'on définit et implémente une nouvelle fonction qui a le même nom qu'une fonction existante, mais dont:
 - Le nombre de paramètres est différent **et/ou**
 - Le type d'un ou plusieurs paramètres est différent
- Certains types sont susceptibles à la conversion automatique (« casts »), il faut donc faire bien attention

Exemples

*Attention: Le type de retour
n'est pas un critère qui satisfait
la surcharge de fonctions*

```
void f(int a) {}
```

```
void f(int a, double b) {}
```

```
void f(int a, double b, int c) {}
```

```
void f(int a, double b, string s) {}
```

```
void f(int a, double b, Fraction f) {}
```

```
void f(int a, double b, Point p) {}
```

Le nombre de paramètres est différent

Le type du dernier paramètre est différent

Surcharges d'opérateurs

- Consiste à **redéfinir la fonctionnalité d'un opérateur** tel que +, -, ou += pour une classe.
- Étant donné qu'on fait de la surcharge de fonctions, **on ne peut pas créer de nouveaux opérateurs**, on se contente de redéfinir ceux existants
- **L'ordre de priorité des opérateurs est conservé**, il faut donc bien comprendre le fonctionnement d'un opérateur avant de le surcharger

Opérateurs définis en C++

cppreference - surcharge des opérateurs

Pouvant être surchargés

+ - * / % ^ & | [] ()
~ = < > += -= *= /=
%=& ^= &= ! ++ -- ->, ,
>> << && != <= >=
|= || <<= >>= ==
new delete

Ne pouvant être surchargés

. :: ?: .* sizeof

Types d'opérateurs

- Les opérateurs ont un **nombre déterminé de paramètres** et ne peuvent pas avoir de paramètres par défaut
- Les **opérateurs binaires** acceptent deux paramètres qui sont, dans l'ordre, l'opérande de gauche et l'opérande de droite. **Les opérateurs ne sont donc pas commutatifs.** L'opérateur d'addition (+) est binaire
- Les **opérateurs unaires** acceptent un seul paramètre qui est l'objet courant. L'opérateur d'incrémentation (++) est unaire

Programmation orientée objet

Surcharge interne des opérateurs

La surcharge interne

- Il est possible de surcharger un opérateur en tant que **fonction membre**
- Dans ce cas, **l'opérande de gauche est toujours l'objet courant**
- La surcharge de l'opérateur prendra alors comme paramètre l'opérande de droite s'il y a lieu
- Ce type de surcharge doit être **priorisé lorsqu'il est possible de le faire** puisqu'elle respecte l'encapsulation de la classe

Exemple - Surcharge des opérateurs de Fraction

- On veut pouvoir faire:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = f1 + f2;  
}
```

- Un appel équivalent serait:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = f1.operator+(f2);  
}
```

Exemple - Surcharge de l'opérateur + pour Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    Fraction operator+ (const Fraction& fract) const;

private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};
```

Exemple - Surcharge de l'opérateur + pour Fraction

```
Fraction Fraction::operator+(const Fraction& fract) const {
    Fraction somme;

    somme.numerateur_ = numerateur_* fract.denominateur_ +
                        fract.numerateur_* denominateur_;

    somme.denominateur_ = denominateur_* fract.denominateur_;

    somme.simplifier();

    return somme;
}
```

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = f1 + 1;  
}
```

- Un appel équivalent serait:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = f1.operator+(1);  
}
```

Exemple - Surcharge de l'opérateur + pour Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    Fraction operator+ (const Fraction& fract) const;
    Fraction operator+ (const long& entier) const;

private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};
```

La signature est la même, sauf pour le type de l'opérande de droite

Exemple - Surcharge de l'opérateur ++ de Fraction

- On veut pouvoir faire:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    ++f1;  
}
```

- Un appel équivalent serait:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f1.operator++();  
}
```

Exemple - Surcharge des opérateurs de Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    Fraction operator+(const Fraction& fract) const;
    Fraction operator+(const long& entier) const;
    Fraction& operator++();
```

private:

```
    long numerateur_;
    long denominateur_;
    void simplifier();
```

};

Parce qu'on modifie l'état de l'objet et qu'on retourne une référence vers celui-ci

Exemple - Surcharge des opérateurs de Fraction

```
Fraction& Fraction::operator++() {  
    *this = *this + 1;  
    return *this;  
}
```

Cette façon est plus facile à comprendre et donc préférable à l'autre implémentation

```
Fraction& Fraction::operator++() {  
    *this = operator+(1);  
    return *this;  
}
```

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = 1 + f1;  
}
```

- Un appel équivalent serait:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = 1.operator+(f1);  
}
```

On peut seulement surchargé les opérateurs pour les classes que l'on définit

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = 1 + f1;  
}
```

- Un appel équivalent serait:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = operator+(1, f1);  
}
```

On doit surcharger l'opérateur + de manière externe

Programmation orientée objet

Surcharge externe des opérateurs

Surcharge externe

- Il est possible de surcharger un opérateur en tant que **fonction globale**
- On surcharge les opérateurs de manière externe **lorsque l'opérande de gauche n'est pas un objet de la classe courante**
- Lorsque l'opérateur est surchargé en fonction globale standard, toutes les fonctions nécessaires pour manipuler l'objet doivent être définies dans l'interface de sa classe puisque la fonction n'est pas membre

Exemple - Surcharge de l'opérateur + de Fraction

- On veut pouvoir faire:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = 1 + f1;  
}
```

- Un appel équivalent serait:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    f3 = operator+(1, f1);  
}
```

Exemple – Surcharge de l'opérateur + de Fraction

```
class Fraction{  
public:  
    Fraction operator+(const Fraction& fract) const;  
    Fraction operator+(const long& entier) const;
```

```
private:  
    long numerateur_;  
    long denominateur_;  
    void simplifier();  
};
```

Parce qu'une fonction globale ne peut pas être constante

```
Fraction operator+(const long&, const Fraction&);
```

```
Fraction operator+(const long& e, const Fraction& f) {  
    return (f + e);  
}
```

L'opérateur + fait partie de l'interface de Fraction

Exemple - Surcharge de l'opérateur << de Fraction

- On veut pouvoir faire:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    cout << f1 << endl;  
}
```

- Un appel équivalent serait:

```
int main() {  
    Fraction f1(1, 2), f2(4, 5), f3;  
    operator<<(cout, f1) << endl;  
}
```

Exemple - Surcharge de l'opérateur << de Fraction

```
class Fraction{  
public:  
    Fraction();  
    Fraction(const double& num, const double& denum);  
  
private:  
    long numerateur_;  
    long denominateur_;  
    void simplifier();  
};  
  
ostream& operator<<(ostream& o, const Fraction& f) {  
    return o << f.getNumerateur() << "/" <<  
        f.getDenominateur();  
}
```

Non, on surcharge l'opérateur
<< en fonction friend

Fonction friend

- Une fonction friend est une **fonction globale** qui a un **accès privilégié aux membres privés** d'une classe
- Ce concept ne détruit pas l'encapsulation puisque:
 - une **classe contrôle** quelle fonctions deviennent friend
 - les opérateurs font partie de l'interface et alors doivent être inclus dans la définition de la classe
 - c'est plus fiable qu'ajouter des accesseurs publiques juste pour l'opérateur

Exemple - Surcharge de l'opérateur << de Fraction

```
class Fraction{
public:
    Fraction();
    Fraction(const double& num, const double& denum);

    friend ostream& operator<<(ostream& o, const Fraction& f);
private:
    long numerateur_;
    long denominateur_;
    void simplifier();
};

ostream& operator<<(ostream& o, const Fraction& f) {
    return o << f.numerateur_ << "/" << f.denominateur_;
}
```

