# *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 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 ajouté un attribut managedEmployees_ à la classe Manager

# Exemple de classe dérivée

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

| Classe | Méthodes accessibles |
|---|---|
| Employee | setSalary();<br>getSalary();<br>getName(); |
| Manager | setSalary();<br>getSalary();<br>getName();<br>addEmployee()<br>getEmployee() |

# Utilisation d'un objet d'une classe dérivée

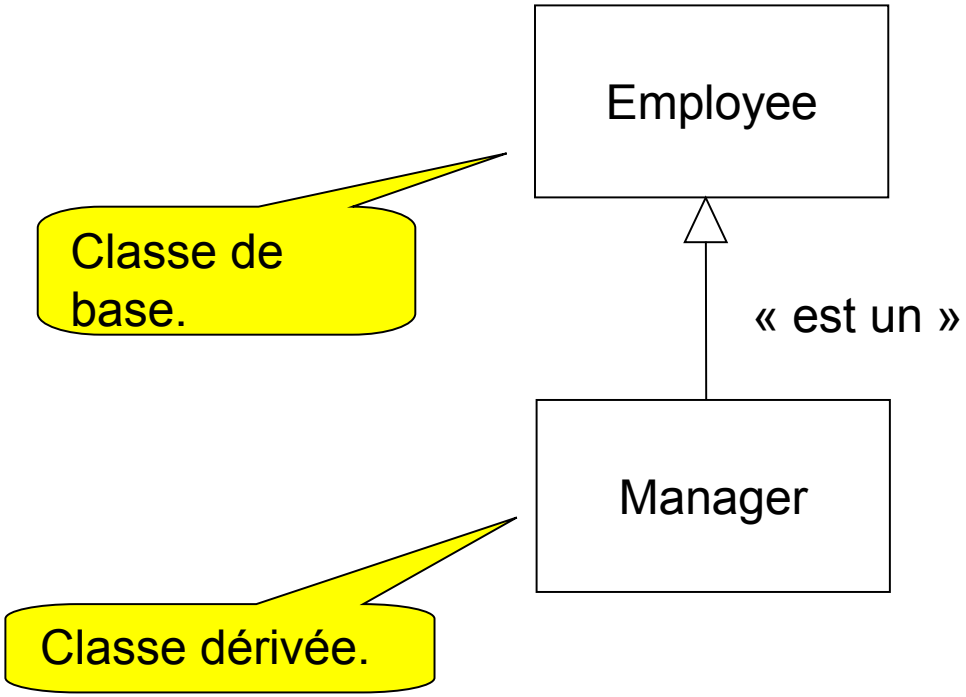
- On peut utiliser les méthodes héritées tout comme les méthodes définies dans la classe dérivée:

```
int main() {
    shared_ptr<Employee> e1 = make_shared<Employee>("John", 15000);
    Manager e2;
    e1->setSalary(29000);
    e2.setSalary(48000);
    e2.addEmployee(e1);
}
```

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

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_;
};
```

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.

Deux nouveaux attributs ajoutés.

# Exemple de l'horloge (suite)

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

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

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

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

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

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

On appel 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()<br>getHours()<br>getMinutes()<br>isMilitary() |
| TravelClock | getLocation()<br>getHours()<br>getMinutes()<br>isMilitary() |

Ces méthodes sont accessibles même si elles n'ont pas été définies dans TravelClock puisqu'elles sont hérités 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:
    Clock(bool useMilitary): military_(useMilitary){ }
private:
        bool military_;
};
class TravelClock : public Clock {
public:
    TravelClock(bool mil, string loc, int diff) : Clock(mil),
        location_(loc), timeDifference_(diff) { }
private:
    string location_;
    int timeDifference_;
};

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

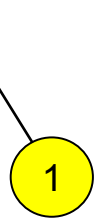
# Ordre d'appel des constructeurs (exemple)

```
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;
   ...
}
```

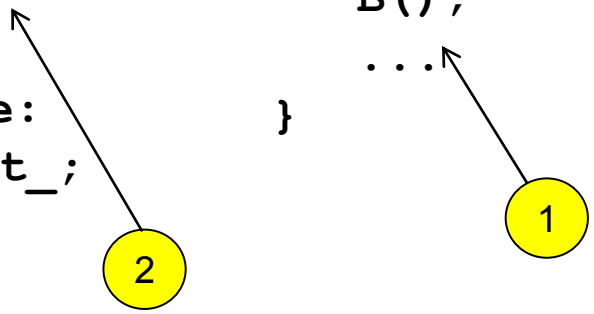
# Ordre d'appel des constructeurs (exemple)

