LANGAGE C++

L'héritage

Principe

Après les notions d'encapsulation et d'abstraction, le troisième aspect essentiel de la « P.O.O. » est la notion d'héritage.

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

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

Exemples :

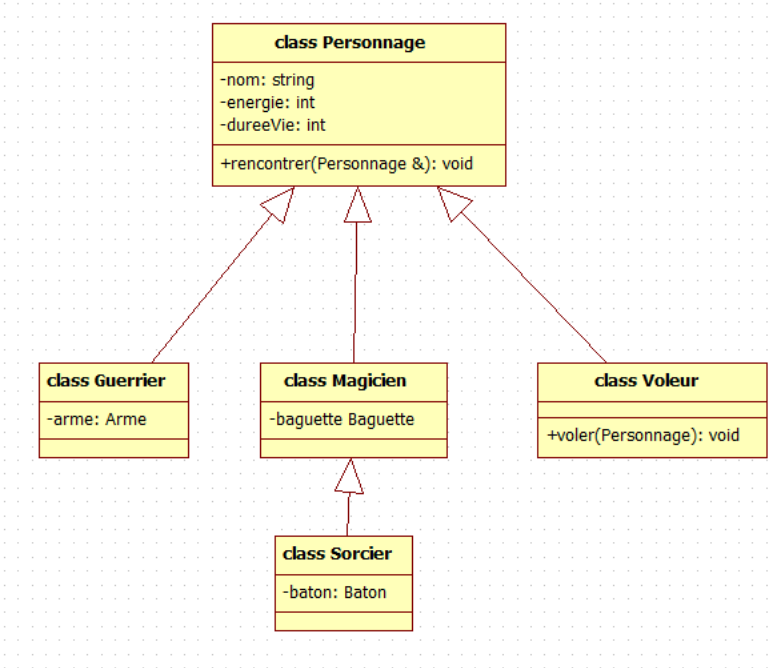
- une voiture est un véhicule (Voiture hérite de Vehicule) ;
- un bus est un véhicule (Bus hérite de Vehicule) ;
- un moineau est un oiseau (Moineau hérite d'Oiseau) ;
- un corbeau est un oiseau (Corbeau hérite d'Oiseau) ; un chirurgien est un docteur (Chirurgien hérite de Docteur) ;
- Une capitale est une ville (Capitale hérite de Ville).

Une sous-classe hérite des propriétés de la classe de base (attributs et méthodes) qu'elle pourra modifier et compléter.

Exemple :

Classes pour les personnages d'un jeu vidéo

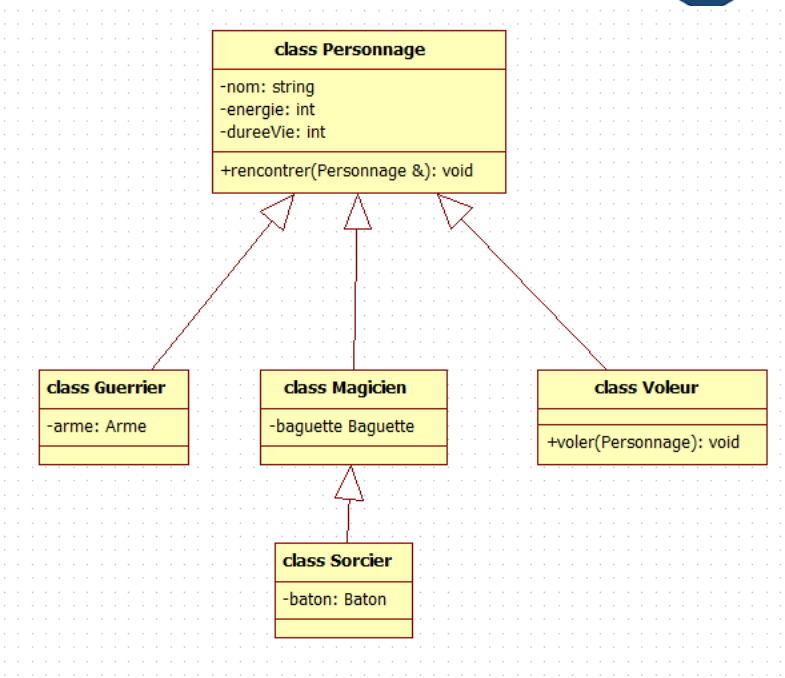
- **Notation UML** :



