



Le langage C++

PROGRAMMATION ORIENTÉE OBJET POO

Notions d'encapsulation

Le **principe d'encapsulation** consiste à **regrouper** dans le même objet informatique («concept»), données et traitements qui lui sont **spécifiques** :

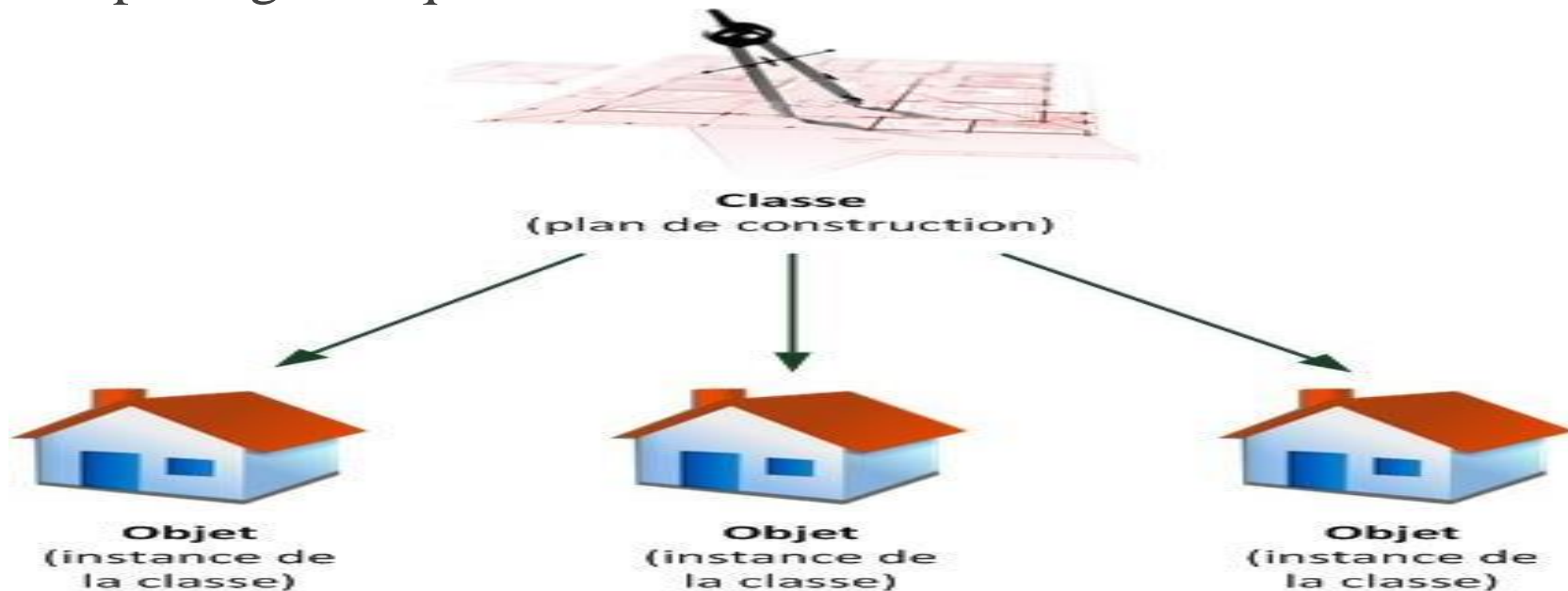
- Les données incluses dans un objet seront appelées les **attributs** de cet objet ;
- Les traitements/fonctions défini(e)s dans un objet seront appelées les **méthodes** de cet objet.

Résumé du premier principe de l'encapsulation

OBJET
=
attributs
+
méthodes

Notion d'abstraction

Les classes sont une "abstraction". Les objets peuvent correspondre à des objets du monde réel, les classes correspondent à une entité abstraite de niveau supérieur, elle est une description générique de l'ensemble considéré.



Encapsulation et Interface

Il y a donc deux facettes à l'encapsulation :

1. regroupement de tout ce qui caractérise l'objet : données (attributs) **et** traitements (méthodes)

2. isolement et dissimulation des détails d'implémentation **Interface** = ce que le programmeur-utilisateur (hors de l'objet) peut utiliser

☞ Concentration sur les attributs/méthodes concernant l'objet (*abstraction*)

Exemple : **L'interface d'une voiture**

▲ Volant, accélérateur, pédale de freins, etc.

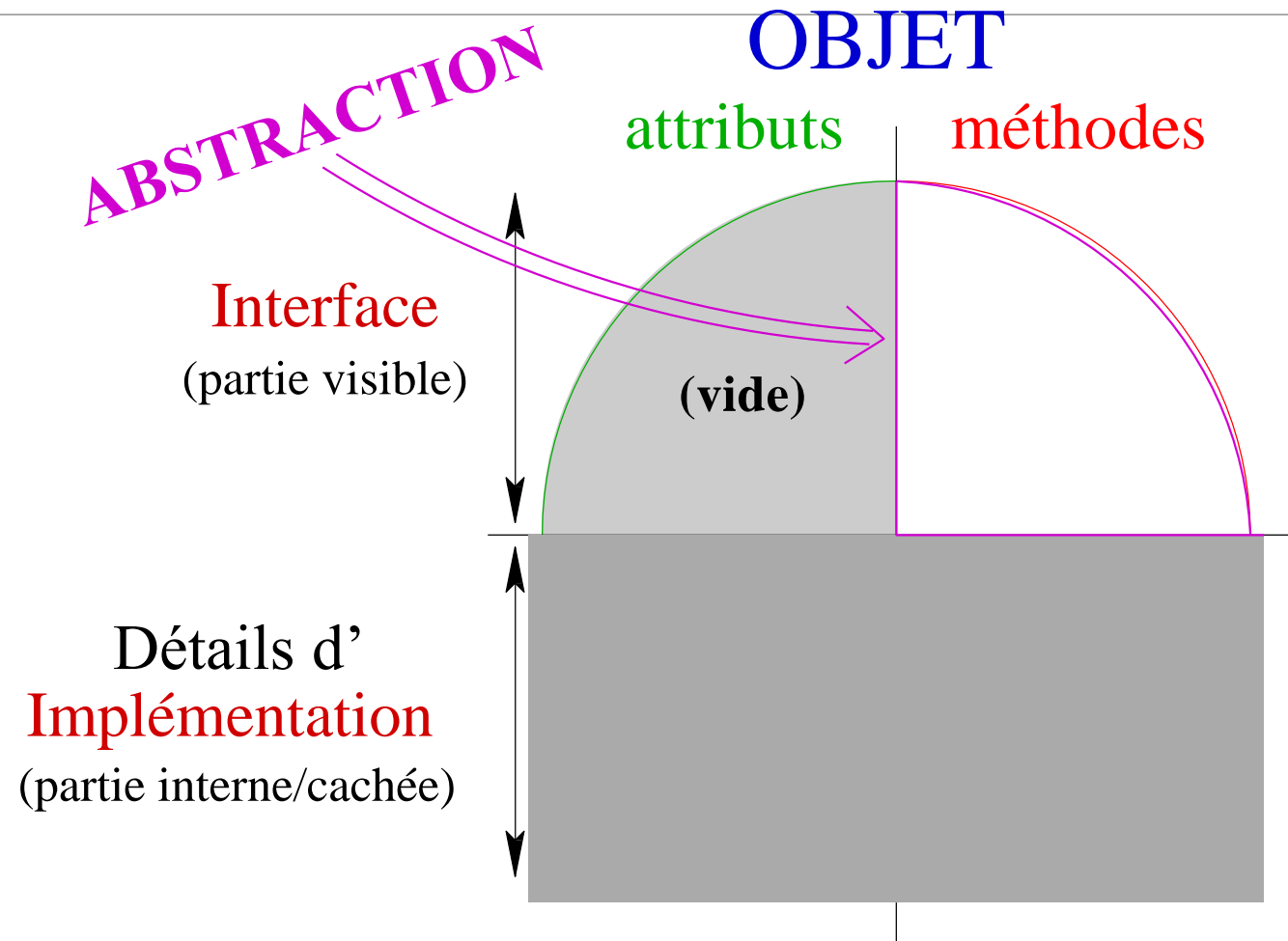
▲ Tout ce qu'il faut savoir pour la conduire (mais pas la réparer ! ni comprendre comment ça marche)

