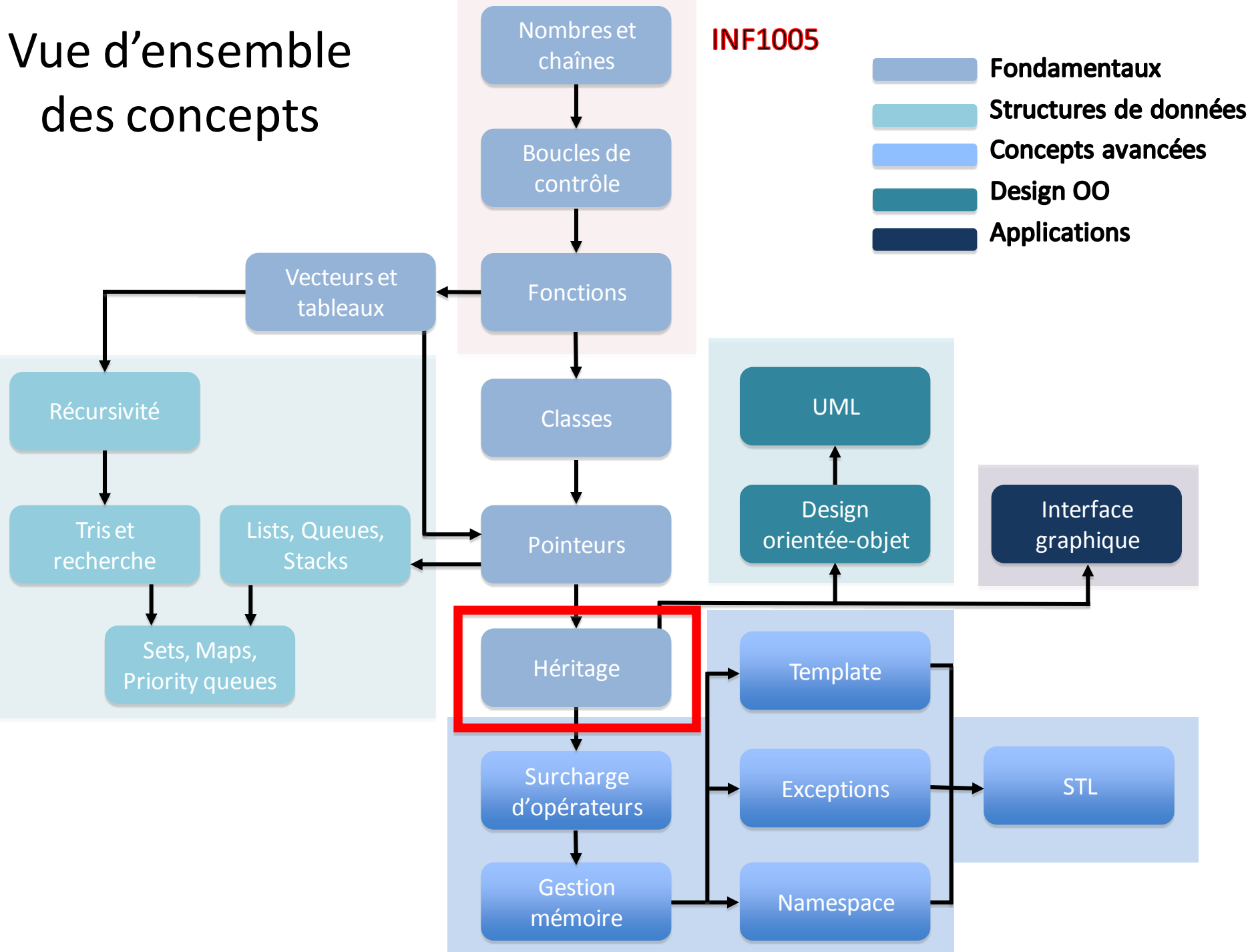


Programmation orientée objet

Héritage

Vue d'ensemble des concepts

INF1005



Héritage

- Mécanisme pour **RÉUTILISATION** du code source:
 - bénéficier automatiquement des fonctionnalités et changements d'une classe existante
 - ajouter de nouvelles fonctionnalités à une classe existante
 - changer un peu le comportement de certaines méthodes d'une classe déjà existante
- ... mais **sans rien changer** à la classe 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 Employe représente un employé, dont les attributs sont son nom et son salaire
- Supposons maintenant qu'on veuille représenter un gérant, qui est un employé, mais qui en plus supervise d'autres employés

Exemple de classe dérivée

- Voyons d'abord la classe de base:

```
class Employe
{
public:
    Employe();
    Employe(string nomEmploye, double salaireInitial);
    void modifierSalaire(double nouveauSalaire);
    double obtenirSalaire() const;
    string obtenirNom() const;

private:
    string nom_;
    double salaire_;
};
```

Exemple de classe dérivée

- Et maintenant la classe dérivée:

Pour spécifier la manière dont on hérite (nous utiliserons toujours l'héritage public).

```
class Gerant : public Employe
{
public:
    Gerant();
    void ajouterEmploye(Employe* employe);
    Employe obtenirEmploye(string nom) const;

private:
    vector< Employe* > employeesSupervises_;
};
```

On indique que **Gerant** est une sous-classe de **Employe**.