Parce qu'une fonction globale
ne peut pas être constante

Programmation orientée objet

Le constructeur de copie &
opérateur =

Motivation

```
void f(Employee p) {}
```

```
int main() {
    Employee e1;//Constructeur par défaut
    Employee e2("Marc", 15000);//Constructeur par
    paramètres
    Employee e3(e2); //???
    Employee e4 = e2; //???
    f(e4); //???
    e3 = e1; //???
}
```

Construction d'un **nouvel objet** en utilisant le **constructeur de copie**

Copie d'un objet dans un autre **objet existant** en utilisant l'**opérateur d'affectation** (=)

Constructeur de copie

- Lorsqu'on fait une **copie** d'un objet, il faut créer un **nouvel objet** qui sera utilisé dans la fonction
- On utilisera donc un constructeur lors de la création de ce nouvel objet
- **Ce constructeur recevra comme paramètre un autre objet de la même classe**, soit celui qu'on doit copier

Constructeur de copie (suite)

- Si on ne définit pas ce constructeur de copie, C++ utilisera un constructeur de copie par défaut, qui copie tout simplement les attributs (“**shallow copy**”). Ce constructeur est correct lorsque l’objet englobant est un **agrégat** ou un **composite par valeur**.
- Par contre, lorsque l’objet englobant est un **composite par pointeurs**, une **copie en profondeur** est nécessaire. Il faut alors **définir un constructeur de copie** qui fera la copie en profondeur.

Constructeur de copie & Agrégation

```
class Company {  
public:  
    Company(const Company& c):  
        employees_(c.employees_),  
        president_(c.president_) {}  
  
private:  
    vector<shared_ptr<Employee>> employees_;  
    Employee& president_;  
};
```

Remarque: des fonctions de la même classe peuvent toujours accéder aux attributs privés des autres objets de cette classe!

Constructeur de copie – Composition par valeurs

```
class Company {  
public:  
    Company(const Company& c):  
        president_(c.president_) {}  
private:  
    Employee president_;  
};
```

Constructeur de copie – Composition par pointeurs

```
class Company {  
public:  
    Company(const Company& c) :  
        employees_(c.employees_) {  
            president_ = make_unique<Employee>(*c.president_);  
    }  
  
private:  
    vector<shared_ptr<Employee>> employees_;  
    unique_ptr<Employee> president_;  
};
```

Opérateur =

- Ce que nous venons de dire pour la copie d'un objet vaut aussi pour l'opérateur =:

```
int main() {  
    Company c1;  
    Company c2;  
  
    c1 = c2;  
}
```

Ici aussi une copie attribut par attribut sera effectuée, à moins qu'on ne redéfinisse l'opérateur =, ce qui, évidemment, doit être fait pour la classe Company, comme on a dû le faire pour le constructeur de copie.

Définition et d'implémentation de l' opérateur =

```
class Company {  
public:  
    Company& operator=(const Company& company) {  
        if (this != &company) {  
            employees_ = company.employees_;  
            president_ = make_unique<Employee>(*company.president_);  
        }  
        return *this;  
    }  
private:  
    vector<shared_ptr<Employee>> employees_;  
    unique_ptr<Employee> president_;  
};
```

Pour éviter l'auto-affectation

On retourne une référence à l' objet parce que l' opérateur = peut être appelé en cascade: **c1 = c2 = c3** (qui est équivalent à **c1 = (c2 = c3)**)

Résumé

- Lorsqu'on définit une classe en C++, il faut toujours penser à définir les items suivants si leur définition par défaut n'est pas adéquate:
 - Le constructeur par défaut
 - Le constructeur de copie
 - L'opérateur =
 - Le destructeur

Programmation orientée objet

Héritage – Concepts de
base

Héritage

- Mécanisme permettant:
 - d'ajouter de nouvelles fonctionnalités à une classe existante
 - changer un peu le comportement de certaines méthodes d'une classe déjà existante
- On veut faire cela sans rien changer à la classe déjà existante
- On définira donc une nouvelle classe qui *héritera* de la classe existante. En C++, on parlera plutôt d'une classe *dérivée*

Héritage (suite)

- Une classe dérivée hérite des méthodes de la classe dont elle dérive (mais pas toujours, comme on le verra plus loin)
- Une classe dérivée peut redéfinir une méthode
- Si une classe dérivée redéfinit une méthode, c'est cette méthode redéfinie qui sera appelée pour un objet de cette classe, et non pas la méthode originale de la classe supérieure

Exemple de classe dérivée

- Rappelons-nous que la classe Employee représente un employé, dont les attributs sont son nom et son salaire
- Supposons maintenant qu'on veuille représenter un Manager, qui est un employé, mais qui en plus supervise d'autres employés

Exemple de classe dérivée

- Voyons d'abord la classe de base:

```
class Employee{
public:
    Employee(string name = "unknown", double salary = 0);
    void setSalary(double salary);
    double getSalary() const;
    string getName() const;

private:
    string name_;
    double salary_;
};
```

Exemple de classe dérivée

• Et maintenant la classe dérivée:

Pour spécifier que la classe hérite publiquement d'une classe parente(nous utiliserons toujours l'héritage public).

```
class Manager : public Employee {  
public:  
    Manager();  
    void addEmployee(Employee* employee);  
    Employee* getEmployee(string name) const;
```

On indique que **Manager** est une sous-classe de **Employee**.

```
private:  
    vector<Employee*> managedEmployees_;  
};
```

On a ajouté un attribut `managedEmployees_` à la classe Manager

En plus des méthodes de la classe **Employee**, dont on hérite, on a deux nouvelles méthodes.

Exemple de classe dérivée

- En plus des constructeurs, l'interface des deux classes contiennent alors:

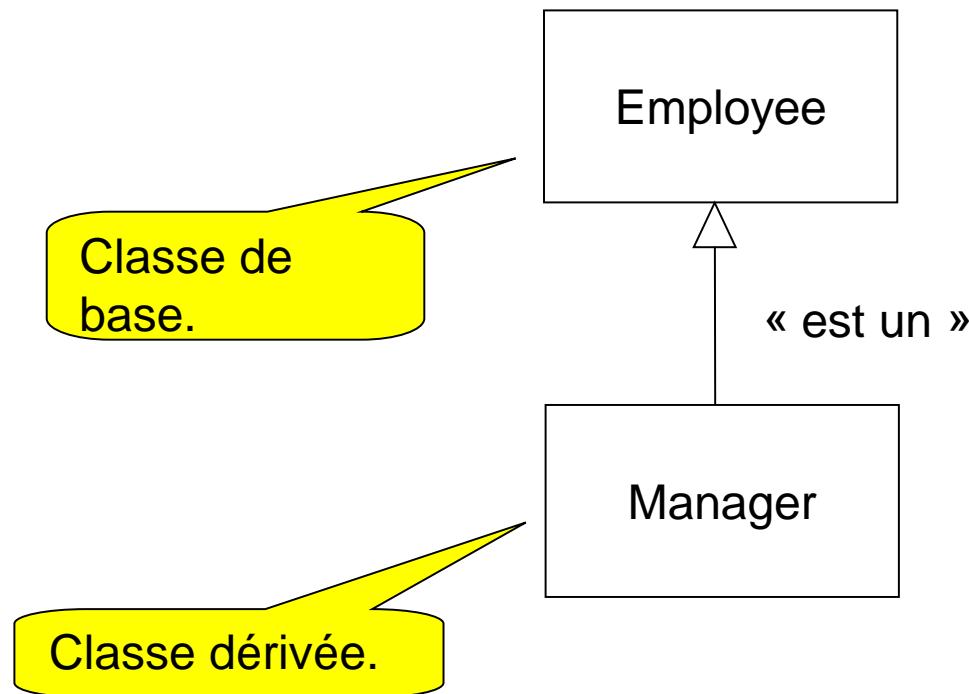
Classe	Méthodes accessibles
Employee	setSalary(); getSalary(); getName();
Manager	setSalary(); getSalary(); getName(); addEmployee() getEmployee()