Exemple :

Classes pour les personnages d'un jeu vidéo

```
class Personnage {
    private:
        string nom;
        int energie;
        int duree_vie;
    public:
        // constructeurs, etc.
        void rencontrer(personnage&);
};
class Voleur : public Personnage {
    public:
        // constructeurs, etc.
        void voler(Peronnage&);
};
```



Syntaxe

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

```
class Magicien : public Personnage {
    public:
        // constructeurs, etc.
    private:
        Baguette baguette;
};
class Sorcier : public Magicien {
    public:
        // constructeurs, etc.
    private:
        Baton baton;
};
```

Exemple :

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

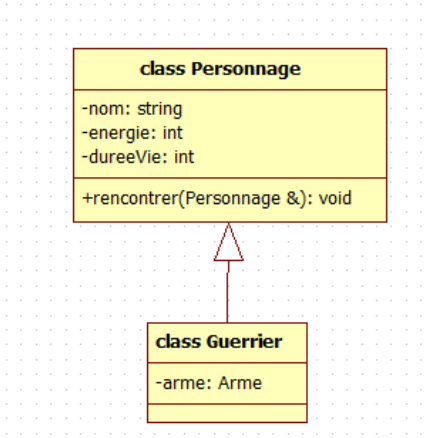
- C1 va hériter de l'ensemble des **attributs** de C, des **méthodes** de C (**sauf les constructeurs**)
- des attributs et/ou méthodes **supplémentaires** peuvent être définis par la sous-classe C1
  => enrichissement
- des méthodes héritées de C peuvent être **redéfinies** dans C1
  => spécialisation
- le **type** est hérité : un C1 **est** (aussi) **un C**

Nous ne copions de g dans p seulement sa partie personnage (attributs hérités).

Note : on ne peut pas écrire g = p (un personnage n'est pas un guerrier).

L'héritage est une relation orientée.

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




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

Héritage - Notion :

L'héritage permet:
- d'expliciter des relations structurelles et sémantiques entre classes
- de réduire les redondances de description et de stockage des propriétés

Par transitivité, les instances d'une sous-classe possèdent les attributs et méthodes (hors constructeurs) de l'ensemble des classes parentes (super-classe, super-super-classe, etc.) : ce réseau de dépendances définit une hiérarchie de classes

Rappel : transitif, se dit d'une relation $\mathcal{R}$ telle que : si (x $\mathcal{R}$ y) et si (y $\mathcal{R}$ z), alors (x $\mathcal{R}$ z)

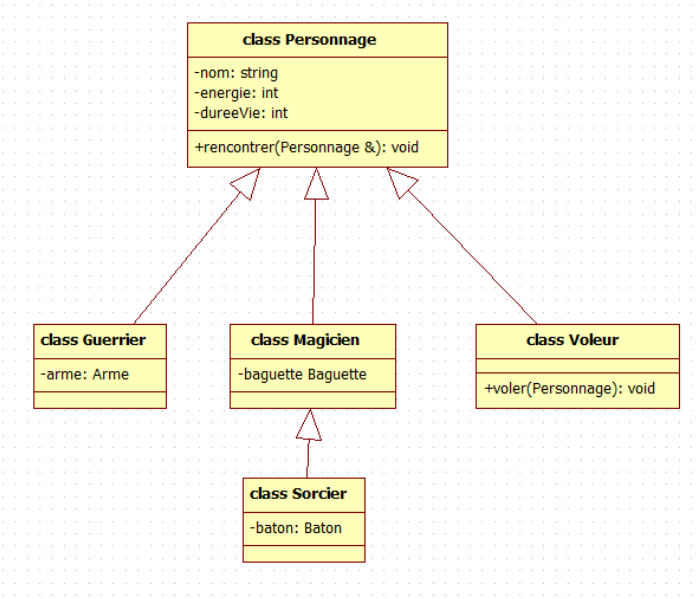
Dans une hiérarchie de classes :

- Un objet d'une sous-classe **hérite le type** de sa super-classe
- L'héritage est transitif
- Un objet peut donc avoir **plusieurs types**

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



=> On évite ainsi la duplication de code

Droit d'accès aux attributs :

public : les éléments qui suivent sont accessibles depuis l'extérieur de la classe ;
private : les éléments qui suivent ne sont pas accessibles depuis l'extérieur de la classe.

