Programmation orientée objet

Héritage

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();
  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 la manière dont on hérite (nous utliserons toujours l'héritage public).

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

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

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

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

On a aussi ajouté un attribut.

Utilisation d'un objet d'une classe dérivée

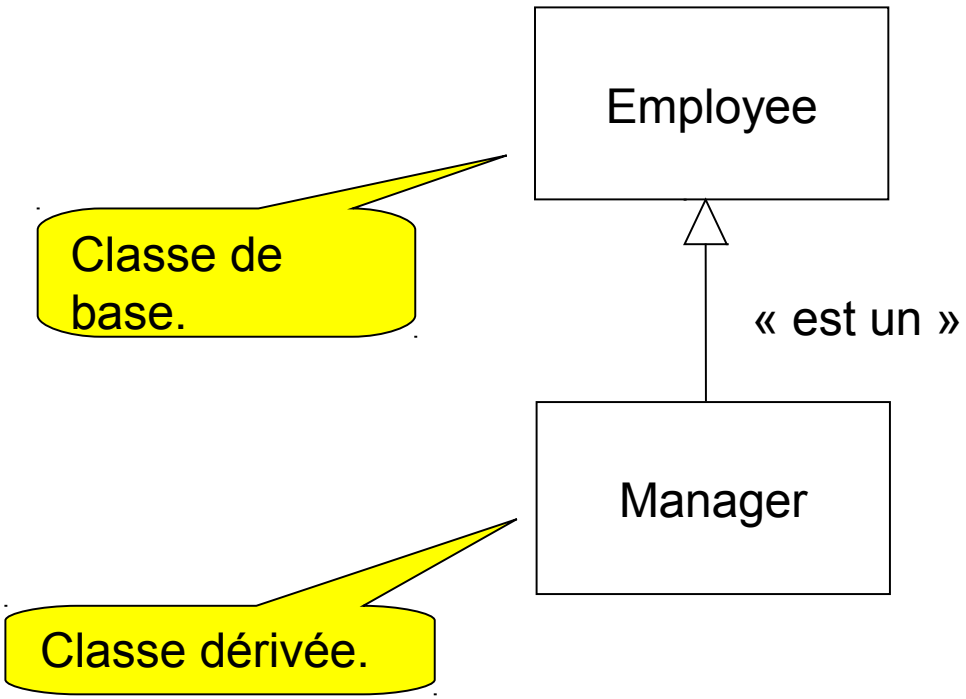
- On peut utiliser les méthodes héritées tout comme les méthodes définies dans la classe dérivée:

```
...
Employee* e1 = new Employee("John", 15000);
Manager e2;
e1->setSalary(29000);
e2.setSalary(48000);
e2.addEmploye(e1);
...
```

C'est la méthode de la classe **Employee** qui est appelée.

C'est la méthode de la classe **Employee** 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)

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)

Exemple de l'horloge (suite)

```
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);
   ...
   return 0;
}
```

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 { return location_; }
   int getHours() const;
private:
   string location_;
   int timeDifference_;
};
```

Encore une fois, pas de constructeur par défaut.

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

La méthode **getHours()** étend celle de la classe de base: elle appelle la méthode de la classe de base pour obtenir l'heure et y ajoute le décalage.

Deux nouveaux attributs ajoutés.

Constructeur de la classe dérivée

- Il est important de noter qu'avant d'appeler le constructeur par défaut d'une classe dérivée, le constructeur par défaut de la classe de base est d'abord appelé
- Si on veut capter le constructeur de la classe de base pour appeler plutôt un constructeur par paramètre, il faut l'appeler dans la liste d'initialisation

Constructeur de la classe dérivée (suite)

- Voici, par exemple, le constructeur de TravelClock:

