

Un exemple

Oublions un peu les rectangles ...



Exemple : classes pour les personnages

class Guerrier

```
string nom
int energie
int duree_vie

Arme arme
rencontrer(Personnage&)
```

class Voleur

```
string nom
int energie
int duree_vie

rencontrer(Personnage&)
voler(Personnage&)
```

class Magicien

```
string nom
int energie
int duree_vie

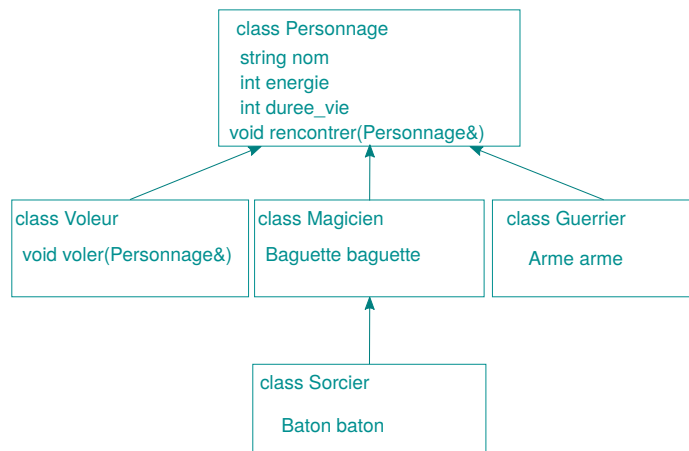
Baguette baguette
rencontrer(Personnage&)
```

class Sorcier

```
string nom
int energie
int duree_vie

Baguette baguette
Baton baton
rencontrer(Personnage&)
```

Exemple : héritage

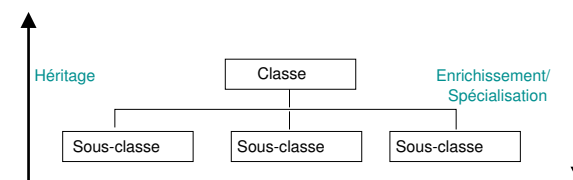


Héritage

Après les notions d'*encapsulation* et d'*abstraction*, le troisième aspect essentiel de la « Programmation Orientée Objet » est la notion d'**héritage**.

L'héritage représente la relation «**est-un**».

Il permet de créer des classes *plus spécialisées*, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **super-classes**.



Héritage (2)

Lorsqu'une sous-classe **C1** est créée à partir d'une super-classe **C**,

- ▶ le type est *hérité* : un **C1** **est** (aussi) **un C**
- ▶ **C1** va *hériter* de l'ensemble :
 - ▶ des attributs de **C**
 - ▶ des méthodes de **C**
(sauf les constructeurs et destructeur)
- ☞ Les attributs et méthodes de **C** vont être disponibles pour **C1** sans que l'on ait besoin de les redéfinir explicitement dans **C1**.
- ▶ Par ailleurs :
 - ▶ des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe **C1**
☞ **enrichissement**
 - ▶ des méthodes héritées de **C** peuvent être redéfinies dans **C1**
☞ **spécialisation**

Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- ▶ le type est *hérité* : un **Guerrier** **est** (aussi) **un Personnage** :

```
Personnage p;  
Guerrier g;  
// ...  
p = g;  
// ...  
void afficher(Personnage const&);  
// ...  
afficher(g);
```

Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- ▶ **Guerrier** va *hériter* de l'ensemble des attributs et des méthodes de **Personnage**
(sauf les constructeurs et destructeur)

```
class Personnage  
{  
    string nom  
    int energie  
    int duree_vie  
    void rencontrer(Personnage&)  
};
```

```
class Guerrier  
{  
    Arme arme  
};
```

```
Guerrier g;  
Voleur v;  
  
g.rencontrer(v);  
//...  
// dans Guerrier::methode():  
    energie = //...
```

Héritage : exemple

Lorsqu'une sous-classe **C1** (ici **Guerrier** ou **Voleur**) est créée à partir d'une super-classe **C** (ici **Personnage**),

- ▶ des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe **Guerrier** : **arme**
- ▶ des méthodes héritées de **Personnage** peuvent être redéfinies dans **Voleur** : **rencontrer(Personnage&)**

Héritage (3)

L'héritage permet donc :

- ▶ d'expliciter des relations structurelles et sémantiques entre classes
- ▶ de réduire les redondances de description et de stockage des propriétés



Attention !