Utilisation d'un objet d'une classe dérivée

- On peut utiliser les méthodes héritées tout comme les méthodes définies dans la classe dérivée:

```
int main() {  
    shared_ptr<Employee> e1 = make_shared<Employee>("John", 15000);  
    Manager e2;  
    e1->setSalary(29000); } C'est la méthode de la classe  
    e2.setSalary(48000); } Employee qui est appelée.  
    e2.addEmployee(e1); } C'est la méthode de la classe  
    Manager qui est appelée
```

Diagramme pour représenter l'héritage



Signification de l'héritage

- Attention, l'héritage est une relation de type « est un »
- Soit une classe Manager qui est une sous-classe (une classe dérivée) de la classe Employee
- Nous considérons donc que tout Manager est aussi un employé, ce qui est tout à fait conforme à l'intuition

Signification de l'héritage (suite)

- Il ne faut pas confondre l'héritage avec l'agrégation (ou la composition)
- Il pourrait être tentant d'utiliser l'agrégation (ou la composition) au lieu de l'héritage
- Du point de vue technique, le résultat peut paraître équivalent
- Mais il s'agit de deux concepts tout à fait différents (nous verrons des exemples plus loin)

Signification de l'héritage (suite)

- Prenons maintenant les classes Point et Cercle. On sait qu'un cercle a essentiellement deux attributs: un point (le centre) et un rayon
- On est donc tenté de définir Cercle comme une sous-classe de Point, dans laquelle on ajoute un attribut pour le rayon. Est-ce raisonnable?
- Pour répondre à cette question, il faut se poser la question suivante: un cercle est-il un point?

Signification de l'héritage (suite)

- Et si on faisait le contraire, c'est-à-dire définir Point comme classe dérivée de Cercle. Est-ce raisonnable?
- Quels seraient les attributs de la classe de base et la classe dérivée?

Signification de l'héritage (suite)

- Soit maintenant une classe Triangle, composée de trois points qui représentent ses sommets
- On veut maintenant définir une classe Fleche, qui est constituée d'un triangle et d'une droite perpendiculaire à un des côtés du triangle
- On pourrait être tenté de faire dériver Fleche de la classe Triangle, en y ajoutant un attribut pour représenter la droite
- Est-ce raisonnable? Une flèche est-elle un triangle?

Signification de l'héritage (suite)

- En résumé, il faut décider si une classe est liée à une autre par une relation d'héritage ou par une relation de composition (ou agrégation)
- Par exemple:
 - Un cercle **est une** forme
 - Une compagnie **utilise** des employés
 - Un ordinateur **a une** carte-mère

Exemple de l'horloge

- Soit une classe Clock, qui permet d'obtenir l'heure locale, de deux façons: à l'américaine (am/pm) ou en utilisant la norme dite « militaire » (23:45):

```
class Clock {  
public:  
    Clock(bool useMilitary);  
    string getLocation() const { return "Local"; }  
    int getHours() const;  
    int getMinutes() const;  
    bool isMilitary() const;  
private:  
    bool military_;  
};
```

Remarquez qu'il n'y a pas de constructeur par défaut.

L'unique attribut, dont la valeur doit être spécifiée lors de la construction de l'objet, détermine si l'heure sera affichée dans le format militaire ou non.

Exemple de l'horloge (suite)

```
int main() {  
    Clock horloge1(true);  
    Clock horloge2(false);  
}
```

Crée une horloge à affichage de style « militaire » (23:45)

Crée une horloge à affichage de style « américain » (11:45)

Exemple de l'horloge (suite)

- Supposons maintenant que l'on veuille créer une horloge qui donne l'heure selon une zone différente de l'heure locale
- On créera donc une classe dérivée TravelClock, que l'on construit en fournissant le nom de la zone et le décalage en méridiens par rapport à l'heure locale

Exemple de l'horloge (suite)

```
class TravelClock : public Clock {  
public:  
    TravelClock(bool mil, string loc, int diff);  
    string getLocation() const;  
    int getHours() const;  
private:  
    string location_;  
    int timeDifference_;  
};
```

Deux nouveaux attributs ajoutés.

Encore une fois, pas de constructeur par défaut.

La méthode **getHours()** étend celle de la classe de base.

Méthode dont l'implémentation est complètement différente de celle de la classe de base.

Exemple de l'horloge (suite)

```
TravelClock::TravelClock(bool mil, string loc, int diff) :  
    Clock(mil), location_(loc), timeDifference_(diff) {}
```

Étant donné que Clock n'a pas de constructeur par défaut, il faut utiliser son constructeur par paramètres

```
string TravelClock::getLocation() const {  
    return location_;  
}
```

Méthode dont l'implémentation est complètement différente de celle de la classe de base.

```
int TravelClock::getHours() const {  
    return Clock::getHours() + timeDifference_;  
}
```

La méthode **getHours()** étend celle de la classe de base.

On appelle la méthode **getHours()** de Clock pour pouvoir ajouter la différence de temps à son résultat

Exemple de l'horloge (suite)

- En plus des constructeurs, l'interface des deux classes contiennent alors:

Classe	Méthodes accessibles
Clock	getLocation() getHours() getMinutes() isMilitary()
TravelClock	getLocation() getHours() getMinutes() isMilitary()

Ces méthodes sont accessibles même si elles n'ont pas été définies dans TravelClock puisqu'elles sont héritées de Clock

Programmation orientée objet

Ordre de construction et
déconstruction

Constructeur de la classe dérivée

- Il est important de noter que le constructeur de la classe de base est appelé avant même que l'objet de la classe dérivée soit construit
- On peut donc voir un objet de la classe dérivé en deux parties:
 - La partie héritée de la classe de base provient d'un objet de celle-ci
 - La partie spécifique à la classe dérivée provient de l'objet de celle-ci

Liste de construction d'un objet

- Dans le cas d'une classe dérivée, la première chose qui est construite est un objet de sa classe de base
- L'ordre de construction sera alors le suivant:
 1. Construction d'un objet de la classe de base
 2. Construction des attributs dans l'ordre de leur définition dans la classe
 3. Construction de l'objet lui-même

Ordre d'appel des constructeurs

```
class Clock {  
public:  
    3 Clock(bool useMilitary): military_(useMilitary){ 5 }  
private:  
    bool military_; 4  
};  
class TravelClock : public Clock {  
public:  
    2 TravelClock(bool mil, string loc, int diff) : Clock(mil),  
        location_(loc), timeDifference_(diff) { 8 }  
private:  
    string location_; 6  
    int timeDifference_; 7  
    int main() {  
        1 TravelClock tclock(true,  
            "Nouveau-Brunswick", 1);  
    }  
};
```

Liste de déconstruction d'un objet

- La déconstruction d'un objet se fait dans le sens inverse de sa construction
- L'ordre de déconstruction sera alors le suivant:
 1. Déconstruction de l'objet lui-même
 2. Déconstruction de ses attributs dans l'ordre inverse de leur définition dans la classe
 3. Déconstruction de l'objet de la classe de base

Ordre d'appel des constructeurs (exemple)

```
class A           class B           class C           class D : public A
{
public:          {
public:          {
public:          {
public:          {
    A();          B();          C();          D();
    ...
}
}
}
}

private:         ...
B att_;          ...
C att_;          ...
};

int main()
{
    D objet;
    ...
}
```