```
TravelClock::TravelClock(bool mil, string loc, int diff)
    : Clock(mil)
{
    location_ = loc;
    timeDifference_ = diff;
    while (timeDifference_ < 0)
       timeDifference_ = timeDifference_ + 24;
}
```

Ici, comme la classe **Clock** n'a pas de constructeur par défaut, il faut absolument passer un paramètre au constructeur.

Ordre d'appel des constructeurs

- Liste de construction pour un objet d'une classe **C**:
  - **C** dérive-t-elle d'une autre classe **D** ?

    Si oui, **D** est-elle spécifiée dans la liste d'initialisation?
    - Si oui, utiliser ce constructeur.
    - Si non, utiliser le constructeur par défaut

    Construire **D** (liste de construction pour **D**)

Ordre d'appel des constructeurs

- Liste de construction pour un objet d'une classe **C**:
  - **C** dérive-t-elle d'une autre classe **D** ?

    Construire **D** (liste de construction pour **D**)
  - **C** a-t-elle des attributs **A1**, **A2**, ... ?
    - Si oui, construire **A1** (liste de construction pour **A1**), **A2**, … avec le constructeur spécifié dans la liste d'initialisation, ou, s'il n'est pas spécifié pour l'attribut, celui par défaut

Ordre d'appel des constructeurs

- Liste de construction pour un objet d'une classe **C**:
  - **C** dérive-t-elle d'une autre classe **D** ?

    Construire **D** (liste de construction pour **D**)
  - **C** a-t-elle des attributs **A1**, **A2**, ... ?

    Construire **A1** (liste de construction pour **A1**), **A2**, ...
  - **C** peut maintenant être construite, on peut exécuter ce qui se trouve entre {} du constructeur appelé pour **C**.

Ordre d'appel des destructeurs

- Liste de destruction pour un objet d'une classe **C**:
  - **C** peut être détruite, on peut exécuter ce qui se trouve entre {} du destructeur de **C** (s'il y en a un)
  - **C** a-t-elle des attributs **A1**, **A2**, … ?

    Détruire …, **A2** (liste de destruction pour **A2**), **A1**.
  - **C** dérive-t-elle d'une autre classe **D** ?

    Détruire **D** (liste de destruction pour **D**)

C'est la même chose que la liste de construction... dans l'autre sens!

Ordre d'appel des constructeurs (exemple)

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

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

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

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

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

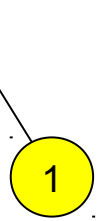
Ordre d'appel des constructeurs (exemple)

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

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

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

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




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

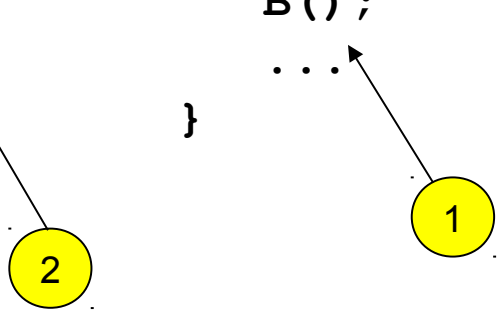
Ordre d'appel des constructeurs (exemple)

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

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

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

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




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

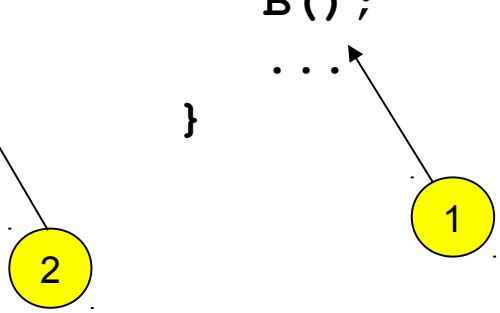
Ordre d'appel des constructeurs (exemple)

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

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

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

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




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

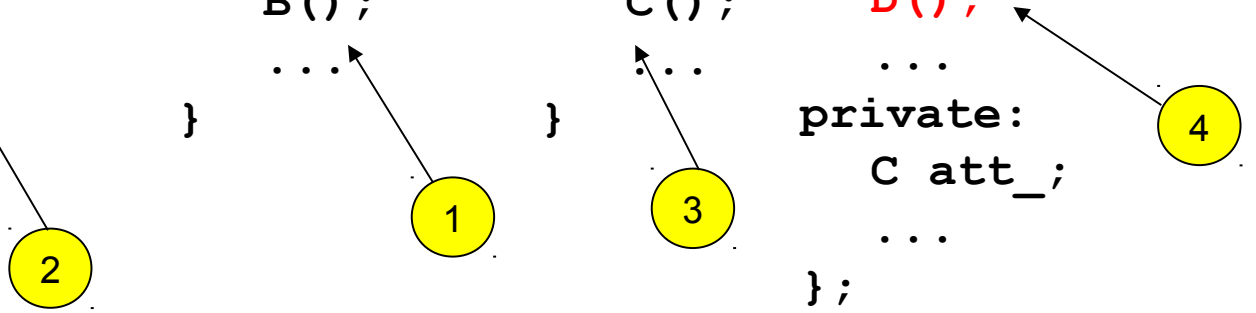
Ordre d'appel des constructeurs (exemple)

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

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

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

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