```
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;
   ...
}
```

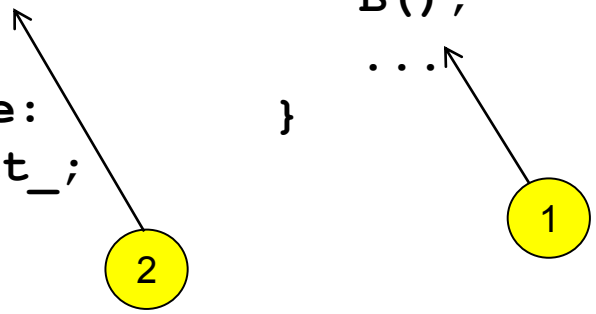
# Ordre d'appel des constructeurs (exemple)

```
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;
   ...
}
```

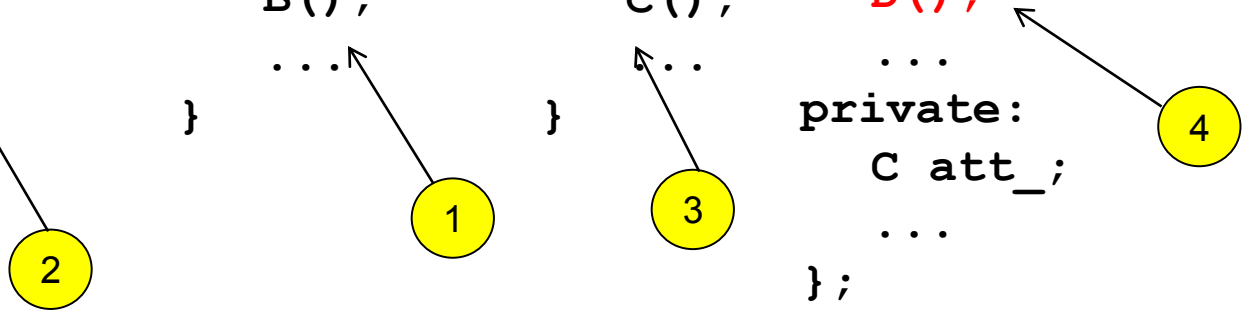
# Ordre d'appel des constructeurs (exemple)

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

```
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)
{
   ...
}


int main()
{
   D objet(3,2);
   ...
}
```

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)
{
   ...
}


int main()
{
   D objet(3,2);
   ...
}
```

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)
{
   ...
}


int main()
{
   D objet(3,2);
   ...
}
```

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_;
};
```

3

# 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);
   ...
}
```

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) {
      ...
   }
private:
   int x_;
};
```