La portée **protected** (protégée) assure la visibilité des membres d'une classe dans les classes de sa descendance.

```
class Personnage {
    private:
        string nom;
        int duree_vie;
        int energie;
    //...
};
class Magicien : public Personnage {
    public:
        void frapper(Personnage& le_pauvre) {
            if (energie > 0) {
                cout << "frapper le perso";
            }
        }
   //...
};
```

```
class Personnage {
    private:
        string nom;
        int duree_vie;
    protected:
        int energie;
    //...
};
class Magicien : public Personnage {
    public:
        void frapper(Personnage& le_pauvre) {
            if (energie > 0) {
                cout << "frapper le perso";
            }
        }
   //...
};
```

```
Message
=== Build: Debug in heritage1 (compiler: GNU GCC Compiler) ==
In member function 'void Magicien::frapper(Personnage&)':
error: 'int Personnage::energie' is private
error: within this context
=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 s
```

Le niveau d'accès protégé correspond à une **extension du niveau privé** permettant l'accès aux sous-classes... **mais uniquement dans leur portée** (de sous-classe), et non pas dans la portée de la super-classe.

Exemple accès protégé :

```
class A {
    // ...
    protected:
        int a;
    private:
        int privee;
};
class B: public A {
    public:
        // ...
        void f(B autreB, A autreA, int x) {
            a = x;                  // OK A::a est protected => accès possible
            prive = x;              // Erreur : A::prive est private
            a += autreB.prive;      // Erreur (même raison)
            a += autreB.a ;         // OK : dans la même portée (B::)
            a += autreA.a ;         // INTERDIT ! : autreA pas a même portée
                                    //              que (B::)
        }
};
```

Autre exemple :

```
class Personnage {
    protected:
        int energie;
};
class Guerrier : public Personnage {
    public:
        void frapper(Personnage& p){
            if(p.energie > energie)
                cout << "Guerrier ne peut rien";
        }
};
```

```
Message
=== Build: Debug in heritage1 (compiler: GNU GCC Compiler) ===
In member function 'void Guerrier::frapper(Personnage&)':
error: 'int Personnage::energie' is protected
error: within this context
=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

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)

Redéfinition des méthodes:

Un personnage et un guerrier ont un comportement différent quand il rencontre un autre personnage. Par exemple, le personnage salue tandis que le guerrier frappe.

Pour un personnage non- Guerrier

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

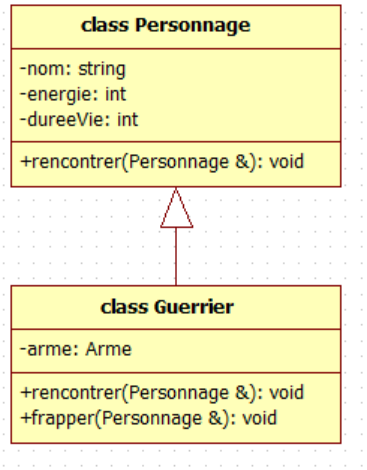
Pour un Guerrier

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

Une classe dérivée peut fournir une **nouvelle définition d'une méthode d'une classe ascendante**.
Les deux méthodes doivent avoir **des signatures identiques** (même nom, type des arguments et type de retour).

Masquage = une propriété redéfinie qui cache celle héritée

- Même **nom** d'attribut ou de méthode utilisé sur plusieurs niveaux
- **Peu** courant pour les attributs
- Très **courant** et **pratique** pour les méthodes

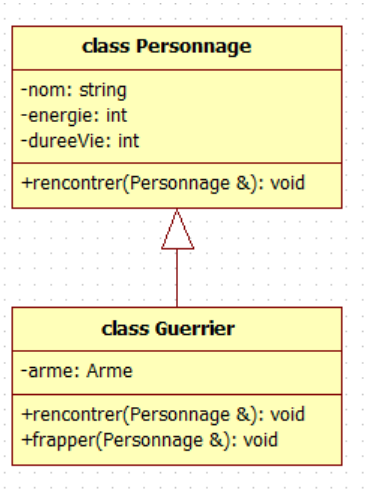


Masquage dans une hiérarchie :

La méthode rencontrer de Guerrier masque celle de Personnage

- Un objet de type Guerrier utilisera la méthode rencontrer de la classe Guerrier
- Méthode qui masque la méthode héritée = **méthode spécialisée**

Le masquage permet d'adapter une hiérarchie de classe à des comportements spécifiques

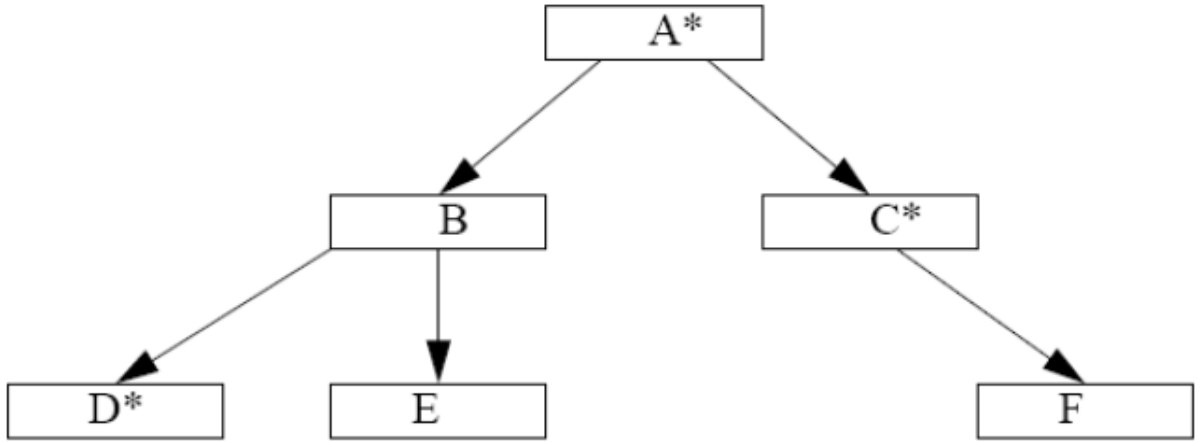