▲ L'interface ne change pas, même si l'on change de moteur...

...et même si on change de voiture (dans une certaine mesure) : *abstraction* de la notion de voiture (en tant qu'« objet à conduire »)

Encapsulation / Abstraction : Résumé

MIEUX :



Les classes en C++

- Pour créer une classe, on utilise le **mot-clé class suivi du nom de la classe** qui commence par une **majuscule**, qui n'est pas obligatoire mais ceci rend la distinction entre les noms des classes et des objets plus simple. La définition de la classe s'écrit entre les accolades. On liste les attributs puis les méthodes dont a besoin la classe. Il y a un **point-virgule après l'accolade fermante**.
- **Une classe est constituée de** variables appelées **attributs** (ou variables membres) et de fonctions appelées **méthodes** (ou fonctions membres).

Les classes en C++

- Chaque **attribut** et chaque **méthode** d'une classe peut **posséder son propre droit d'accès**. On peut citer les deux droits d'accès :
 - public: l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
 - private: l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet. **Par défaut, tous les éléments d'une classe sont private.**
- Les mots clés **public** et **private** peuvent **apparaître à plusieurs reprises** dans la définition d'une classe.

Les classes en C++

Une classe se déclare par le mot-clé class
(définit un nouveau type de variable)

Déclaration des attributs

Syntaxe

```
type nomAttribut;
```

L'accès aux valeurs des attributs d'une instance

Exemple

```
class Rectangle {  
    ...  
};
```

```
class Rectangle {  
    double hauteur;  
    double largeur;  
};
```

```
nom_instance.nom_attribut
```

Définition des méthodes

Syntaxe

```
typeRetour nomMethode (typeParam1 nomParam1, ...)  
{  
    // corps de la méthode  
    ...  
}
```

Exemple

```
class Rectangle {  
    double hauteur;  
    double largeur;  
    double surface() {  
        return hauteur * largeur;  
    }  
};
```

Portée des attributs

Les attributs d'une classe constituent des variables **directement accessibles dans toutes les méthodes** de la classe. On parle de « portée de classe » (« variables globales à la classe »).

=> Il n'est donc pas nécessaire de les passer comme arguments des méthodes.

Les méthodes sont :

- des fonctions propres à la classe
- qui ont accès aux attributs de la classe

Déclaration d'une instance (un objet)

(se fait de façon similaire à la déclaration d'une variable)

```
nom_classe nom_instance;
```

Exemple

```
Rectangle rect;
```

Les droits d'accès

Chaque attribut et chaque méthode d'une classe peut posséder son propre droit d'accès.

- **public** : l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
- **private** : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet.

```
class Rectangle {  
    public:  
        double hauteur;  
        double largeur;  
        double surface() {  
            return hauteur * largeur;  
        }  
};
```

Appels aux méthodes

Syntaxe

```
nom_instance.nom_methode(val_arg1, ...)
```

Exemple

```
Rectangle rect;  
rect.surface();
```

Méthodes «get» et «set» (« accesseurs » et « manipulateurs »)

- △ Tous les attributs sont privés ?
- △ Et si on a besoin de les utiliser depuis l'extérieur de la classe ?!
- △ Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...

1. Manipulateurs (« méthodes set ») :

- △ Modification (action)
- △ Affectation de l'argument à une variable d'instance précise

```
void setHauteur(double h) { hauteur = h;}
```

```
void setLargeur(double L) { largeur = L;}
```

2. Accesseurs (« méthodes get ») :

- △ Consultation (prédicat)
- △ Retour de la valeur d'une variable d'instance précise

```
double getHauteur() const { return hauteur;}
```

```
double getLargeur() const { return largeur;}
```

Notre programme:

```
#include <iostream>
using namespace std;

class Rectangle {
public:
double surface() const{ return hauteur * largeur; }

double getHauteur() const{ return hauteur; }
double getLargeur() const{ return largeur; }

void setHauteur(double h){ hauteur = h; }
void setLargeur(double l){ largeur = l; }
```

```
private:
double hauteur; double largeur;
};

int main()
{
Rectangle rect1;

rect1.setHauteur(3.0); rect1.setLargeur(4.0);

cout << "hauteur : "
<< rect1.getHauteur()
<< endl;

return 0;
}
```

Fonction Amie

Une fonction amie d'une classe est définie en dehors de la portée de cette classe, mais elle a le droit d'accéder à tous les membres privés et protégés de la classe. Même si les prototypes des fonctions d'amis apparaissent dans la définition de la classe, les amis ne sont pas des fonctions membres.