- ▶ l'héritage doit être utilisé pour décrire une relation « **est-un** » ("is-a")
- ▶ il ne doit **jamais** décrire une relation « a-un »/« possède-un » ("has-a")

Transitivité de l'héritage

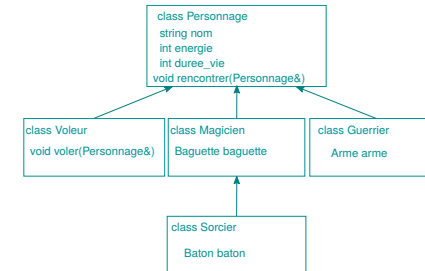
Par transitivité, les instances d'une sous-classe possèdent :

- ▶ les attributs et méthodes (hors constructeurs/destructeur) de l'ensemble des classes parentes (super-classe, super-super-classe, etc.)

Enrichissement par héritage :

- ▶ crée un *réseau de dépendances* entre classes,
- ▶ ce réseau est organisé en une *structure arborescente* où chacun des nœuds hérite des propriétés de l'ensemble des nœuds du chemin remontant jusqu'à la racine.

- ▶ ce réseau de dépendances définit une **hiérarchie de classes**



Sous-classe, Super-classes

Une **super-classe** :

- ▶ est une classe « parente »
- ▶ déclare les attributs/méthodes communs
- ▶ peut avoir plusieurs sous-classes

Une **sous-classe** est :

- ▶ une classe « enfant »
- ▶ étend **une (ou plusieurs)** super-classe(s)
- ▶ hérite des **attributs**, des **méthodes** et du **type** de la super-classe

Un attribut/une méthode hérité(e) peut s'utiliser comme si il/elle était déclaré(e) dans la sous-classe au lieu de la super-classe (en fonction des droits d'accès, voir plus loin)

- ▶ On évite ainsi la **duplication de code**

Passons à la pratique...

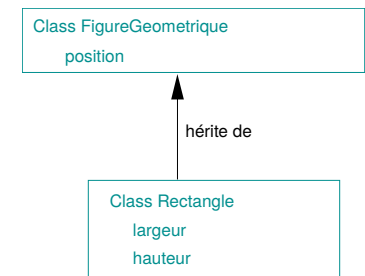
Définition d'une sous-classe en C++ :

Syntaxe :

```
class NomSousClasse : public NomSuperClasse
{
    /* Déclaration des attributs et méthodes
       spécifiques à la sous-classe */
};
```

Exemple :

```
class Rectangle : public FigureGeometrique
{
    //...
private:
    double largeur; double hauteur;
};
```



Pratique : exemple 2

```
class Personnage {  
    // ...  
};  
// ...  
class Guerrier : public Personnage {  
public:  
    // constructeurs, etc.  
private:  
    Arme arme;  
};
```


Utilisation des droits d'accès

- ▶ Membres *publics* : accessibles pour les **programmeurs utilisateurs** de la classe
- ▶ Membres *protégés* : accessibles aux **programmeurs d'extensions** par héritage de la classe
- ▶ Membres *privés* : pour le **programmeur de la classe** : structure interne, (modifiable si nécessaire sans répercussions ni sur les utilisateurs ni sur les autres programmeurs)

Les Guerrier font bande à part

- Pour un personnage non-Guerrier :

```
void rencontrer(Personnage& le_perso) const { saluer(le_perso); }
```

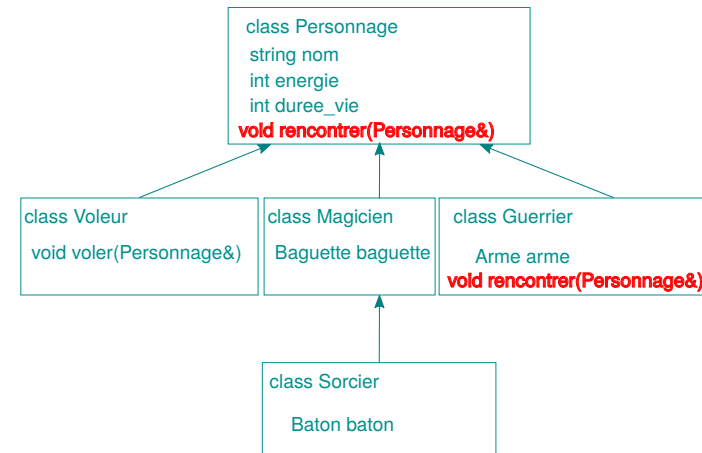
- Pour un Guerrier

```
void rencontrer(Personnage& le_pauvre) const { frapper(le_pauvre); }
```

Faut-il re-concevoir toute la hiérarchie ?