```
class Personnage {
    public:
        void rencontrer(Personnage& le_perso) const {
            cout << "Personnage vous salue" << endl;
        }
    // ...
};
class Guerrier : public Personnage {
    public:
        void rencontrer(Personnage& le_pauvre) const {
            cout << "Guerrier vous frappe" << endl;
        }
    // ...
};
```

```
class Magicien : public Personnage {
    private:
        int baguette;
    // ...
};
int main() {
    Guerrier guerrier;
    Magicien magicien;
    guerrier.rencontrer(magicien);
    magicien.rencontrer(guerrier);
    return 0;
}
```

Redéfinition de méthode et dérivations successives

La redéfinition d'une méthode s'applique à une classe et à toutes ses descendantes jusqu'à ce qu'éventuellement l'une d'entre elles redéfinisse à nouveau la méthode.



La présence d'un astérisque (*) signale la définition ou la redéfinition d'une méthode *f*

Classe de l'objet qui appel f	Méthode appelé
classe *A*	méthode *f* de *A*
classe *B*	méthode *f* de *A*
classe C	méthode *f* de *C*
classe *D*	méthode *f* de *D*
classe *E*	méthode *f* de *A*
classe *F*	méthode *f* de *C*

Accès à une méthode masquée

Dans certaines situations, Il est souhaitable d'accéder à une méthode masqué(e)

Le Guerrier commence par rencontrer le personnage comme le fait n'importe quel personnage (il le salue) avant de le frapper !

```
class Guerrier : public Personnage {
    public:
        void rencontrer(Personnage& le_pauvre) const {
            Personnage::rencontrer(le_pauvre);
            cout << "Guerrier vous frappe" << endl;
        }
    // ...
};
```

```
int main() {
    Guerrier guerrier;
    Magicien magicien;
    guerrier.rencontrer(magicien);
    return 0;
}
```

```
Personnage vous salue
Guerrier vous frappe

Process returned 0 (0x0)
Press any key to continue.
```

Pour accéder aux attributs/méthodes masqué(e)s
=> **on utilise l'opérateur de résolution de portée**

NomClasse::méthode ou *attribut*

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

L'accès aux attributs hérités pourrait notamment être interdit !

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

L'initialisation des attributs hérités doit se faire au niveau des classes où ils sont explicitement définis en invoquant les constructeurs des super-classes.

Appel explicite

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