Pour déclarer une fonction en tant qu'une amie d'une classe, faites précéder le prototype de fonction dans la définition de la classe avec le mot-clé **friend** comme suit :

```
class nomClasse{  
  
    ...  
  
    friend type_retour nomFonction(liste des paramètres);  
  
};  
  
type_retour nomFonction(liste des paramètres){  
  
    // Les données privées et protégées de nomClasse sont accessibles  
  
    // à partir de cette fonction car il s'agit d'une fonction amie de nomClasse.  
  
}
```

Classe Amie

De même, à l'instar d'une fonction amie, une classe peut également être transformée en amie d'une autre classe à l'aide du mot clé **friend**. Par exemple:

```
...class B;  
  
class A{.....  
    friend class B;};  
  
class B{.....  
  
}...
```

Lorsqu'une classe devient une classe amie, toutes les fonctions membres de cette classe deviennent des fonctions amies.

Dans le programme ci-dessus, toutes les fonctions membres de la classe B deviennent des fonctions amies de la classe A. Ainsi, toute fonction membre de la classe B peut accéder aux données privées et protégées de la classe A. Toutefois, les fonctions membres de la classe A ne peuvent pas accéder aux données de classe B.

Constructeur / Destructeur

- A priori, les **objets** suivent les règles habituelles concernant leur **initialisation par défaut**. Il est donc nécessaire de faire appel à une méthode pour attribuer des valeurs aux données d'un objet, comme la méthode initialise de la classe Point.
- Cependant, une telle démarche oblige à **compter sur l'utilisateur** de l'objet pour effectuer l'appel voulu au bon moment. En outre, L'**absence** de procédure d'initialisation peut être **catastrophique** dans le cas où l'objet doit **allouer dynamiquement de la mémoire**. Le C++ offre un mécanisme très performant pour traiter ces problèmes : **le constructeur**.

Constructeur / Destructeur

- Le **constructeur** est une méthode qui sert à construire un objet de la classe en question. Il est appelé automatiquement à chaque création d'un objet basé sur cette classe. Un objet pourra aussi posséder un **destructeur**, une méthode appelée automatiquement au moment de la destruction de l'objet.
- Les **objets automatiques**, auxquels on se limite jusqu'à présent, sont **créés par une déclaration** et leur destruction a lieu lorsque **l'on quitte** le bloc ou la fonction **où ils ont été déclaré**.
- Par convention, le **constructeur** se reconnaît à ce qu'il porte le **même nom que la classe**. Quant au **destructeur**, il porte le **même nom que la classe précédé d'un tilde ~**.

Constructeur / Destructeur

- A priori, les **objets** suivent les règles habituelles concernant leur **initialisation par défaut**. Il est donc nécessaire de faire appel à une méthode pour attribuer des valeurs aux données d'un objet, comme la méthode initialise de la classe Point.
- Cependant, une telle démarche oblige à **compter sur l'utilisateur** de l'objet pour effectuer l'appel voulu au bon moment. En outre, L'**absence** de procédure d'initialisation peut être **catastrophique** dans le cas où l'objet doit **allouer dynamiquement de la mémoire**. Le C++ offre un mécanisme très performant pour traiter ces problèmes : **le constructeur**.

Constructeur / Destructeur

- Le **constructeur** est une méthode qui sert à construire un objet de la classe en question. Il est appelé automatiquement à chaque création d'un objet basé sur cette classe. Un objet pourra aussi posséder un **destructeur**, une méthode appelée automatiquement au moment de la destruction de l'objet.
- Les **objets automatiques**, auxquels on se limite jusqu'à présent, sont **créés par une déclaration** et leur destruction a lieu lorsque **l'on quitte** le bloc ou la fonction **où ils ont été déclaré**.
- Par convention, le **constructeur** se reconnaît à ce qu'il porte le **même nom que la classe**. Quant au **destructeur**, il porte le **même nom que la classe précédé d'un tilde ~**.

Constructeur (Syntaxe)

Pour faire ces initialisations, il existe en C++ des méthodes particulières appelées **constructeurs**.

Un constructeur est une méthode :

- invoquée **automatiquement** lors de la déclaration d'un objet
- chargée d'effectuer toutes les opérations requises en « **début de vie** » de l'objet (dont l'initialisation des attributs)

Syntaxe

```
NomClasse(liste_paramètres)  
{  
    /* initialisation des attributs  
       en utilisant liste_paramètres */  
}
```

- pas de type de retour (pas même void)
- même nom que la classe

Exemple

```
Rectangle(double h, double L) {  
    hauteur = h;  
    largeur = L;  
}
```

Comme les autres méthodes :

- les constructeurs peuvent être surchargés
- on peut donner des valeurs par défaut à leurs paramètres

Une classe peut donc avoir plusieurs constructeurs (listes de paramètres différentes).

Initialisation par constructeur

```
NomClasse instance(valarg1, ..., valargN);
```

Exemple

```
Rectangle rect(4.2, 8.4);
```

Exemple

```
#include <iostream>
using namespace std;
class Rectangle {
public:
    Rectangle(double h, double L) {
        hauteur = h;
        largeur = L;
    }
    double surface() const {
        return hauteur * largeur;
    }
private:
    double hauteur;
    double largeur;
};
```

```
int main() {
    double luH, luL;
    cout << "Quelle hauteur ? : ";
    cin >> luH;
    cout << "Quelle largeur ? : ";
    cin >> luL;
    Rectangle rect(luH, luL);
    cout << "surface = " << rect.surface();
    return 0;
}
```

```
Quelle hauteur ? : 4.2
Quelle largeur ? : 8.6
surface = 36.12
Process returned 0 (0x0)
Press any key to continue.
```

Appel aux constructeurs des attributs

Liste d'initialisation

Que se passe-t-il si les attributs sont eux-mêmes des objets ?

```
class RectangleColore {
    // ...
private:
    Rectangle rectangle;
    int couleur;
};
```

Mauvaise solution :

```
RectangleColore(double h, double L, int c){
    rectangle = Rectangle(h, L);
    couleur = c;
}
```

Le constructeur d'un rectangleColore devrait faire appel au constructeur de Rectangle, ce qui n'est pas possible directement. On peut alors imaginer passer par une instance anonyme intermédiaire (Mauvais)

=> Il faut initialiser directement les attributs en faisant directement appel à leurs propres constructeurs !