- ☞ Non, on ajoute simplement une méthode `rencontrer(Personnage&)` spéciale dans la sous-classe `Guerrier`

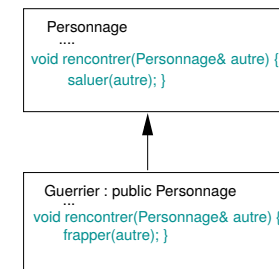
Les Guerrier font bande à part : masquage



Masquage dans une hiérarchie

- Masquage : un identificateur qui en cache un autre
- Situations possibles dans une hiérarchie :
 - Même nom d'attribut ou de méthode utilisé sur plusieurs niveaux
 - Peu courant pour les attributs
 - Très courant et **pratique** pour les méthodes

Masquage dans une hiérarchie (2)



La méthode `rencontrer` de `Guerrier` **masque** celle de `Personnage`

- Un objet de type `Guerrier` n'utilisera donc **jamais** la méthode `rencontrer` de la classe `Personnage`
- Vocabulaire OO :
 - Méthode héritée = méthode générale, *méthode par défaut*
 - Méthode qui masque la méthode héritée = *méthode spécialisée*

Accès à une méthode masquée

- ▶ Il est parfois souhaitable d'accéder à une méthode/un attribut masqué(e)
- ▶ Exemple :
 - ▶ Le `Guerrier` commence par rencontrer le personnage comme le fait n'importe quel personnage (il le salue) avant de le frapper !
- ▶ Code désiré :
 1. Personnage non-`Guerrier` :
 - ▶ Méthode générale (`rencontrer` de `Personnage`)
 2. Personnage `Guerrier` :
 - ▶ Méthode spécialisée (`rencontrer` de `Guerrier`)
 - ▶ Appel à la méthode générale depuis la méthode spécialisée

Accès à une méthode masquée (2)

Pour accéder aux attributs/méthodes masqué(e)s de la

- ▶ on utilise l'**opérateur de résolution de portée**
- ▶ Syntaxe : `NomClasse::méthode` ou `attribut`
- ▶ Exemple :

```
class Guerrier : public Personnage {  
    //...  
    void rencontrer (Personnage& perso) {  
        Personnage::rencontrer(perso); // salutation d'usage !!  
        frapper(perso);  
    }  
};
```


Constructeurs et héritage

Lors de l'instanciation d'une sous-classe, il faut initialiser :

- ▶ les attributs *propres à la sous-classe*
- ▶ les attributs *hérités des super-classes*

MAIS...

...il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui-même l'*initialisation des attributs hérités*

L'accès à ces attributs pourrait notamment être interdit ! (*private*)

L'initialisation des attributs hérités doit donc se faire au niveau des classes où ils sont explicitement définis.

Solution : l'initialisation des attributs hérités doit se faire en **invoquant les constructeurs des super-classes**.

Constructeurs et héritage : appel explicite

L'invocation du constructeur de la super-classe se fait au **début de la section d'appel aux constructeurs des attributs**.

Syntaxe :

```
SousClasse(liste de paramètres)
: SuperClasse(Arguments),
  attribut1(valeur1),
  ...
  attributN(valeurN)
{
    // corps du constructeur
}
```

Lorsque la super-classe admet un constructeur par défaut, l'invocation explicite de ce constructeur dans la sous-classe n'est pas obligatoire

👉 le compilateur se charge de réaliser l'invocation du constructeur par défaut

Constructeurs et héritage : exemple 1

Si la classe parente n'admet pas de constructeur par défaut, l'**invocation explicite** d'un de ses constructeurs **est obligatoire** dans les constructeurs de la sous-classe

👉 La sous-classe doit admettre *au moins un constructeur explicite*.

Exemple :

```
class FigureGeometrique {
protected:  Position position;
public:
    FigureGeometrique(double x, double y) : position(x, y) {}
    // ...
};

class Rectangle : public FigureGeometrique {
protected:  double largeur; double hauteur;
public:
    Rectangle(double x, double y, double l, double h)
        : FigureGeometrique(x,y), largeur(l), hauteur(h) {}
    // ...
};
```

Constructeurs et héritage : exemple 2

Autre exemple (qui ne fait pas la même chose) :

```
class FigureGeometrique {
protected:  Position position;
public:
    /* Note : le constructeur par défaut par défaut de FigureGeometrique
     *         appelle le constructeur par défaut de Position.
     */
    // ...
};

class Rectangle : public FigureGeometrique {
protected:  double largeur; double hauteur;
public:
    Rectangle(double l, double h)
        : largeur(l), hauteur(h)
    {}
    // ...
};
```

Encore un exemple

Il n'est pas nécessaire d'avoir des attributs supplémentaires...

```
class Carre : public Rectangle {
public:
    Carre(double taille)
        : Rectangle(taille, taille)
    {}
    /* Et c'est tout !
       (sauf s'il y avait des manipulateurs,
        il faudrait alors sûrement aussi les
        redéfinir)
    */
};
```

Constructeurs et héritage : résumé (1)

1. Chaque constructeur d'une sous-classe *doit* appeler un des constructeurs de la super-classe
2. L'appel est la **1^{re} instruction**

Constructeurs et héritage : résumé (2)

Et si l'on oublie l'appel à un constructeur de la super-classe ?