Ordre d'appel des constructeurs (exemple)

```
class A          class B          class C          class D : public A
{
public:        {
public:        public:        public:        public:
    A();      B();      C();      D();
    ...
private:      ...
B att_;
};
```

```
... } } ... }
```

```
int main()
{
    D objet;
    ...
}
```

Ordre d'appel des constructeurs (exemple)

```
class A          class B          class C          class D : public A
{
public:          {
public:          public:          public:
    A();          B();          C();          D();
    ...
private:         ...
B att_;          ...
};               }               }               }
} ;               }               }               } ;
```

The diagram illustrates the call order of constructors. The constructor A() in class A is called first (indicated by a yellow circle labeled '2'). This is followed by the constructor B() in class B (indicated by a yellow circle labeled '1'). The constructors in classes C and D are not explicitly shown to be called, but they are part of the inheritance chain.

```
int main()
{
    D objet;
    ...
}
```

Ordre d'appel des constructeurs (exemple)

```
class A          class B          class C          class D : public A
{
public:          {
public:          public:          public:
    A();          B();          C();          D();
    ...
private:         ...
private:         ...
B att_;          }           }           }
};               }           }           }
```

The diagram illustrates the call order of constructors for classes A, B, C, and D. The arrows show the flow from the constructor definitions in each class to the corresponding yellow circles labeled 2, 1, and 3. These circles then point to the constructor definitions in class D.

```
int main()
{
    D objet;
    ...
}
```

Ordre d'appel des constructeurs (exemple)

```
class A
{
public:
    A();
    ...
private:
    B att_;
};
```

2

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

1

```
class C
{
public:
    C();
    ...
};
```

3

```
class D : public A
{
public:
    D();
    ...
private:
    C att_;
};
```

4

```
int main()
{
    D objet;
    ...
}
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ...  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ...  
}
```

```
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
A::A(int x): att_(x)  
{  
    ...  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
B::B(int x): x_(x)  
{  
    ...  
}  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ...  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ...  
}
```

```
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
A::A(int x): att_(x)  
{  
    ...  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
B::B(int x): x_(x)  
{  
    ...  
}  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ...  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ...  
} 1  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ... 2  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ... 1  
}  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ... ②  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ... ①  
}  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ... ②  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ... ①  
}  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    }  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ... ②  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ...  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ... ①  
}  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    } ③  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (second exemple)

```
class A {  
public:  
    A();  
    A(int x);  
    ...  
private:  
    B att_;  
};  
  
A::A(int x): att_(x)  
{  
    ... ②  
}  
  
int main()  
{  
    D objet(3,2);  
    ...  
}
```

```
class D : public A {  
public:  
    D();  
    D(int p, int q);  
    ...  
private:  
    C att_;  
    ...  
};  
  
D::D(int p, int q)  
    : att_(p), A(q)  
{  
    ... ④  
}
```

```
class B{  
public:  
    B();  
    B(int x);  
private:  
    int x_;  
};  
  
B::B(int x): x_(x)  
{  
    ... ①  
}  
  
class C{  
public:  
    C(int x): x_(x) {  
        ...  
    } ③  
private:  
    int x_;  
};
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {
public:
    Employee();
    Employee(string name);
    Employee(string name, double salary);
    /* ... */
private:
    string name_;
    double salary_;
};

int main() {
    Manager m1;
    Manager m2("Jenny", 12000);
    Manager m3("Jenny");

    return 0;
}

class Manager : public Employee {
public:
    Manager();
    Manager(string name);
    Manager(string name, double salary);
    Manager(string name, double salary,
            double bonus);
    /* ... */
private:
    double bonus_;
    vector<Employee*> managedEmployees_;
};

Manager::Manager() : Employee(), bonus_(15) {}

Manager::Manager(string name)
: Employee(name), bonus_(15) {}

Manager::Manager(string name, double salary)
: Employee(name, salary), bonus_(15) {}

Manager::Manager(string name, double salary,
                 double bonus)
: Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee();  
    Employee(string name);  
    Employee(string name, double salary);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
}  
  
int main() {  
    Manager m1;  
    Manager m2("Jenny", 12000);  
    Manager m3("Jenny");  
  
    return 0;  
}
```

Que se passe-t-il pour
m1 ?

```
class Manager : public Employee {  
public:  
    Manager();  
    Manager(string name);  
    Manager(string name, double salary);  
    Manager(string name, double salary,  
            double bonus);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
}  
  
Manager::Manager() : Employee(), bonus_(15) {}  
  
Manager::Manager(string name)  
: Employee(name), bonus_(15) {}  
  
Manager::Manager(string name, double salary)  
: Employee(name, salary), bonus_(15) {}  
  
Manager::Manager(string name, double salary,  
                double bonus)  
: Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee();  
    Employee(string name);  
    Employee(string name, double salary);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
}  
  
int main() {  
    Manager m1;  
    Manager m2("Jenny", 12000);  
    Manager m3("Jenny");  
  
    return 0;  
}
```

Que se passe-t-il pour
m1 ?

```
class Manager : public Employee {  
public:  
    Manager();  
    Manager(string name);  
    Manager(string name, double salary);  
    Manager(string name, double salary,  
            double bonus);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
}  
  
Manager::Manager() : Employee(), bonus_(15) {}  
  
Manager::Manager(string name)  
: Employee(name), bonus_(15) {}  
  
Manager::Manager(string name, double salary)  
: Employee(name, salary), bonus_(15) {}  
  
Manager::Manager(string name, double salary,  
                double bonus)  
: Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee();  
    Employee(string name);  
    Employee(string name, double salary);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
};  
  
int main() {  
    Manager m1;  
    Manager m2("Jenny", 12000);  
    Manager m3("Jenny");  
  
    return 0;  
}
```

Que se passe-t-il pour
m1 ?

```
class Manager : public Employee {  
public:  
    Manager();  
    Manager(string name);  
    Manager(string name, double salary);  
    Manager(string name, double salary,  
            double bonus);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
};  
  
Manager::Manager() : Employee(), bonus_(15) {}  
  
Manager::Manager(string name)  
: Employee(name), bonus_(15) {}  
  
Manager::Manager(string name, double salary)  
: Employee(name, salary), bonus_(15) {}  
  
Manager::Manager(string name, double salary,  
                double bonus)  
: Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee();  
    Employee(string name);  
    Employee(string name, double salary);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
}  
  
int main() {  
    Manager m1;  
    Manager m2 ("Jenny", 12000);  
    Manager m3 ("Jenny");  
  
    return 0;  
}
```

Que se passe-t-il pour
m2 ?

```
class Manager : public Employee {  
public:  
    Manager();  
    Manager(string name);  
    Manager(string name, double salary);  
    Manager(string name, double salary,  
            double bonus);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
}  
  
Manager::Manager() : Employee(), bonus_(15) {}  
  
Manager::Manager(string name)  
: Employee(name), bonus_(15) {}  
  
Manager::Manager(string name, double salary)  
: Employee(name, salary), bonus_(15) {}  
  
Manager::Manager(string name, double salary,  
                double bonus)  
: Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee();  
    Employee(string name);  
    Employee(string name, double salary);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
};  
  
int main() {  
    Manager m1;  
    Manager m2 ("Jenny", 12000);  
    Manager m3 ("Jenny");  
  
    return 0;  
}
```

Que se passe-t-il pour
m2 ?