Syntaxe générale

```
NomClasse(liste_paramètres)
// liste d'initialisation
: attribut1(...), // appel au constructeur de attribut1
...
attributN(...) // appel au constructeur de attributN
{ // autres opérations }
```

Exemple

```
RectangleColore(double h, double L, Couleur c)
: rectangle(h, L), couleur(c)
{ }
```

Un constructeur devrait normalement contenir une section d'appel aux constructeurs des attributs.

Ceci est également valable pour l'initialisation des attributs de type de base.

Il s'agit de la

« section deux-points »

des constructeurs :

Cette section introduite par « : » est optionnelle mais **recommandée**.

Exemple

```
#include <iostream>
using namespace std;
class Rectangle {
public:
    Rectangle(double h, double L)
        : hauteur(h), largeur(L)
    {}
    double surface() const {
        return hauteur * largeur;
    }
private:
    double hauteur;
    double largeur;
};
```

```
int main() {
    double luH, luL;
    cout << "Quelle hauteur ? : ";
    cin >> luH;
    cout << "Quelle largeur ? : ";
    cin >> luL;
    Rectangle rect(luH, luL);
    cout << "surface = " << rect.surface();
    return 0;
}
```

Remarque :

Les attributs non-initialisés :

- prennent une valeur par défaut si ce sont des objets
- restent indéfinis s'ils sont de type de base

Les attributs initialisés dans la liste d'initialisation peuvent être changés dans le corps du constructeur.

```
Rectangle(double h, double L)
: hauteur(h) //initialisation
{
    // largeur a une valeur indéfinie jusqu'ici
    largeur = 2.0 * L + h; // par exemple...
    // la valeur de largeur est définie à partir d'ici
}
```

Constructeur par défaut

Remarque

```
public:
    Rectangle(double hauteur, double largeur)
    : hauteur(hauteur), largeur(largeur) // plus d'ambiguïté
    {}
```

Instanciation d'objets et appels aux constructeur

```
class Rectangle {
public:
    // Le constructeur par défaut
    Rectangle() : hauteur(1.2), largeur(2.6)
    {}
    // 2ème constructeur
    Rectangle(double c) : hauteur(c), largeur(2.0*c)
    {}
    // 3ème constructeur
    Rectangle(double h, double L) : hauteur(h), largeur(L)
    {}
    double surface() const {
        return hauteur * largeur;
    }
private:
    double hauteur;
    double largeur;
};
```

```
surface 1 = 3.12
surface 2 = 11.52
surface 3 = 50.88

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

```
int main() {
    Rectangle rect1;
    cout << "surface 1 = " << rect1.surface() << endl;
    Rectangle rect2(2.4);
    cout << "surface 2 = " << rect2.surface() << endl;
    Rectangle rect3(4.8,10.6);
    cout << "surface 3 = " << rect3.surface() << endl;
    return 0;
}
```

Constructeur par défaut

- Le constructeur par défaut est un constructeur qui n'a **pas de paramètre**
- ou dont **tous les paramètres** ont des **valeurs par défaut**.

Exemples

```
// avec liste d'initialisation
Rectangle(): hauteur(2.4), largeur(8.4)
{}

// sans liste d'initialisation
Rectangle() {
    hauteur = 0.0;
    largeur = 0.0;
}
```

Autre façon de faire : regrouper le premier constructeur en utilisant les valeurs par défaut des arguments :

```
// constructeur dont le constructeur par défaut
Rectangle(double c = 1.0) : hauteur(c),
    largeur(2.0*c)
{ }
```

A

```
class Rectangle {
    private:
        double h; double L;
    // suite ...
};
```

B

```
class Rectangle {
    private:
        double h; double L;
    public:
        Rectangle()
        : h(0.0), L(0.0)
        {}
    // suite ...
};
```

C

```
class Rectangle {
    private:
        double h; double L;
    public:
        Rectangle(double h=0.0,
        double L=0.0)
        : h(h), L(L)
        {}
    // suite ...
};
```

D

```
class Rectangle {
    private:
        double h; double L;
    public:
        Rectangle(double h,
        double L)
        : h(h), L(L)
        { }
    // suite ...
};
```

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);				
(A)	constructeur par défaut par défaut	<table border="1"><tr><td>?</td><td>?</td></tr></table>	?	?	Illicite !		
?	?						
(B)	constructeur par défaut explicitement déclaré	<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	Illicite !		
0	0						
(C)	un des trois constructeurs est par défaut	<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2
0	0						
1	2						
(D)	pas de constructeur par défaut	Illicite !	<table border="1"><tr><td>1</td><td>2</td></tr></table>	1	2		
1	2						

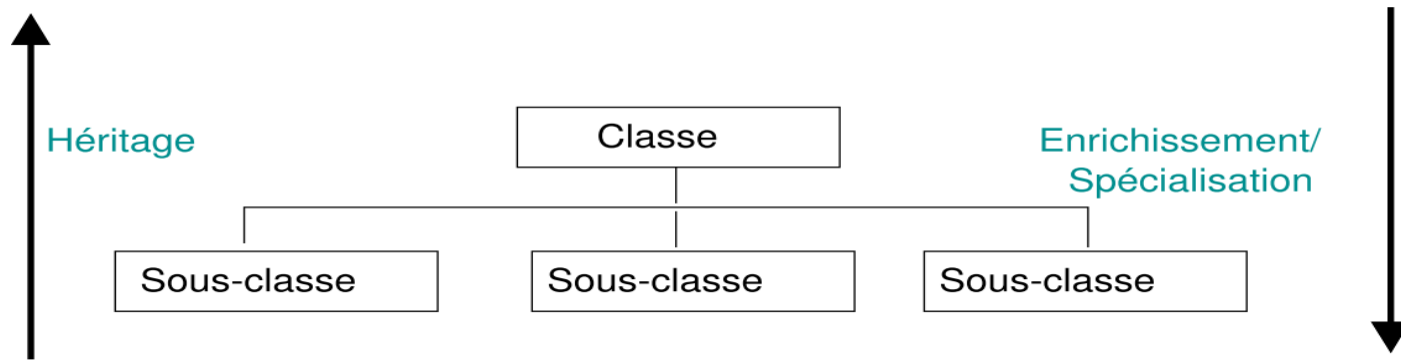
Héritage - Notion

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**, à partir de classes plus générales déjà existantes, appelées **super-classes**.

Exemples :

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

- 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).

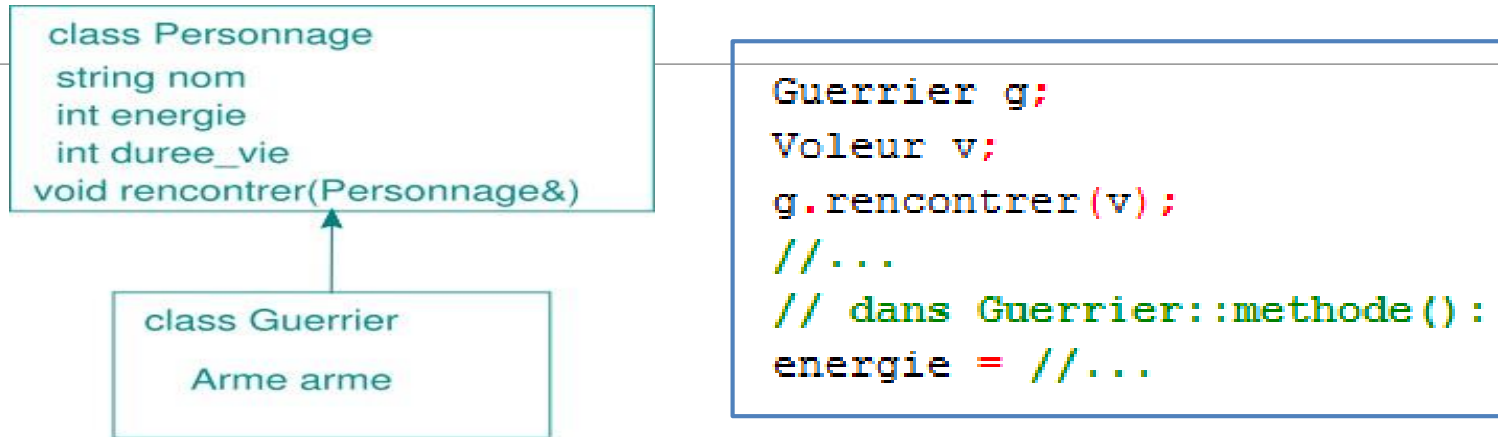


Une sous-classe hérite des propriétés de la classe de base (attributs et méthodes) qu'elle pourra modifier et compléter.

Héritage - Notion

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 écrire `g = p` (un personnage n'est pas un guerrier).
L'héritage est une relation orientée.