- ▶ Appel automatique au constructeur par défaut de la super-classe
- ▶ Pratique parfois, mais erreur si le **constructeur par défaut** n'existe pas

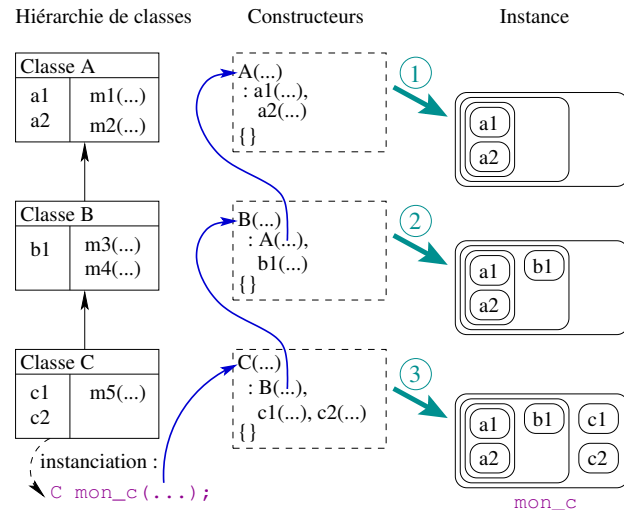
Rappel : le constructeur par défaut est particulier

- ▶ Il existe par défaut pour chaque classe qui n'a aucun autre constructeur
- ▶ Il disparaît dès qu'il y a un autre constructeur

Pour éviter des problèmes avec les hiérarchies de classes, dans un premier temps :

- ▶ Toujours déclarer au moins un constructeur
- ▶ Toujours faire l'appel à un constructeur de la super-classe

Ordre d'appel des constructeurs



Ordre d'appel des destructeurs

Les destructeurs sont toujours appelés dans l'ordre inverse (/symétrique) des constructeurs.

Par exemple dans l'exemple précédent, lors de la destruction d'un **C**, on aura appel et exécution de :

- ▶ C::~~C()
- ▶ B::~~B()
- ▶ A::~~A()

(et dans cet ordre)

(puisque les constructeurs avaient été appelés dans l'ordre

- ▶ A::A()
- ▶ B::B()
- ▶ C::C()

)

Héritage et constructeur de copie

Le constructeur de copie d'une sous-classe doit invoquer explicitement le constructeur **de copie** de la super-classe

☞ Sinon c'est le constructeur **par défaut** de la super-classe qui est appelé !

Exemple :

```
Rectangle(Rectangle const& autre)
: FigureGeometrique(autre),
  largeur(autre.largeur),
  hauteur(autre.hauteur)
{ }
```

C++11 Héritage des constructeurs

Les constructeurs ne sont, en général, **pas hérités**

mais en **C++11** on peut *demandeur leur héritage* en utilisant le mot clé « **using** ».

On récupère alors **tous** les constructeurs de la super-classe,

i.e. on peut construire la sous-classe avec les mêmes arguments, mais...



Attention ! ces constructeurs n'initialisent donc **pas** les **attributs spécifiques de la sous-classe**.

C'est donc **très risqué**, et je vous conseille de ne l'utiliser que pour des sous-classes n'ayant *pas de nouvel attribut* (et si c'est approprié) !

Exemple :

```
class A {
public:
  A(int);
  A(double, double);
  // ...
};
```

```
class B : public A {
using A::A;
/* existent alors maintenant
  B::B(int)
  et B::B(double, double) */
};
```

Petit rappel

Nous avons vu qu'il existe en C++, des méthodes particulières permettant :

- ▶ d'initialiser les attributs d'un objet en début de vie :
constructeurs
- ▶ de copier un objet dans un autre objet :
constructeurs de copie
- ▶ de libérer les ressources utilisées par un objet en fin de vie :
destructeurs

Une **version par défaut**, minimale, de ces méthodes est **automatiquement générée** si on ne les définit pas explicitement.