```
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

invocation explicite obligatoire

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**

```
class FigureGeometrique {
    private:
        double abs;
        double ord;
    public:
    FigureGeometrique(double x, double y) : abs(x), ord(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) {}
    // ...
};
```

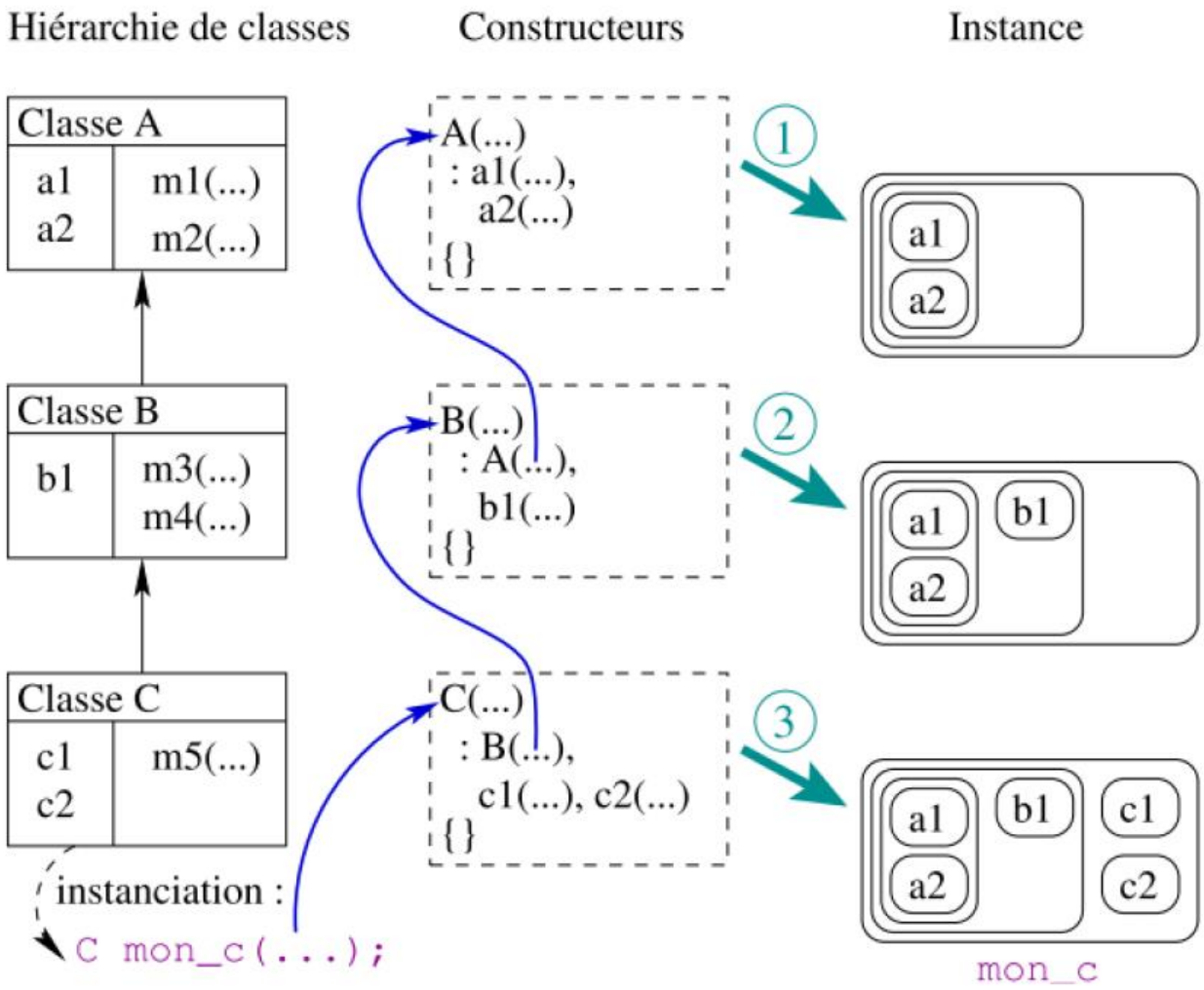
Autre exemple

```
class FigureGeometrique {
    private:
        double abs;
        double ord;
    // ...
};
```

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

Le constructeur de la classe FigureGeometrique est généré automatiquement par le compilateur

Ordre d'appel des constructeurs



Ordre d'appel des destructeurs

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

Dans l'exemple précédent, lors de la destruction de mon_C, on aura appel et exécution de C::~C() puis B::~B() et A::~A()

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 !