3

# 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...

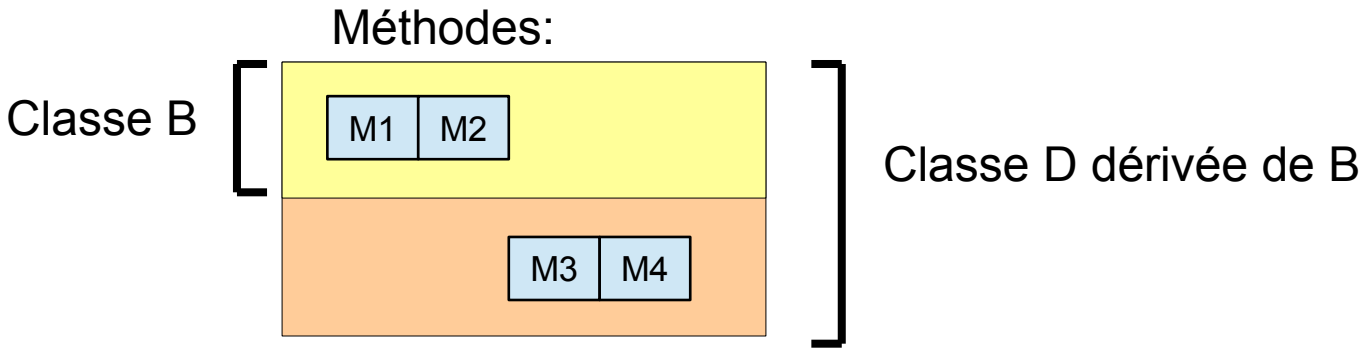
# *Programmation orientée objet*

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

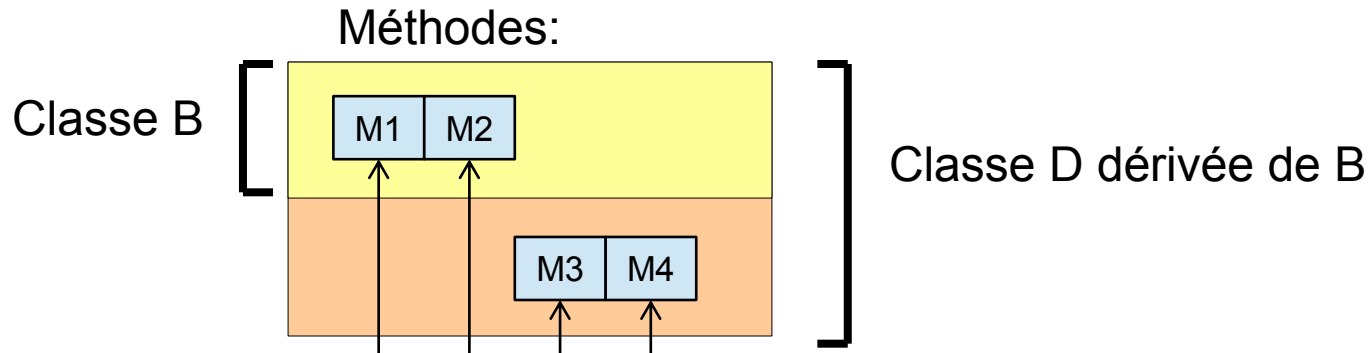



```
int
D d;
```

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

```
return
```

# Appel d'une méthode



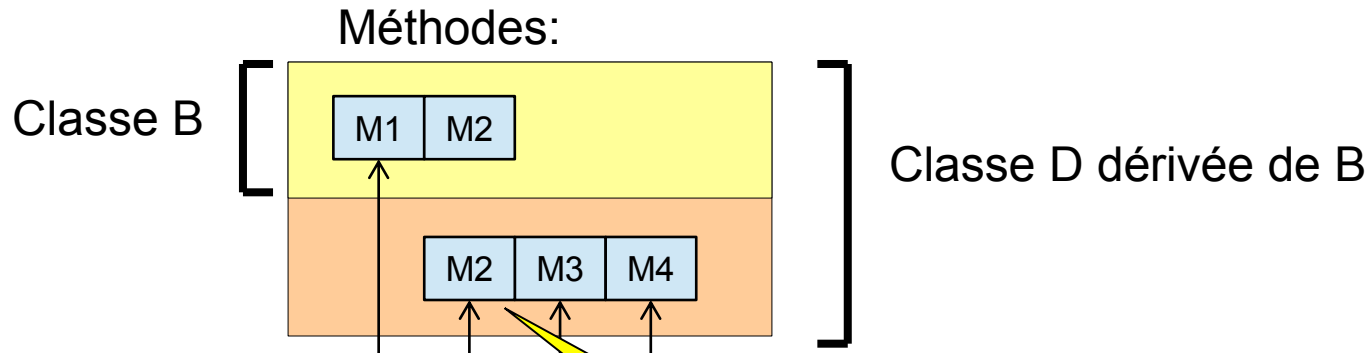

```
int
D d;

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

return
```

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



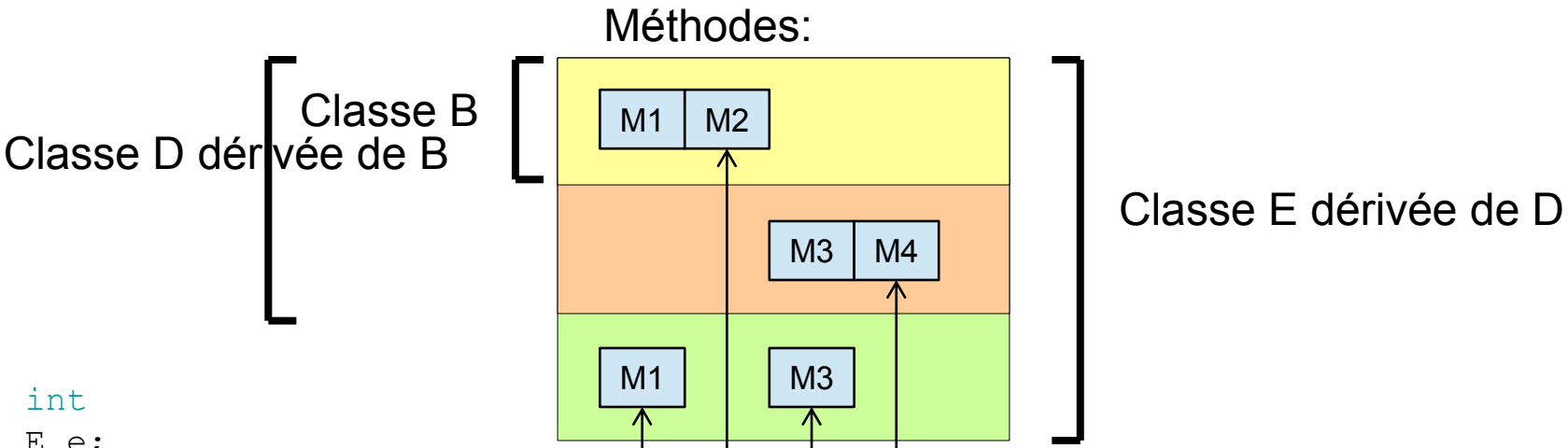

```
int
D d;

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

return
```

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
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

**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_;
};
```

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