```
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)
{
   ...
}
```

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);
   ...
}
```

On construit d'abord cet attribut, mais en utilisant son constructeur qui prend un paramètre entier.

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

D::D(int p, int q)
   : att_(p), A(q)
{
   ...
}
```

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);
   ...
}
```

On exécute ensuite le constructeur de **A**.

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

D::D(int p, int q)
   : att_(p), A(q)
{
   ...
}
```

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)
{
   ...
}
```

On construit cet attribut, mais en utilisant son constructeur qui prend un paramètre entier.

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)
{
   ...
}
```

On exécute ensuite le constructeur de **D**.

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)

- Si on veut construire un objet de la classe **Employee**:
  - **Employee** n'a pas de classe de base
  - **Employee** a deux attributs: **name_** et **salary_**, on doit les construire
  - On exécute l'implémentation du constructeur d'**Employee**
- Si on veut construire un objet de la classe **Manager**:
  - **Manager** dérive d'**Employee**, on doit donc construire **Employee** en premier!
  - **Manager** a deux attributs: **bonus_** et **managedEmployees_**, on doit les construire
  - On exécute l'implémentation du constructeur de **Manager**

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...

Appel des méthodes de la classe de base

- Soit B une sous-classe de A
- Si on considère que tout objet de type B est aussi de type A, une méthode de B devrait pouvoir appeler une méthode de la classe A
- On fait cela en préfixant par « A:: » la méthode appelée dans la classe B

Appel des méthodes de la classe de base (suite)

- Voici un exemple d'appel d'une méthode de la classe de base:

```
int TravelClock::getHours() const
{
    int h = Clock::getHours();
    if (isMilitary())
        ... on retourne l'heure entre 0 et 24
    else
    {
        ... on retourne l'heure entre 0 et 12
    }
}
```

Il faut d'abord appeler la méthode **get_hours()** de la classe de base pour obtenir l'heure locale.

Cette méthode de la classe de base peut être appelée sans préfixe puisque qu'elle est héritée.

On ajoute ensuite à l'heure locale le décalage de la zone.

Autre exemple de classe dérivée Gerant

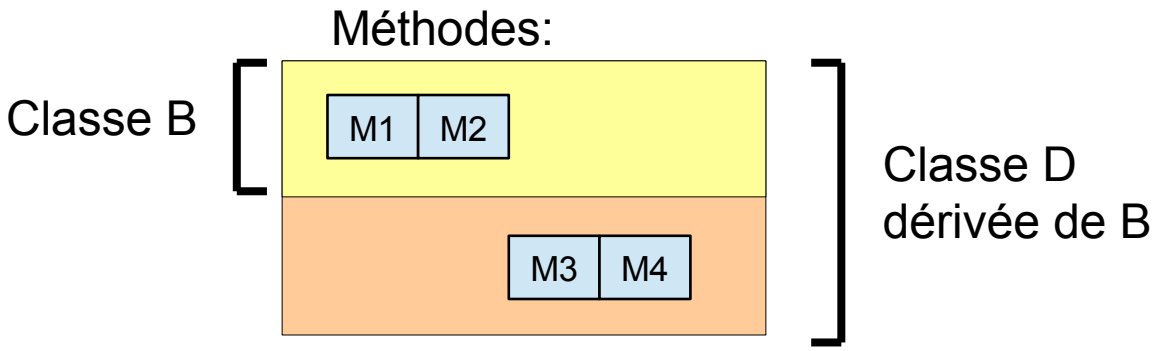
```
class Manager : public Employee
{
public:
  Manager();
  void addEmploye(Employee* employee);
  Employee* getEmploye(string name) const;
  double getSalary() const;
private:
  double bonus_;
  vector<Employee*> managedEmployees_;
};