```

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

Le principe de l'héritage

- ❑ Le concept d'héritage constitue l'un des fondements de la Programmation Orientée Objet.

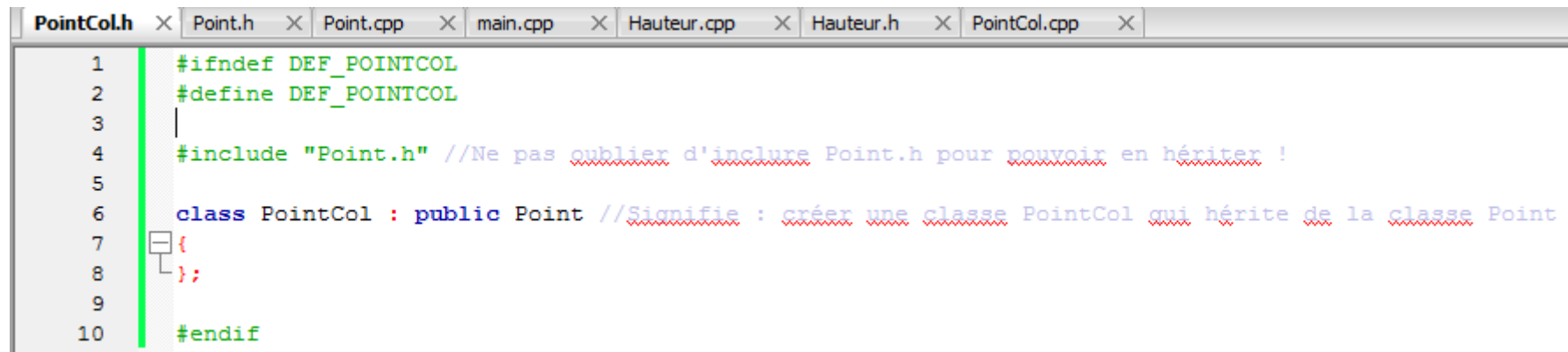
Il permet de **définir une nouvelle classe B dite dérivée, à partir d'une classe existante A, dite de base**. Pour ce faire, on procède ainsi:

```
Class B : public A // ou : private A // ou : protected A {  
    //définition de nouveaux membres (attributs ou méthodes)  
    //ou redéfinition de membres existants dans A.  
};
```

- ❑ Avec public A, on parle de **dérivation publique**; avec private A, on parle de **dérivation privée** et avec protected A, on parle de **dérivation protégée**.

Le principe de l'héritage

- Créant une nouvelle classe PointCol qui soit une sous-classe de Point. On dit que cette calsse **Point Col Hérite de Point**.



```
1  #ifndef DEF_POINTCOL
2  #define DEF_POINTCOL
3
4  #include "Point.h" //Ne pas oublier d'inclure Point.h pour pouvoir en hériter !
5
6  class PointCol : public Point //Signifie : créer une classe PointCol qui hérite de la classe Point
7  {
8  };
9
10 #endif
```

- La classe **PointCol** contient de base tous les attributs et toutes les méthodes de la classe **Point**. Dans un tel cas, la classe **Point** est appelée **la classe Mère** et la classe **PointCol** la classe **Fille**.

Le principe de l'héritage

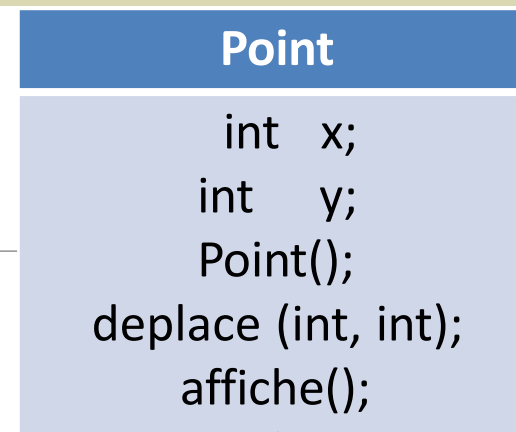
Ajoutant des attributs et des méthodes spécifiques à la classe fille PointCol.

- Un **attribut cl** de type int contenant la couleur d'un point;
- Une **méthode affiche (redéfinition)** qui affiche les coordonnées et la couleur d'un objet de type PointCol;
- Une **méthode colore (int couleur)** qui permet de définir la couleur d'un objet de type PointCol;
- Un **constructeur** permettant de définir la couleur et les coordonnées.