```
class Manager : public Employee {  
public:  
    Manager();  
    Manager(string name);  
    Manager(string name, double salary);  
    Manager(string name, double salary,  
            double bonus);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
};  
  
Manager::Manager() : Employee(), bonus_(15) {}  
  
Manager::Manager(string name)  
: Employee(name), bonus_(15) {}  
  
Manager::Manager(string name, double salary)  
: Employee(name, salary), bonus_(15) {}  
  
Manager::Manager(string name, double salary,  
                double bonus)  
: Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee();  
    Employee(string name);  
    Employee(string name, double salary);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
}  
  
int main() {  
    Manager m1;  
    Manager m2 ("Jenny", 12000);  
    Manager m3 ("Jenny");  
  
    return 0;  
}
```

Que se passe-t-il pour
m3 ?

```
class Manager : public Employee {  
public:  
    Manager();  
    Manager(string name);  
    Manager(string name, double salary);  
    Manager(string name, double salary,  
            double bonus);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
}  
  
Manager::Manager() : Employee(), bonus_(15) {}  
  
Manager::Manager(string name)  
: Employee(name), bonus_(15) {}  
  
Manager::Manager(string name, double salary)  
: Employee(name, salary), bonus_(15) {}  
  
Manager::Manager(string name, double salary,  
                double bonus)  
: Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee();  
    Employee(string name);  
    Employee(string name, double salary);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
}  
  
int main() {  
    Manager m1;  
    Manager m2("Jenny", 12000);  
    Manager m3("Jenny");  
  
    return 0;  
}
```

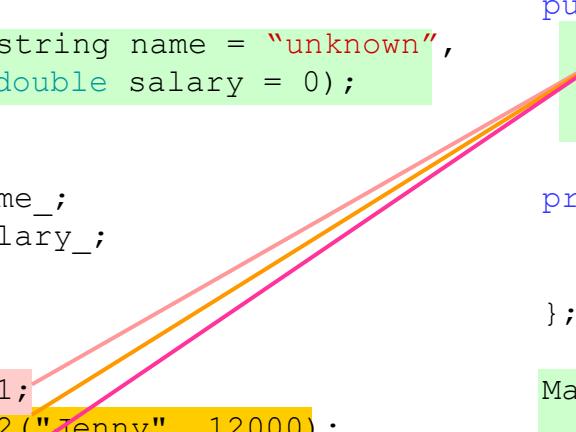
Que se passe-t-il pour
m3 ?

```
class Manager : public Employee {  
public:  
    Manager();  
    Manager(string name);  
    Manager(string name, double salary);  
    Manager(string name, double salary,  
           double bonus);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
};  
  
Manager::Manager() : Employee(), bonus_(15) {}  
  
Manager::Manager(string name)  
    : Employee(name), bonus_(15) {}  
  
Manager::Manager(string name, double salary)  
    : Employee(name, salary), bonus_(15) {}  
  
Manager::Manager(string name, double salary,  
                 double bonus)  
    : Employee(name, salary), bonus_(bonus) {}
```

Ordre d'appel des constructeurs (troisième exemple)

```
class Employee {  
public:  
    Employee(string name = "unknown",  
             double salary = 0);  
    /* ... */  
private:  
    string name_;  
    double salary_;  
};  
  
int main() {  
    Manager m1;  
    Manager m2("Jenny", 12000);  
    Manager m3("Jenny");  
  
    return 0;  
}
```

```
class Manager public Employee {  
public:  
    Manager(string name = "unknown",  
            double salary = 0,  
            double bonus = 15);  
    /* ... */  
private:  
    double bonus_;  
    vector<Employee*> managedEmployees_;  
};  
  
Manager::Manager(string name, double salary,  
                 double bonus)  
: Employee(name, salary), bonus_(bonus) {}
```



Bien sûr, on peut, dans certains cas comme le notre, se passer de plusieurs constructeurs en utilisant les valeurs par défaut...

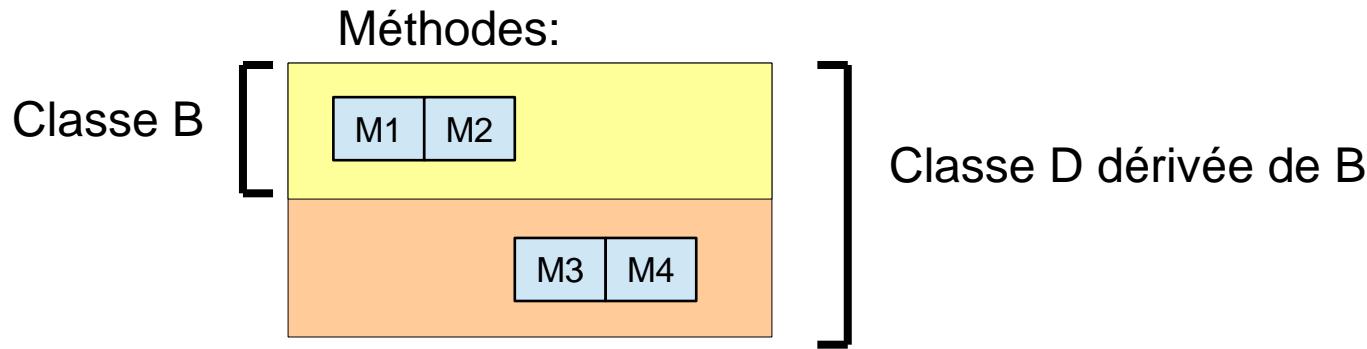
Programmation orientée objet

Appel de méthodes et
accès aux membres

Appel d'une méthode d'une classe dérivée

- Si une classe de base définit et implémente une méthode, la classe dérivée peut:
 - Redéfinir cette méthode en:
 - Lui attribuant un nouveau comportement qui est complètement différent de celui de la classe de base
 - Étendant le comportement de la méthode de la classe de base
 - Hériter de la méthode qui a été définie dans la classe de base

Appel d'une méthode

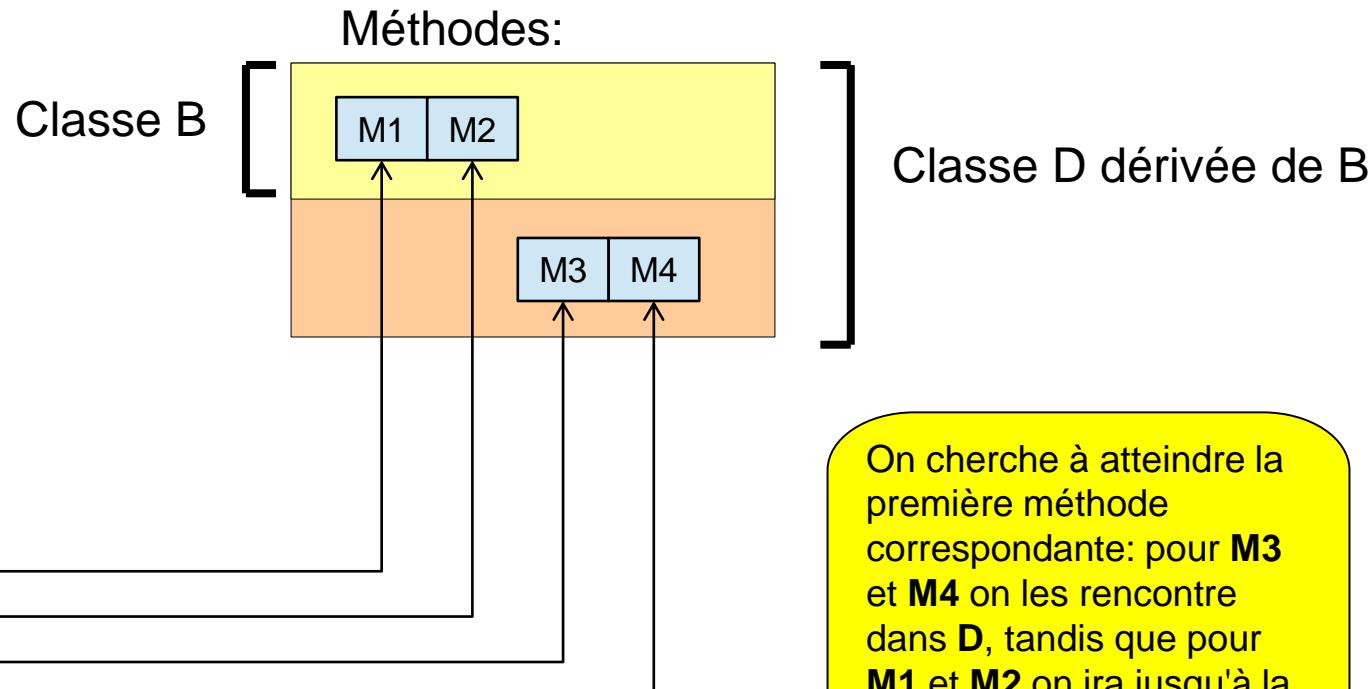