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

Copie de copie : contexte

Rappels

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.

- Dans certains cas, les versions minimales par défaut des méthodes constructeurs/destructeurs **ne sont pas adaptées**, exemple du comptage des instances.

- 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**.

Exemple

```
AVANT = (2.2, 4.6)
Largeur: 2.2
APRES = (1.13945e-305, 4.6)

Process returned 0 (0x0)   e
Press any key to continue.
```

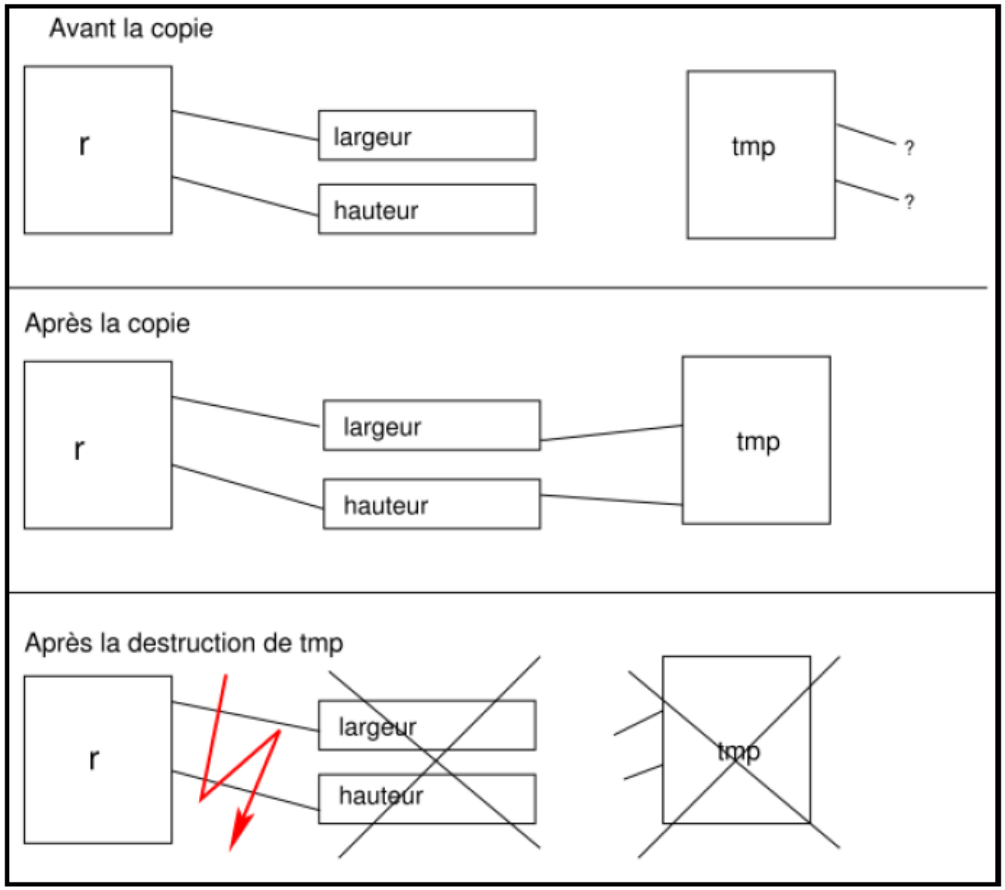
```
class Rectangle {
    private:
        double* largeur; // Pointeurs !
        double* hauteur;
    public:
        Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
        ~Rectangle() { delete largeur; delete hauteur; }
        double getLargeur() const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