Le principe de l'héritage

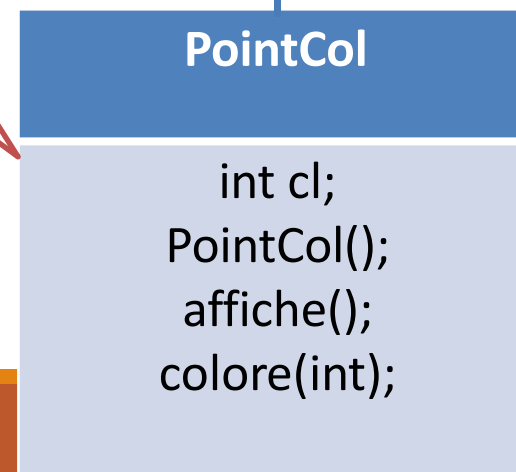
La classe PointCol
contient 3 attributs : x, y
(hérités) et cl. Elle
contient aussi 3
méthodes : deplace
(héritée), affiche
(surdéfinie) et colore.

Un objet de la classe PointCol
est un Point amélioré qui
possède un attribut et des
méthodes supplémentaires



Classe Mère
/ de base

Hérite de



Classe Fille /
dérivée

Le principe de l'héritage

```
PointCol.h X PointCol.cpp X main.cpp X Point.h X Point.cpp X Hauteur.cpp X Hauteur.h X
1  #include "Point.h"
2  #include "PointCol.h"
3  #include <iostream>
4
5  using namespace std;
6
7  PointCol :: PointCol(int abs, int ord, int ht, int c) : Point(abs, ord, ht), cl(c){
8
9  }
10 void PointCol :: affiche()const {
11     Point :: affiche();
12     cout << "couleur : " << cl << endl;
13 }
14 void PointCol :: colore(int couleur){
15     cl = couleur;
16 }
17
```

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";
            }
        }
        //...
};
```

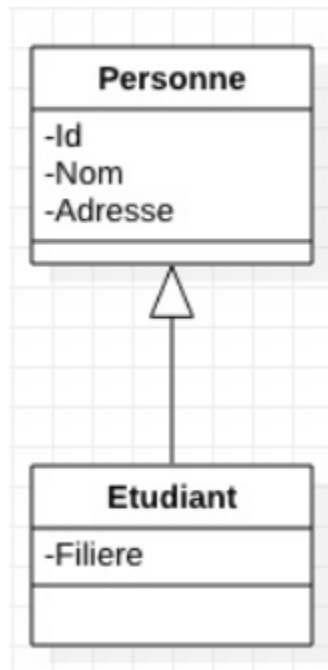
Message

```
=== Build: Debug in heritagel (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.

Types d'héritage

HÉRITAGE SIMPLE



DANS CE TYPE D'HÉRITAGE, UNE CLASSE DÉRIVÉE HÉRITE D'UNE SEULE CLASSE DE BASE. C'EST LA FORME LA PLUS SIMPLE DE L'HÉRITAGE.

```
class Personne
{
    private:
        int Id;
        char Nom[30];
        char Adresse[100];

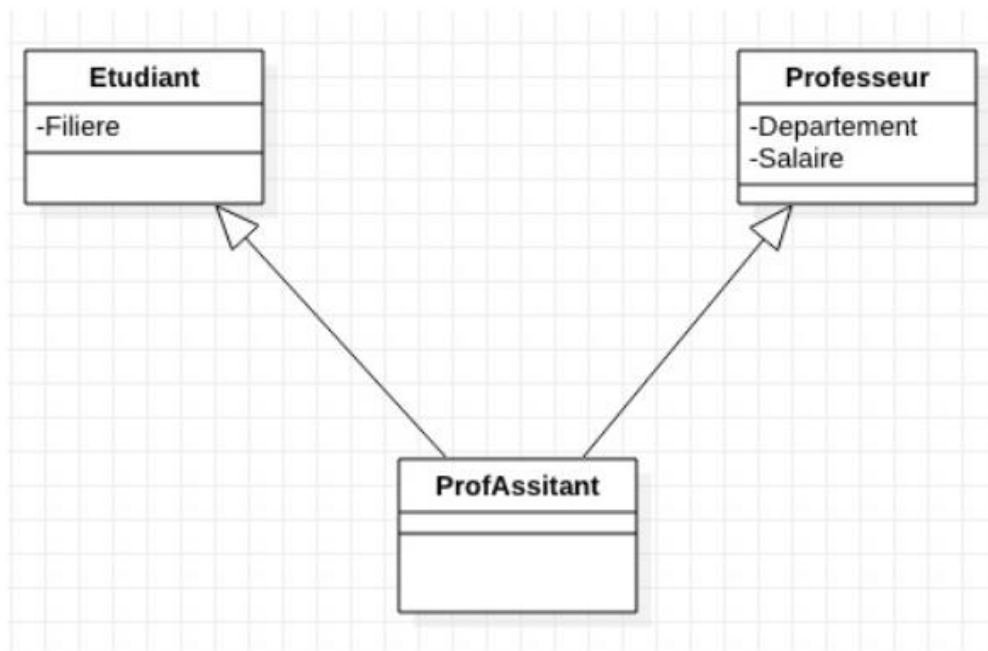
    public:
        Personne(int, char[], char[]);
        void afficher();
};

class Etudiant : public Personne
{
    private:
        char Filiere[40];

    public:
        Etudiant(int, char[], char[], char[]);
};
```