double Manager::getSalary() const
{
  double baseSalary = Employee::getSalary();
  return (baseSalary + (1 + bonus_ / 100.0));
}
```

On redéfinit la méthode **getSalary()**.

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

Appel d'une méthode



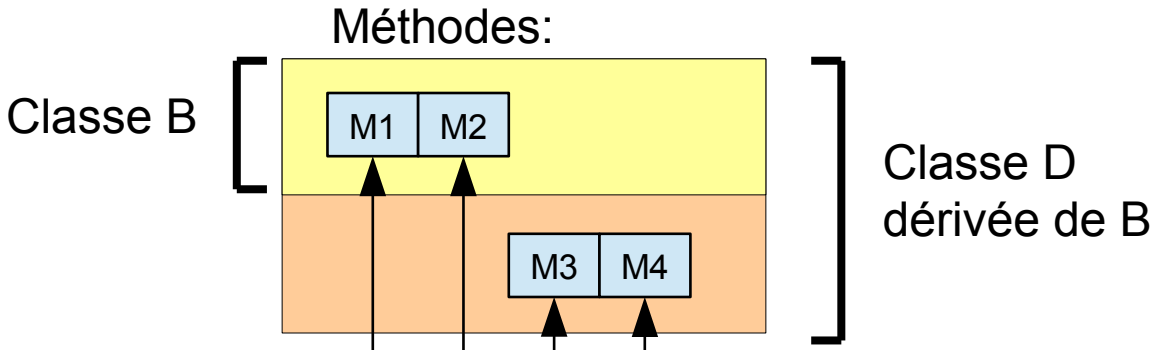

```
int main() {
    D d;

    d.M1();
    d.M2();
    d.M3();
    d.M4();

    return 0;
}
```

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

Appel d'une méthode



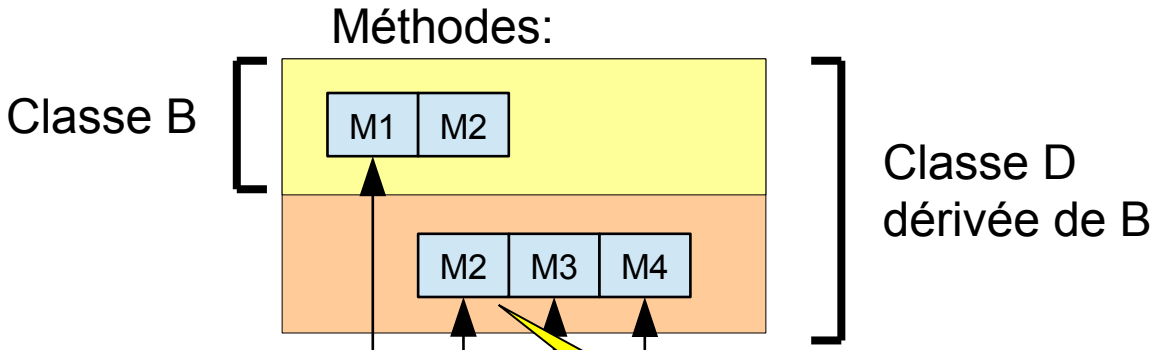