```
void afficher_largeur(Rectangle tmp) {
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
int main() {
    Rectangle rect(2.2,4.6);
    cout << "AVANT = (" << rect.getLargeur() << ", " << rect.getHauteur() << ")" << endl;
    afficher_largeur(rect);
    cout << "APRES = (" << rect.getLargeur() << ", " << rect.getHauteur() << ")" << endl;
    return 0;
}
```

- Le paramètre tmp de la fonction afficher_largeur est **passé par valeur**
  => appel du constructeur de copie
- La copie est effectuée membre à membre (**copie superficielle**)
- A la fin de la fonction afficher_largeur, la variable locale tmp est **détruite**
  => appel du destructeur

- **Désallocation** des espaces mémoire pointées par largeur et hauteur de tmp
  => Segmentation Fault

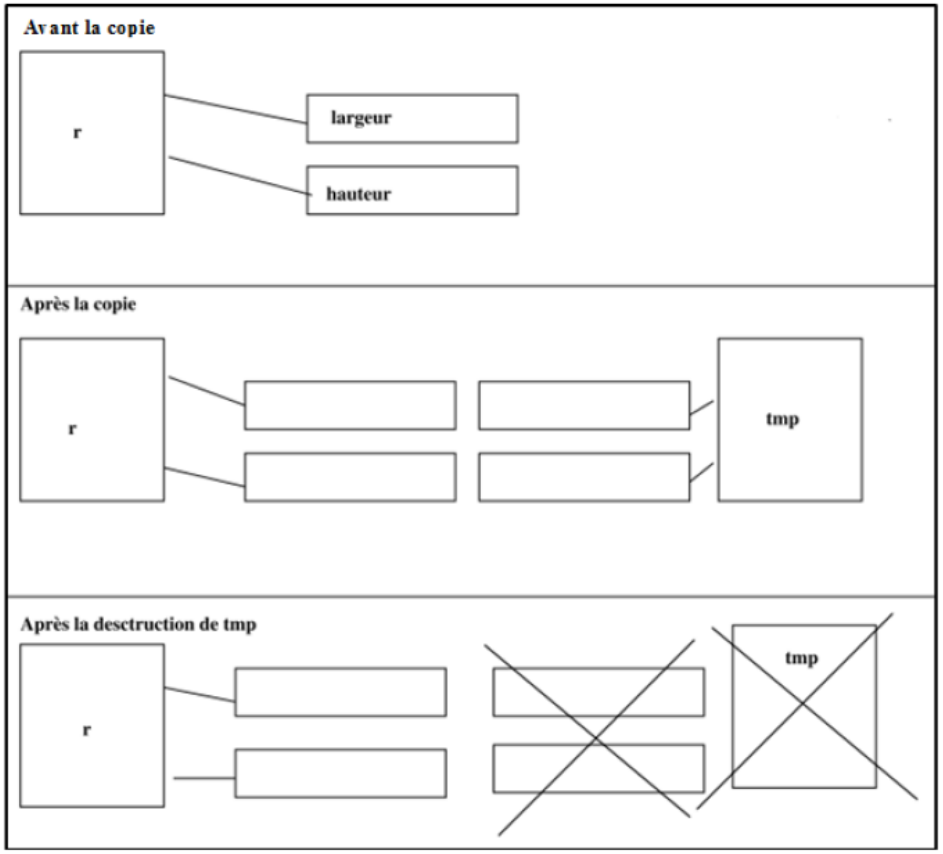
Etat mémoire



Il faut redéfinir le constructeur de copie de sorte à ce qu'il duplique véritablement les champs
=> concernés copie profonde

Copie profonde

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



Exemple complet

```
AVANT = (2.2, 4.6)
Largeur: 2.2
APRES = (2.2, 4.6)

Process returned 0 (0x0)
Press any key to continue.
```

```
class Rectangle {
    private:
        double* largeur; // Pointeurs !
        double* hauteur;
    public:
        Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
        ~Rectangle() { delete largeur; delete hauteur; }
        Rectangle(const Rectangle& obj)
        : largeur(new double(*(obj.largeur))) ,
          hauteur(new double(*(obj.hauteur)))
        {}
        // Note: il faudrait aussi redefinir operator= !
        double getLargeur() const { return *largeur; }
        double getHauteur() const { return *hauteur; }
};
```

```
void afficher_largeur(Rectangle tmp) {
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
int main() {
    Rectangle rect(2.2,4.6);
    cout << "AVANT = (" << rect.getLargeur() << ", " << rect.getHauteur() << ")" << endl;
    afficher_largeur(rect);
    cout << "APRES = (" << rect.getLargeur() << ", " << rect.getHauteur() << ")" << endl;
    return 0;
}
```

Conclusion

Si une classe contient des pointeurs, pensez à **la copie profonde** (au moins se poser la question) :

- constructeur de copie ;
- surcharge de l'opérateur = ;
- destructeur.