Types d'héritage

HÉRITAGE MULTIPLE

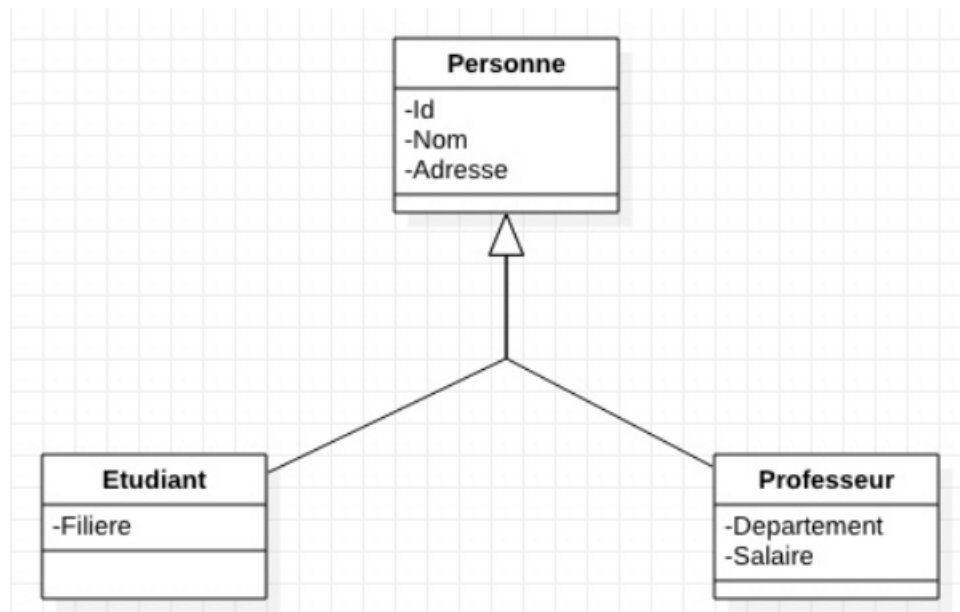


DANS CE TYPE D'HÉRITAGE, UNE SEULE CLASSE DÉRIVÉE PEUT HÉRITER DE DEUX OU PLUS DE DEUX CLASSES DE BASE.

```
class ProfAssistant : public Etudiant, public Professeur
{
};
```

Types d'héritage

HÉRITAGE HIÉRARCHIQUE



DANS CE TYPE D'HÉRITAGE, UNE CLASSE DE BASE A PLUS D'UNE CLASSE ENFANT. PAR EXEMPLE:

```
class Personne
{
private:
    int Id;
    char Nom[30];
    char Adresse[100];
```

```
public:
    Personne(int, char[], char[]);
    void afficher();
};
```

```
class Professeur : public Personne
{
private:
    double salaire;
    char departement[40];

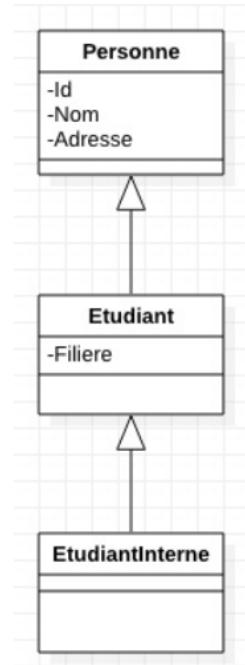
public:
    Professeur(int, char[], char[], double, char[]);
};
```

```
class Etudiant : public Personne
{
private:
    char Filiere[40];

public:
    Etudiant(int, char[], char[], char[]);
};
```

Types d'héritage

HÉRITAGE À PLUSIEURS NIVEAUX



DANS CE TYPE D'HÉRITAGE, LA CLASSE DÉRIVÉE HÉRITE D'UNE CLASSE, QUI HÉRITE À SON TOUR D'UNE AUTRE CLASSE. LA CLASSE DE BASE POUR L'UNE EST LA SOUS-CLASSE POUR L'AUTRE

```
class Personne
{
    private:
        int Id;
        char Nom[30];
        char Adresse[100];

    public:
        Personne(int, char[], char[]);
        void afficher();
};

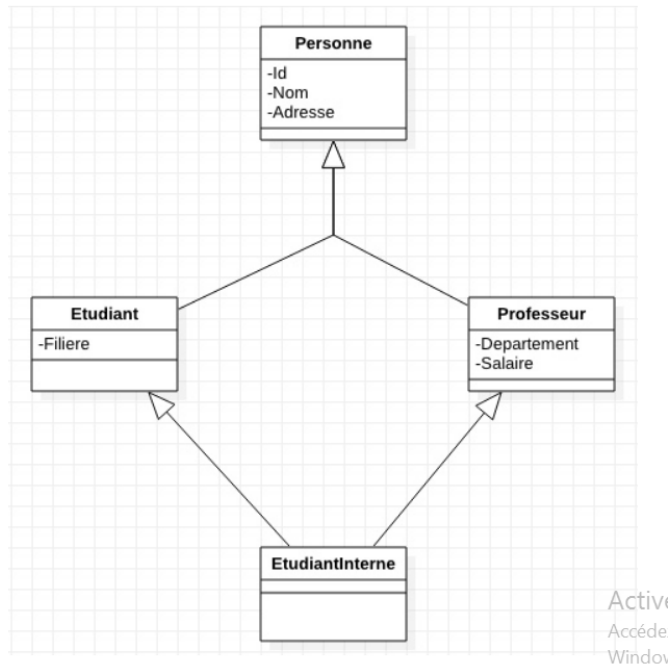
class Etudiant : public Personne
{
    private:
        char Filiere[40];

    public:
        Etudiant(int, char[], char[], char[]);
};

class EtudiantInterne : public Etudiant
{
    public:
        EtudiantInterne(int, char[], char[], char[]);
};
```

Types d'héritage

HÉRITAGE HYBRIDE



```

class Personne
{
    private:
        int Id;
        char Nom[30];
        char Adresse[100];

    public:
        Personne(int, char[], char[]);
        void afficher();
};

class Professeur : public Personne
{
    private:
        double salaire;
        char departement[40];

    public:
        Professeur(int, char[], char[], double, char[]);
};

class Etudiant : public Personne
{
    private:
        char Filiere[40];

    public:
        Etudiant(int, char[], char[], char[]);
};

class ProfAssistant : public Etudiant, public Professeur
{
    public:
        ProfAssistant(int, char[], char[], char[], double, char[]);
};
    
```

L'héritage hybride est une combinaison d'héritage hiérarchique et d'héritage multiniveau.

La portée Protected

La portée protected est un autre type de droit d'accès qu'on peut classer entre public (le plus permissif) et private (le plus restrictif).