En plus des méthodes de la classe **Employe**, 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:

...

```
Employe* employe1 = new Employe("Mehdi", 18000);
```

```
Gerant employe2;
```

```
employe1->modifierSalaire(29000);
```

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

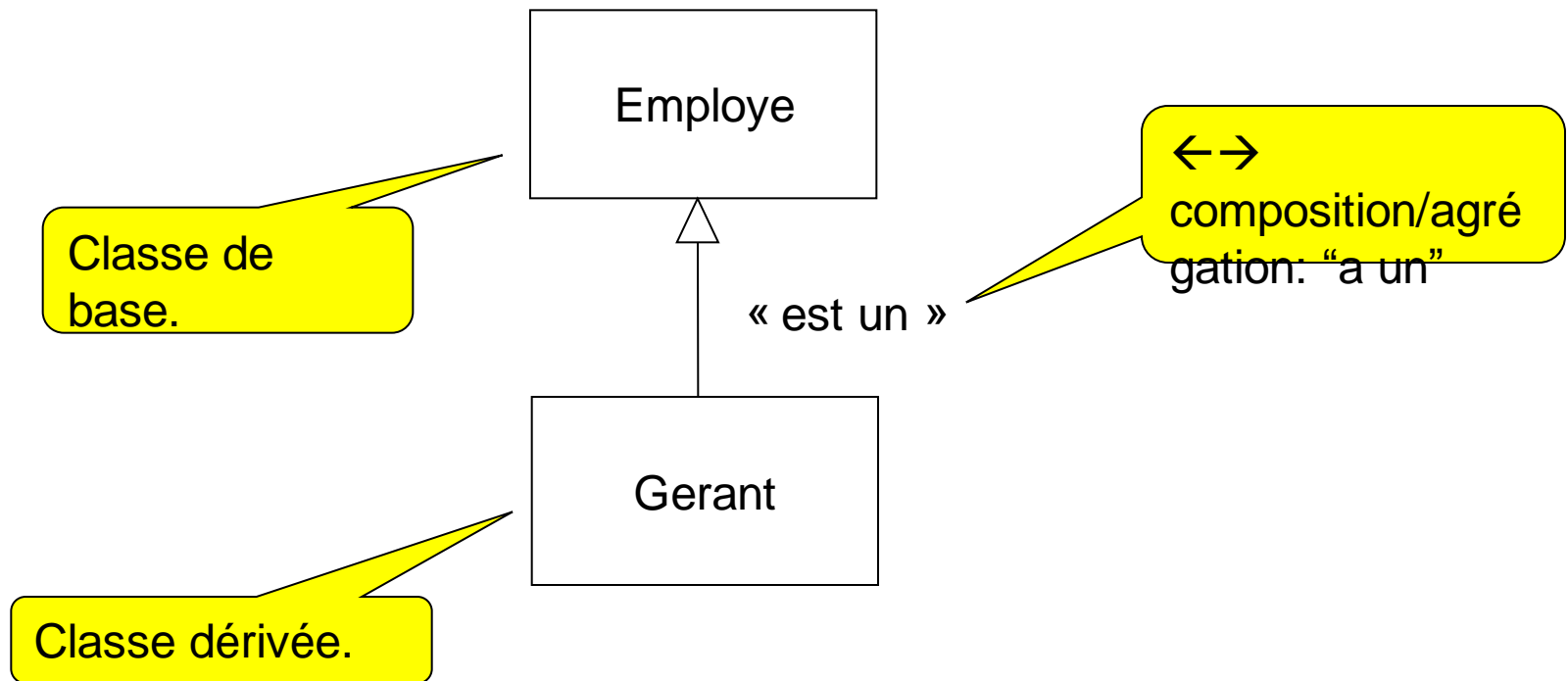
```
employe2.modifierSalaire(48000);
```

```
employe2.ajouterEmploye(employe1);
```

C'est la méthode de la classe **Employe** 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 Gerant qui est une sous-classe (une classe dérivée) de la classe Employe
- Nous considérons donc que tout gérant 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()
```