```
int main() {
     D d;

     d.M1();
     d.M2();
     d.M3();
     d.M4();

     return 0;
}
```

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



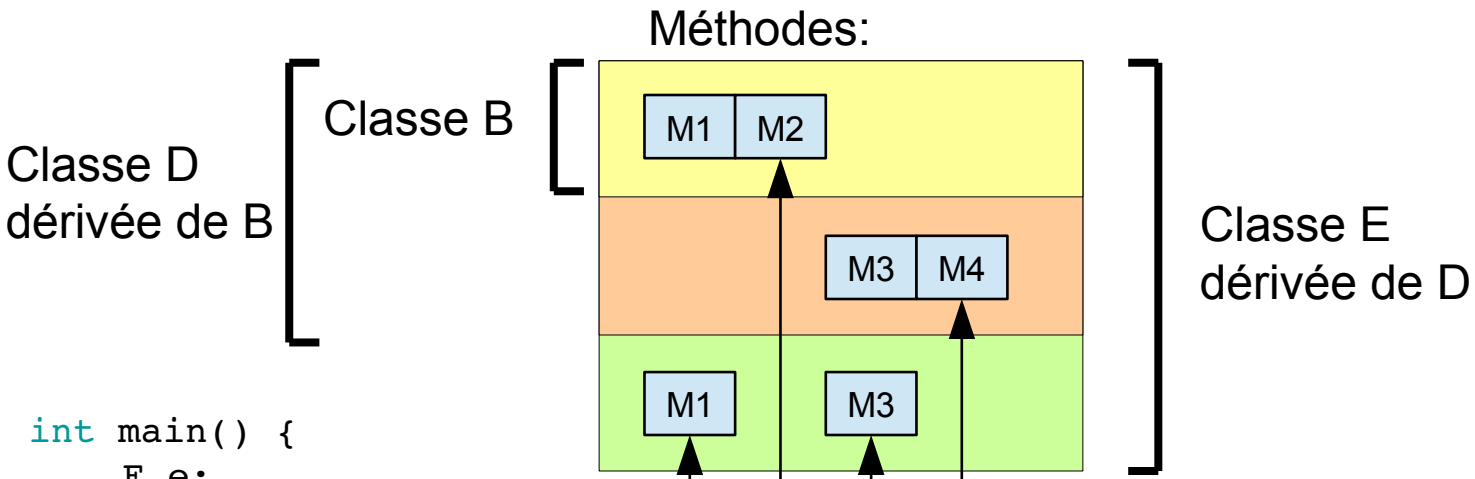

```
int main() {
     D d;

     d.M1();
     d.M2();
     d.M3();
     d.M4();

     return 0;
}
```

Ce qui veut aussi dire que si **D** redéfinit une méthode existante dans **B**, c'est celle de **D** que l'on appellera ici!

Appel d'une méthode




```
int main() {
     E e;

     e.M1();
     e.M2();
     e.M3();
     e.M4();

     return 0;
}
```

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**.

Appel d'une méthode d'une classe dérivée

- Si une classe **B** a une méthode **M()**. Pour une classe **D** qui dérive de **B** il y a donc deux possibilités:
  - **D** surcharge **M()** en déclarant une méthode **M()**, c'est donc cette méthode qui sera appelée lorsqu'on appellera **M()** sur un objet de la classe **D**
  - **D** ne surcharge pas **M()**, c'est donc la méthode **M()** de **B** qui sera appelée lorsqu'on appellera **M()** sur un objet de la classe **D**. On dira que **M()** est héritée.
- Dans le cas ou la surcharge de **M()** dans **D** fait appel à **M()** dans **B** (e.g. **getHours()**) on dira que **M()** est étendue

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
- Tout membre *protégé* est inaccessible à l'extérieur de la classe, mais accessible à ses classes dérivées
- En général, un attribut est toujours privé
- Donc, 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

Accès aux membres pour une classe C

public:

protected:

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:

Accessible à la classe C seulement:

```
class C
```

Accessible à tout le monde:

```
class A
class B
class C
...
```

Accès aux membres pour une classe C

private et **protected** sont aussi accessibles aux classes et fonctions **friend**

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 (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_;
};
```

Accessibles à tout le monde.

Accessibles seulement à la classe **A** et ses classes dérivées, ainsi qu'aux classes et fonctions amies.

Accessibles seulement à la classe **A** et ses amis.