```
int  
D d;
```

Comment fonctionne l'appel
d'une méthode sur un objet
d'une classe dérivée ?!

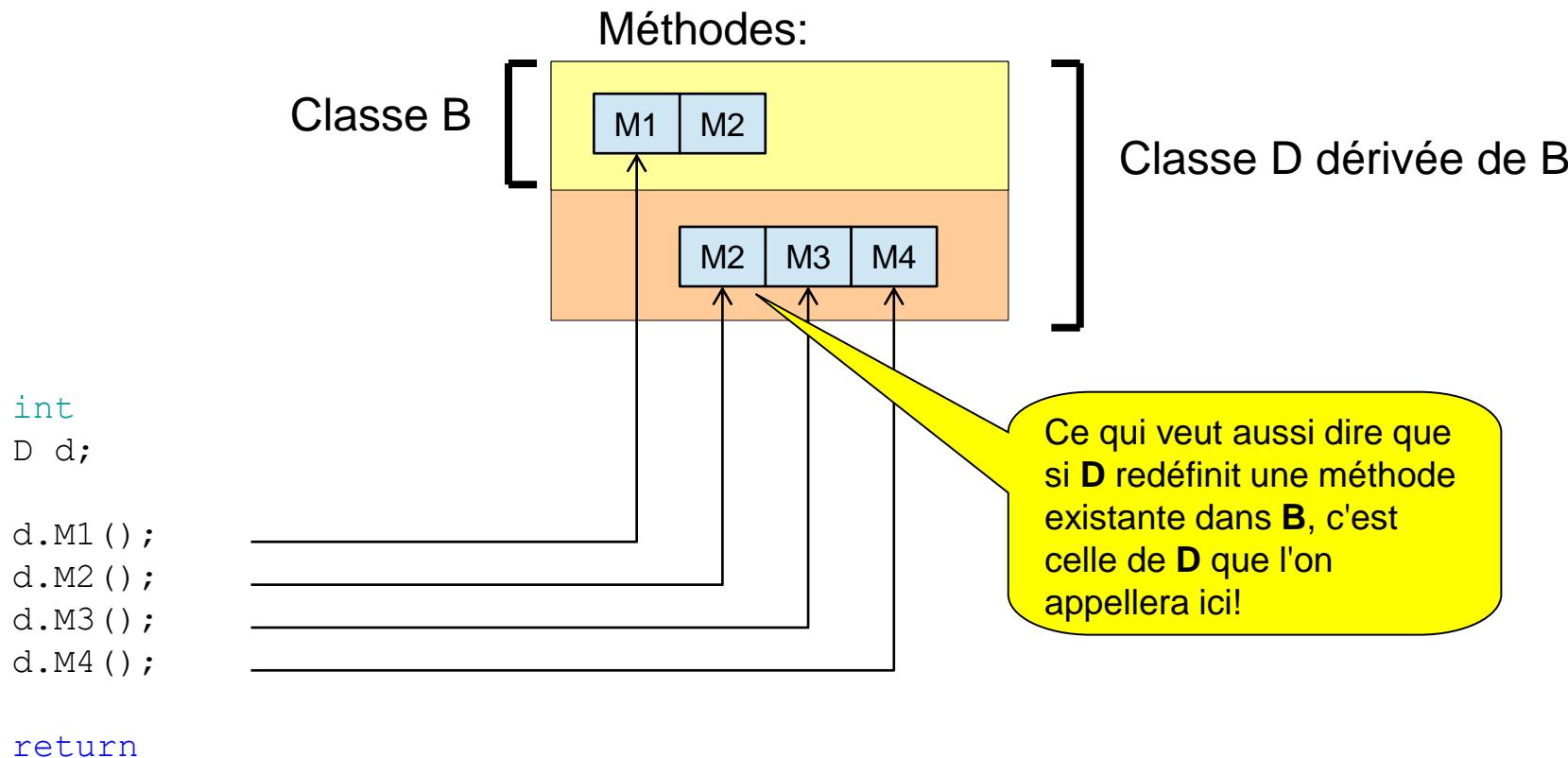
```
return
```

Appel d'une méthode



On cherche à atteindre la première méthode correspondante: pour **M3** et **M4** on les rencontre dans **D**, tandis que pour **M1** et **M2** on ira jusqu'à la classe de base **B**

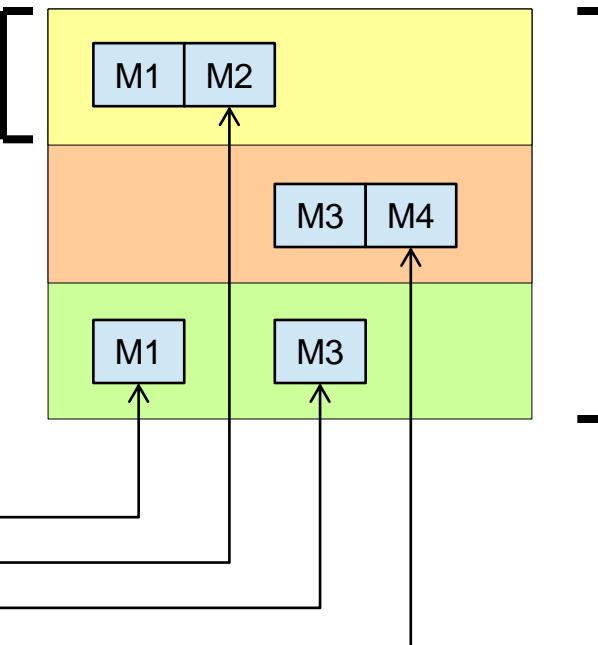
Appel d'une méthode



Appel d'une méthode

Classe D dérivée de B

Méthodes:



Classe E dérivée de D

```
int  
E e;  
  
e.M1 ();  
e.M2 ();  
e.M3 ();  
e.M4 ();  
  
return
```

On reste dans la même idée lorsqu'il y a plusieurs niveau d'héritage, comme ici avec **E** qui hérite de **D**, qui lui-même hérite de **B**.

Accès aux membres d'une classe de base

- Tout membre **privé** est inaccessible non seulement à l'extérieur d'une classe, mais aussi à ses classes dérivées
- Pour qu'un membre soit accessible aussi par ses classes dérivées, on le déclare comme **protégé**
- Un attribut est toujours privé. Pour qu'une classe dérivée puisse accéder à un attribut de la classe de base, on lui fournira des méthodes d'accès protégées, s'il n'en existe pas déjà qui sont publiques

Autre exemple de classe dérivée Gerant