```
{
```

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

```
    Clock horloge1(true);
```

```
    Clock horloge2(false);
```

```
    ...
```

```
    return 0;
```

```
}
```

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

1. [construction de l'objet de base]
 1. Les constructeurs des attributs de la classe de base (défaut OU appelés dans liste d'init. du constr. de la classe de *base*)
 2. Le constructeur par défaut de la classe de base OU appelé dans liste d'init. du constr. de la classe *dérivée*
2. [construction de l'objet dérivé]
 1. Les constructeurs des attributs de la classe dérivée (défaut OU appelés dans liste d'init. du constr. de la classe *dérivée*)
 2. Le constructeur de la classe dérivée appelé

Ordre d'appel des destructeurs

1. [destruction de l'objet dérivé]
 1. Le destructeur de la classe dérivée
 2. Les destructeurs des attributs de la classe dérivée
2. [destruction de l'objet de base]
 1. Le destructeur de la classe de base
 2. Les destructeurs des attributs 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 att2_;
    ...
};
```

```
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 att2_;
    ...
};
```

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

1



Ordre d'appel des constructeurs (exemple)

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

2

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

1

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

```
class D : public A
{
public:
    D();
    ...
private:
    C att2_;
    ...
};
```

```
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 att2_;
    ...
};
```

```
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 att2_;
    ...
};
```

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 att2_;
    ...
};
```

```
D::D(int p, int q)
: att2_(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 att2_;
```

```
};

D::D(int p, int q)
: att2_(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)
{
    ...
}
```

On exécute ensuite le constructeur de A.

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

```
class D : public A
{
public:
    D();
    D(int p, int q);
    ...
private:
    C att2_;
    ...
};

D::D(int p, int q)
: att2_(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 att2_;
    ...
};

D::D(int p, int q)
: att2_(p), A(q)
{
    ...
}
```

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

Ordre d'appel des constructeurs (second exemple)

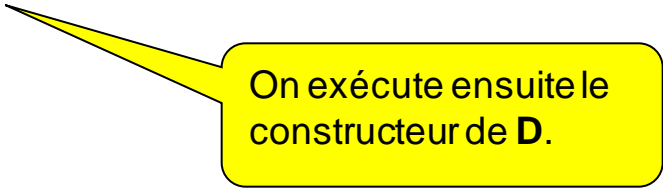
```
class A
{
public:
    A();
    A(int x);
    ...
private:
    B att_;
};

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 att2_;
    ...
};

D::D(int p, int q)
: att2_(p), A(q)
{
    ...
}
```



On exécute ensuite le constructeur de D.

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
    }
}
```

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

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

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

Autre exemple de classe dérivée Gerant

```
class Gerant : public Employe
```

```
{
```

```
public:
```

```
    Gerant();
```

```
    void ajouterEmploye(Employe* employe);
```

```
    Employe obtenirEmploye(string nom) const;
```

```
    double obtenirSalaire() const;
```

```
private:
```

```
    double prime_;
```

```
    vector< Employe* > employeesSupervises_;
```

```
};
```

```
double Gerant::obtenirSalaire() const
```

```
{
```

```
    double salaireDeBase = Employe::obtenirSalaire() ;
```

```
    return (salaireDeBase + prime_);
```

```
}
```

en anglais: "method **overriding**"

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

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

Appel d'une méthode d'une classe dérivée

- Soit B une classe de base et `B::fct()` une de ses méthodes
- Pour une classe D dérivée de B, il y a trois possibilités pour la méthode `D::fct()` :

Appel d'une méthode d'une classe dérivée

- Soit B une classe de base et `B::fct()` une de ses méthodes
- Pour une classe D dérivée de B, il y a trois possibilités pour la méthode `D::fct()` :
 - `D::fct()` étend `B::fct()` en appelant la méthode `B::fct()` et en réalisant de nouveaux calculs à partir du résultat obtenu (ex.: `getHours()`)

Appel d'une méthode d'une classe dérivée

- Soit B une classe de base et `B::fct()` une de ses méthodes
- Pour une classe D dérivée de B, il y a trois possibilités pour la méthode `D::fct()` :
 - `D::fct()` étend `B::fct()` en appelant la méthode `B::fct()` et en réalisant de nouveaux calculs à partir du résultat obtenu (ex.: `getHours()`)
 - **`D::fct()` est une implémentation complètement indépendante de `B::fct()` (ex.: `getLocation()`)**

Appel d'une méthode d'une classe dérivée

- Soit B une classe de base et `B::fct()` une de ses méthodes
- Pour une classe D dérivée de B, il y a trois possibilités pour la méthode `D::fct()` :
 - `D::fct()` étend `B::fct()` en appelant la méthode `B::fct()` et en réalisant de nouveaux calculs à partir du résultat obtenu (ex.: `getHours()`)
 - `D::fct()` est une implémentation complètement indépendante de `B::fct()` (ex.: `getLocation()`)
 - **`D::fct()` n'est pas définie dans D, ce qui implique qu'on appellera directement la méthode `B::fct()` lorsqu'on voudra exécuter la méthode `fct()` sur un objet de classe D (on dit que la méthode `fct()` est *héritée*) (ex.: `isMilitary()`)**

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 d'une classe de base (exemple)

```
class A
{
public:
    A();
    ~A();
    int obtenirAtt1() const;
    void modifierAtt1(int x);
protected:
    int obtenirAtt2() const;
    void modifierAtt2(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.