Il n'a de sens que pour les classes qui se font hériter (les classes mères) mais on peut l'utiliser sur toutes les classes, même quand il n'y a pas d'héritage. Les éléments qui suivent protected ne sont pas accessibles depuis l'extérieur de la classe, sauf si c'est une classe fille.

```
class Point
{
protected :
int x;
int y;
Hauteur *z;
public :
~Point();
Point();
Point(int , int , int);
Point(Point const& );
void deplace (int , int , int);
void affiche () const;
Point& operator=(Point const& );
};
```

On peut alors
directement
manipuler les
coordonnées dans
tous les éléments
enfants de Point
comme PointCol

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

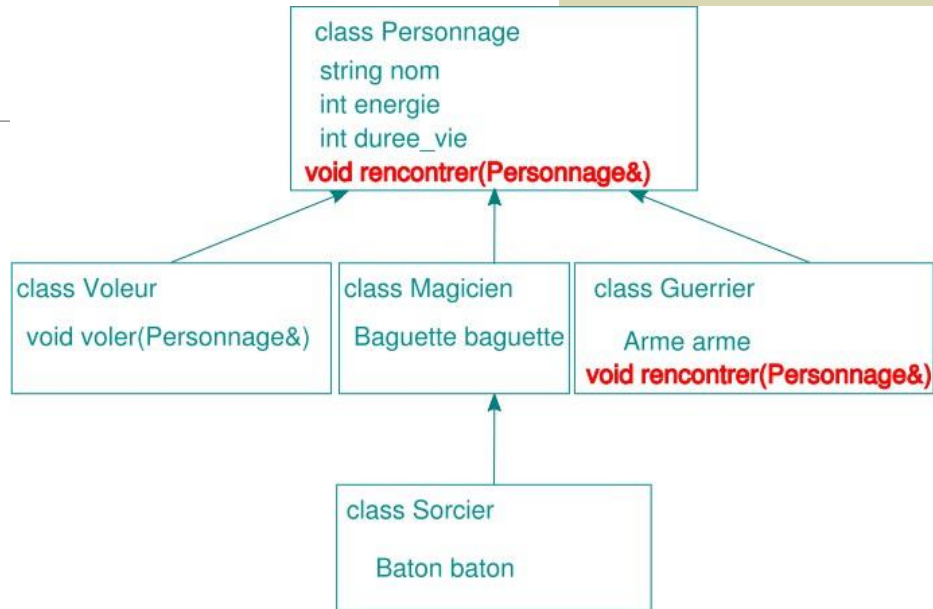
```
=== Build: Debug in heritagel (compiler: GNU GCC Compiler) ===  
In member function 'void Guerrier::frapper(Personnage&)':  
error: 'int Personnage::energie' is protected  
error: within this context  
=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

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 de méthode

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

```
Personnage
...
void rencontrer(Personnage& autre) {
    saluer(autre);
}
```

```
Guerrier : public Personnage
...
void rencontrer(Personnage& autre) {
    frapper(autre);
}
```

```
Guerrier vous frappe
Personnage vous salue

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

```
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;
}
```

- 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

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.

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

Les chaînes de caractères

Le type **char** permet de stocker un nombre qui est interprété comme un **caractère** (lettre, symbole,).

- Pour manipuler du **texte**, on peut utiliser des tableaux. Une chaîne de caractère **est un tableau de caractères**.
- Etant donné que la gestion du texte en mémoire est complexe, le langage C++ propose le **type string** pour simplifier les choses. Il permet de **créer des objets de type string** et de manipuler du texte sans avoir à nous soucier du fonctionnement de la mémoire.

Les chaînes de caractères

- Pour pouvoir utiliser des objets de type `string` dans le code, il est nécessaire d'inclure l'en-tête de la bibliothèque `string` : `#include <string>`.
- La notion de caractère de fin de chaîne n'existe plus et ce caractère peut apparaître au sein de la chaîne, éventuellement à plusieurs reprises.
- Un tel objet ressemble donc à un conteneur de type `vector<char>` et possède d'ailleurs un certain nombre de fonctionnalités communes.

Les chaînes de caractères

- L'accès aux éléments existants peut se faire avec l'opérateur `[]` ou avec la méthode `const char at(int i)` : `s.at(i);`. Le premier élément correspond à l'indice 0. La classe `string` dispose de beaucoup de constructeurs :
 - `string ch1;`
 - `string ch2(10, 'z');` string de 10 caractères égaux à `z`.
 - `string s1("ENSMR");`
 - `string s2(s1);` construction de `s2` par copie de `s1`.

Les chaînes de caractères

- `string nom ("ENIM"); nom[2] = 'S';` La variable `nom` contient alors ENSM.
- `nom = " EMI ";` la variable `nom` contient alors EMI; `nom += " ";` `nom += "Maroc";` La variable `nom` contient alors EMI Maroc. C'est ce qu'on appelle la concaténation de chaînes.
- L'opérateur `+=` est défini de façon concomitante. `string texte ("Ecole ");`
`texte += nom;` Résultat Ecole ENSMR Maroc.

Les chaînes de caractères

- Les méthodes de recherche dans une chaîne permettent de retrouver **la première ou la dernière occurrence** d'une chaîne ou d'un caractère donnés, d'un caractère appartenant à une suite de caractères donnés, d'un caractère n'appartenant pas à une suite de caractères donnés.
- Lorsqu'un tel caractère ou une telle chaîne a été **localisé**, on obtient en **retour l'indice correspondant au premier caractère concerné**. Si la recherche **n'aboutit pas**, on obtient **une valeur d'indice en dehors des limites permises pour la chaîne**, ce qui rend quelque peu difficile l'examen de sa valeur.

Les chaînes de caractères

Il existe une **classe string**, c e n'est un pas un type élémentaire. Pour l'utiliser, il faut placer tete du fichier :

```
# include <string>
```

- **string t** ; définit t comme une variable...
- **string s(25,'a')** ;
- **string mot = "bonjour"** ;
- **s.size()** représente la longueur de s
- **s[i]** est le i-ème caractère de s (i = 0,1,.. . s.size()-1)
- **s+t** est une nouvelle chaine correspondant à la concaténation de s et t.

NB: il existe une autre sorte de chaine de caractères en C/C++

La classe dispose également de différentes méthodes (présentation non exhaustive)

```
void main(void) {  
    string s1("abcdefg");  
    cout << "Longueur = " << s1.length() << endl;  
    cout << "Capacite = " << s1.capacity() << endl;  
    cout << s1 << endl;                // s1 = abcdefg  
    s1.insert(1,"ABC");                // insère la chaîne "ABC" à l'indice 1  
    cout << s1 << endl;                // s1 = aABCbcdefg  
    s1.insert(2,4,'a');                // insère 4 fois la lettre 'a' à l'indice 2  
    cout << s1 << endl;                // s1 = aAaaaaBCbcdefg  
    s1.erase(0,1);                    // supprime 1 caractère à l'indice 0  
    cout << s1 << endl;                // s1 = AaaaaBCbcdefg  
    s1.erase(2,5);                    // supprime 5 caractères à l'indice 2  
    cout << s1 << endl;                // s1 = Aabcdefg  
    // possibilité d'obtenir un pointeur compatible const char *  
    const char * ptr=s1.c_str();  
}
```