```
class Manager : public Employee {  
public:  
    Manager();  
    void addEmployee(const shared_ptr<Employee>& employee);  
    Employee* getEmployee(string name) const;  
    double getSalary() const {  
        double baseSalary = Employee::getSalary();  
        return (baseSalary + (1 + bonus_ / 100.0))  
    }  
  
private:  
    vector<shared_ptr<Employee>> managedEmployees_;  
    double bonus_;  
};
```

Comme l'attribut **salary_** est privé dans la classe **Employee**, il faut utiliser la méthode de cette classe pour y accéder.

Accès aux membres pour une classe C

public:

protected:

private:

Accessible
à la classe C
seulement:

class C

Accessible seulement
à la classe C et toutes
les classes qui
en dérivent:

class C

class X : public C

class Y : public C

class Z : public C

...

Accessible
à tout le monde:

class A

class B

class C

...

Accès aux membres pour une classe C

public:

protected:

private:

Accessible à la classe C seulement:

class C

Accessible seulement à la classe C et toutes les classes qui en dérivent:

class C

class X : public C

class Y : public C

class Z : public C

...

private et protected sont aussi accessibles aux classes et fonctions friend

Accessible à tout le monde:

class A

class B

class C

...

Accès aux membres pour une classe C (exemple)

```
class A {  
public:  
    A();  
    ~A();  
    int getAtt1() const;  
    void setAtt1(int x);  
protected:  
    int getAtt2() const;  
    void setAtt2(int x);  
private:  
    int att1_;  
    int att2_;  
};
```

Interface accessible à tout le monde

Accessibles à la classe A, ses classes dérivées et aux classes et fonctions amies

Accessibles seulement à A et ses amies

Programmation orientée objet

Polymorphisme – Concepts
de base

Qu'est-ce que le polymorphisme?

- Le polymorphisme est un concept fondamental de la programmation orientée objet
- Il permet de prendre en compte le type réel d'un pointeur ou d'une référence lorsqu'il est manipulé au travers de l'interface d'une classe de base
- Il s'applique donc à l'exécution du programme

Méthode virtuelle

- En C++, le polymorphisme fonctionne en déclarant des méthodes virtuelles:

```
class Clock {  
public:  
    Clock(bool useMilitary) {}  
    string getLocation() const { return "Local"; }  
    virtual int getHours() const;  
    int getMinutes() const;  
    bool isMilitary() const;  
private:  
    bool military_;  
};
```

Méthode virtuelle (suite)

- Le type d'une copie d'un objet est déterminé à la compilation, **le polymorphisme ne s'applique donc pas dans le cas de copies**
- Une méthode virtuelle prend effet seulement au travers de **pointeur et de référence** d'objet puisque leur type est déterminé à l'exécution du programme

Méthode virtuelle (suite)

- Considérons le programme suivant:

```
int main() {
    Clock c1(true);
    TravelClock c2(true, "Paris", 6);
    TravelClock c3(true, "Vancouver", -3);

    vector<Clock*> clocks = {&c1, &c2, &c3};

    for (Clock* c : clocks) {
        cout << c->getHours() << ":" << c->getMinutes() << endl;
    }
}
```

Nous avons un vecteur de pointeurs de Clock, donc les méthodes déclarées virtuelles seront choisies à l'exécution

Méthode déclarée virtuelle, le choix de la méthode qui sera appelée est basé sur la classe réel de l'objet pointé

Méthode standard, la méthode appelée est déterminée à la compilation et basée sur le type du pointeur

Méthode virtuelle (suite)

- Lors de l'exécution du programme, les méthodes appelées seront:

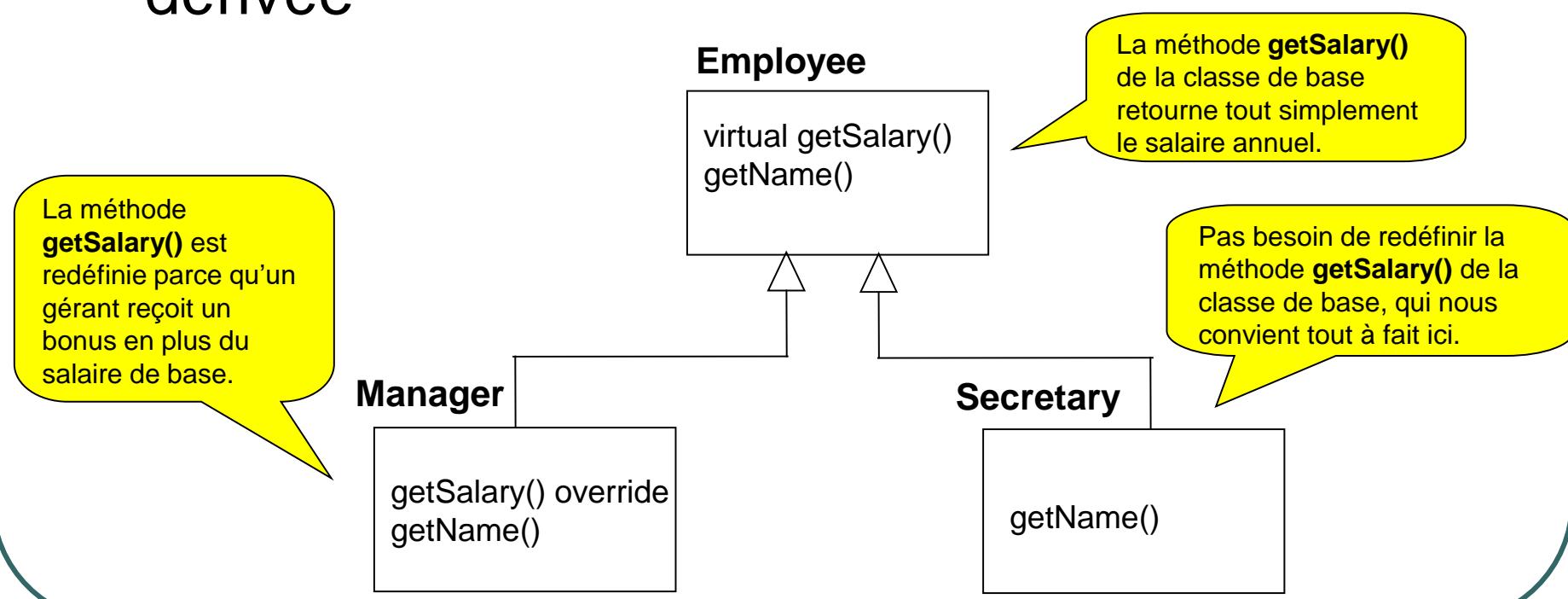
1. Clock::getHours()
 2. Clock::getMinutes()
 3. TravelClock::getHours()
 4. Clock::getMinutes
 5. TravelClock::getHours()
 6. Clock::getMinutes()
-
- Pointeur vers c1
de type Clock
- Pointeur vers c2 de
type TravelClock
- Pointeur vers c3 de
type TravelClock

Méthode virtuelle (suite)

- *Attention:* si une méthode est déclarée virtuelle dans une classe, elle le sera automatiquement dans toutes les classes qui en dérivent (**le polymorphisme est hérité**)
- Pour éviter toute confusion, on ajoute le mot clé « `override` » à la signature des méthodes virtuelles dans les classes dérivées
- L'avantage de cela est:
 - qu'on n'aura pas besoin d'aller consulter la classe de base pour savoir si une méthode est virtuelle
 - « `override` » s'assure que la méthode était déjà virtuelle dans la classe de base

Polymorphisme et méthode héritée

- Même si une méthode est virtuelle, on est libre de la redéfinir ou non dans une classe dérivée



Choix du type

- Le *type statique* est déterminé lors de la compilation
- Le *type dynamique* est déterminé seulement lors de l'exécution