Petit rappel (2)

Dans certains cas, les versions minimales par défaut des méthodes constructeurs/destructeurs **ne sont pas adaptées** : exemple du *comptage des instances* (cf. semaine passée).

Autre exemple :

Le *constructeur de copie par défaut* réalise une copie membre à membre des attributs
→ **copie de surface**

Ceci pose typiquement problème lorsque **certains attributs** de la classe sont des **pointeurs**.

Examinons pourquoi sur un exemple concret...

Exemple

Soit une autre définition possible (farfelue, mais possible !) de classe *Rectangle* :

```
class Rectangle {  
private:  
    double* largeur; // aïe, un pointeur !  
    double* hauteur;  
public:  
    Rectangle(double l, double h)  
        : largeur(new double(l)), hauteur(new double(h)) {}  
    ~Rectangle() { delete largeur; delete hauteur; }  
    double getLargeur() const;  
    double getHauteur() const;  
    // ...  
};
```

Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)  
{  
    cout << "Largeur: " << tmp.getLargeur() << endl;  
}
```

Exemple (2)

```
void afficher_largeur(Rectangle tmp) { // une copie !..
    cout << "Largeur: " << tmp.getLargeur() << endl;
} // destruction de tmp...
```

- ▶ Lorsque `afficher_largeur` a fini de s'exécuter, l'objet `tmp` est automatiquement détruit par le destructeur de la classe `Rectangle`
- ▶ le destructeur va libérer la mémoire pointée par les champs `largeur` et `hauteur` de `tmp`

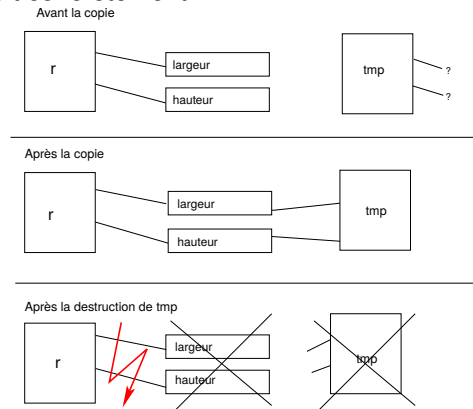


Attention ! cette portion de mémoire est aussi utilisée par `r` dans un appel comme `afficher_largeur(r)` !

☞ (gros risque de) **Segmentation Fault** lors de la prochaine utilisation de `r` !!

Exemple (3)

Voilà ce qui se produit concrètement :

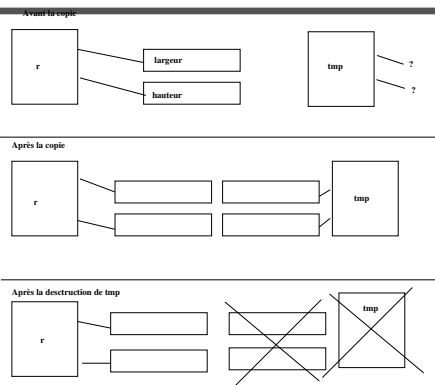


- ☞ il faut **redéfinir le constructeur de copie** de sorte à ce qu'il **duplique véritablement les champs concernés** → **copie profonde**

Exemple (4)

Une bonne solution consiste alors à **redéfinir le constructeur de copie** :

```
Rectangle(const Rectangle& obj)
: largeur(new double(*(obj.largeur))) ,
  hauteur(new double(*(obj.hauteur)))
{};
```



Exemple : Définition complète de la classe

```
class Rectangle {
public:
    Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
    Rectangle(const Rectangle& obj);
    ~Rectangle();
    // Note: il faudrait aussi redéfinir operator= !
private:
    double* largeur;    double* hauteur;
};

// constructeur de copie
Rectangle::Rectangle(const Rectangle& obj)
    : largeur(new double(*(obj.largeur))),
      hauteur(new double(*(obj.hauteur)))
{}

// destructeur
void Rectangle::~~Rectangle() {
    delete largeur;
    delete hauteur;
}
```

Il faudra aussi **penser à redéfinir l'opérateur =**

Pour conclure

- ▶ Si une classe contient des pointeurs, penser à la copie profonde (**au moins** se poser la question) :
 - ▶ constructeur de copie ;
 - ▶ surcharge de l'opérateur = ;
 - ▶ destructeur.
- ▶ Remarque : si l'on redéfinit le constructeur de copie d'une sous-classe, penser à explicitement mettre l'appel au constructeur **de copie** de la super-classe (sinon c'est le constructeur **par défaut** de la super-classe qui est appelé !!)

Exemple :

```
Rectangle(const Rectangle& obj)
: FigureGeometrique(obj), // ...etc.
{ // ...etc.
}
```