```
int main() {  
    vector<unique_ptr<Employee>> employees;  
    employees.push_back(make_unique<Secretary>("Mark", 16000.0));  
    employees.push_back(make_unique<Manager>("Jenny", 12000.0, 10));  
  
    for (const unique_ptr<Employee>& emp : employees) {  
        cout << emp->getSalary() << endl;  
    }  
}
```

Types de liaison

- Il y a *liaison statique* lorsque la méthode appelée est choisie à la compilation et donc basée sur son type statique
- Il y a *liaison dynamique* lorsque la méthode appelée est choisie à l'exécution et donc basée sur son type dynamique (méthodes virtuelles)

```
int main() {
    vector<unique_ptr<Employee>> employees;
    employees.push_back(make_unique<Secretary>("Mark", 16000.0));
    employees.push_back(make_unique<Manager>("Jenny", 12000.0, 10));
    for (const unique_ptr<Employee>& emp : employees) {
        cout << emp->getSalary() << endl;
        cout << emp->getName() << endl;
    }
}
```

Exemple avec Employee et ses dérivées

```
class Employee {
public:
    Employee(string name = "unknown", double salary = 0) :
        name_(name), salary_(salary) {}
    virtual double getSalary() const {
        return salary_;
    }
    string getName() const {
        return name_;
    }
private:
    string name_;
    double salary_;
};
```

Exemple avec Employee et ses dérivées

```
class Manager : public Employee {
public:
    Manager(string name = "unknown", double salary = 0,
            double bonus = 0) : Employee(name, salary),
            bonus_(bonus) {}
    double getSalary() const override {
        return Employee::getSalary() * (1 + bonus_ / 100);
    }
    string getName() const {
        return Employee::getName() + " (Manager)";
    }
private:
    double bonus_;
};
```

Exemple avec Employee et ses dérivées

```
class Secretary : public Employee {
public:
    Secretary(string name = "unknown", double salary = 0) :
        Employee(name, salary) {}
    string getName() const {
        return Employee::getName() + " (Secretary)";
    }
};
```

Polymorphisme et méthode héritée (exemple)

```
void afficher_statique(Employee employee) {
    cout << "Name: " << employee.getName() << ", " << "Salary: "
    << employee.getSalary() << endl;
}

void afficher_dynamique(const Employee& employee) {
    cout << "Name: " << employee.getName() << ", " << "Salary: "
    << employee.getSalary() << endl;
}

void afficher_dynamique(Employee* employee) {
    cout << "Name: " << employee->getName() << ", " << "Salary:
" << employee->getSalary() << endl;
}
```

Polymorphisme et méthode héritée (exemple)

```
int main() {
    vector<unique_ptr<Employee>> employees;
    employees.push_back(make_unique<Secretary>("Mark", 16000.0));
    employees.push_back(make_unique<Manager>("Jenny", 12000.0, 10));

    for (const unique_ptr<Employee>& emp : employees) {
        afficher_statique(*emp);
        afficher_dynamique(*emp);
        afficher_dynamique(emp.get());
    }
}
```

Méthode virtuelle appelée dans une autre méthode de la classe

- On sait qu'une méthode peut appeler une autre méthode de la même classe
- Que se passe-t-il si cette autre méthode est virtuelle?
- Supposons que la classe **Employee** a une méthode **print()** qui n'est pas virtuelle et définie de la manière suivante:

```
void Employee::print(ostream& out) const
{
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
```

Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
int main() {  
    Manager manager("John", 15000, 15);  
    manager.print(cout);  
}
```

```
John  
Salary: 17250
```

- Pour répondre, considérez le code équivalent suivant:

```
void Employee::print(ostream& out) const{  
    out << this->getName() << endl;  
    out << "Salary: " << this->getSalary() << endl;  
}
```

Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
int main() {
    Secretary secretaire("John", 15000);
    secretaire.print(cout);
}
```

John
Salary: 15000

- Pour répondre, considérez le code équivalent suivant:

```
void Employee::print(ostream& out) const{
    out << this->getName() << endl;
    out << "Salary: " << this->getSalary() << endl;
}
```

Méthode virtuelle appelée dans une autre méthode de la classe

- Pour avoir le résultat désiré, il faut que **getName()** soit elle aussi déclarée virtuelle:

```
class Employee {  
public:  
    Employee(string name = "unknown", double salary = 0);  
    virtual double getSalary() const;  
    virtual string getName() const;  
    void print(ostream& out) const;  
  
private:  
    string name_;  
    double salary_;  
};
```

Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
int main() {
    Employee employe("John", 15000);
    employe.print(cout);
}
```

```
John
Salary: 15000
```

- Pour répondre, considérez le code équivalent suivant:

```
void Employee::print(ostream& out) const{
    out << this->getName() << endl;
    out << "Salary: " << this->getSalary() << endl;
}
```

Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
int main() {  
    Manager manager("John", 15000, 15);  
    manager.print(cout);  
}  
John (Manager)  
Salary: 17250
```

- Pour répondre, considérez le code équivalent suivant:

```
void Employee::print(ostream& out) const{  
    out << this->getName() << endl;  
    out << "Salary: " << this->getSalary() << endl;  
}
```

Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
int main() {
    Secretary secretaire("John", 15000);
    secretaire.print(cout);
}
```

John (Secretary)
Salary: 15000

- Pour répondre, considérez le code équivalent suivant:

```
void Employee::print(ostream& out) const{
    out << this->getName() << endl;
    out << "Salary: " << this->getSalary() << endl;
}
```

Programmation orientée objet

Destructeurs virtuels

Destructeur virtuel

- Soient les classes suivantes:

```
class StockExchangeEntity {  
public:  
    StockExchangeEntity();  
    ~StockExchangeEntity();  
    virtual void addShare();  
};  
  
class Company : public StockExchangeEntity {  
public:  
    Company();  
    ~Company();  
    void addShare() override;  
};
```

Destructeur virtuel (suite)

- Considérons maintenant le programme suivant:

```
int main() {  
    unique_ptr<StockExchangeEntity> s = make_unique<Company>();  
}
```

- À la sortie du programme, quel destructeur sera appelé lors de la désallocation de l'espace mémoire gérée par **s**?

Destructeur virtuel (suite)

- Si on regarde en détail la définition de la classe StockExchangeEntity:

```
class StockExchangeEntity{
public:
    StockExchangeEntity();
    ~StockExchangeEntity();
    virtual void addShare();
};
```

Destructeur virtuel (suite)

- Étant donné que le destructeur de la classe StockExchangeEntity n'a pas été déclaré virtuel, le destructeur de la classe Company ne sera jamais appelé
- Il est donc important de déclarer virtuel le destructeur de StockExchangeEntity
- Ceci sera vrai pour toutes les situations où on utilise le polymorphisme. **Toute classe de base polymorphe devrait avoir un destructeur virtuel**

Destructeur virtuel (suite)

```
class StockExchangeEntity {  
public:  
    StockExchangeEntity();  
    virtual ~StockExchangeEntity() = default;  
    virtual void addShare();  
};
```

Si ce destructeur n'a rien à faire de spécial.
(C++11)

```
class Company: public StockExchangeEntity {  
public:  
    Company();  
    virtual ~Company();  
    void addShare() override;  
};
```

La convention actuelle est de mettre
“virtual”, pas “override”, sur le destructeur,
car un destructeur n’“override” pas les
autres, au sens anglais du mot,
il s’ajoute aux autres.