

Université Mouloud Mammeri Tizi-ouzou
Faculté d'électronique et d'informatique
Département d'électronique



Cours C++ et programmation orientée objet
La programmation orientée objet

Mr. ABAINIA

Master μ Electronique et instrumentation



Concept de POO ?



Apparu dans les années 60s au sein de MIT

Offre une grande **souplesse** de travail + **maintenance** aisée

Objet en programmation = objet dans le monde réel

Objet = propriétés (**attributs**) + actions (**méthodes**)

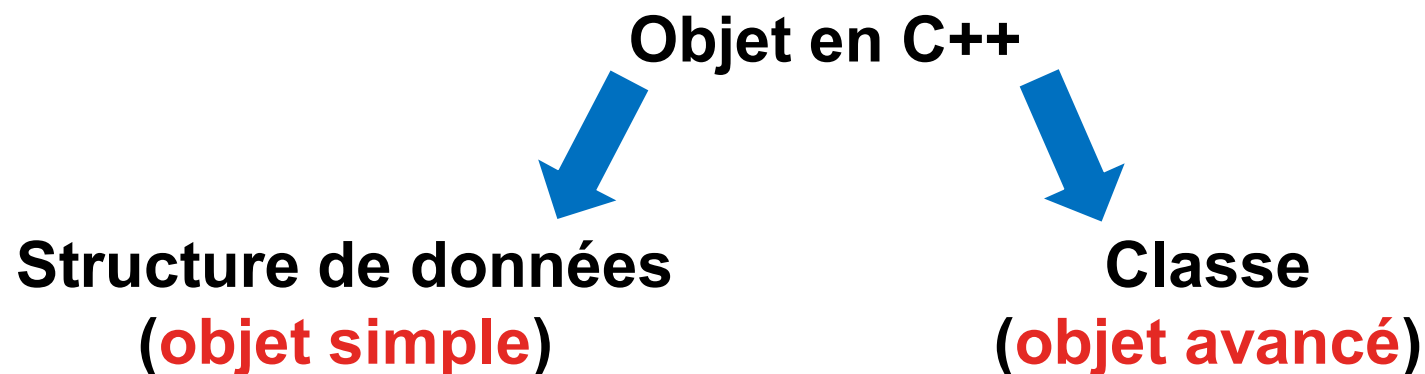




Figure tirée du blog <http://oldiesfan67.canalblog.com>

Objet = voiture

Propriétés = nb. roues
puissance
suspension
matricule
fabricant
etc.

Actions = marche arrière
marche avant
etc.



Problème réel !

On veut représenter numériquement une liste d'étudiants qui vont passer un examen.

Solution basique (sans objet) = on crée un tableau pour chaque propriété (caractéristique) des étudiants, e.g. *nom*, *prénom*, *date de naissance*, etc.

Inconvénients = on manipule les différents tableaux pour accéder aux informations d'un étudiant quelconque.

Résultats = perte de temps et manipulation complexe.



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

int main()
{
    int nb_inscription[100];
    string nom[100];
    string prenom[100];
    string date_naissance[100];
```

```
// affichage des informations du troisième étudiant
```

```
cout<<" Nom : "<< nom[2] <<endl;
cout<<" Prénom : "<< prenom[2] <<endl;
cout<<" Date de naissance : "<< date_naissance[2] <<endl;
cout<<" Nombre d'inscription : "<< nb_inscription[2] <<endl;
```

```
return 0;
```

```
}
```

**On risque de se
tremper sur les
indices des tableaux**



Solution avec les objets = on crée un objet regroupant toutes les propriétés des étudiants, puis un tableau d'objets.

Avantages = on manipule un seul tableau dont la liste des étudiants. Les informations sont hiérarchiquement structurées.

Résultats = gain de temps et programmation facile.



Structure de données

(objet simple)



- Un concept utilisé pour **créer** des **types composés** dont plusieurs variables de **différents types** (attributs).

- Il sert également pour **décomposer** un **type en bits**.

- Une **structure** peut **contenir** des **fonctions** pour manipuler les attributs et effectuer différentes opérations.

Pour créer une structure de données, on utilise le mot clé **struct** en suivant la syntaxe suivante:

Syntaxe:

```
struct nom_type  
{  
    type variable;  
    type variable;  
    type variable;  
    ...  
};
```

Exemple:

```
struct Etudiant  
{  
    string nom;  
    string prenom;  
    string date_naissance;  
    int nombre_inscription;  
};
```



```
// Utilisation de l'objet créé
```

```
Etudiant etud;
```

```
etud.nom = "nom quelconque";
```

```
etud.prenom = "quelconque";
```

```
etud.date_naissance = "12/10/2000";
```

```
etud.nombre_inscription = 7;
```

```
// tableau dynamique de type Etudiant
```

```
Etudiant* liste_etudiant;
```

```
liste_etudiant = new Etudiant[100];
```

```
// tableau statique de type Etudiant
```

```
Etudiant liste_etudiant_2[100];
```

```
// définition de l'objet
```

```
struct Etudiant
```

```
{
```

```
    string nom;
```

```
    string prenom;
```

```
    string date_naissance;
```

```
    int nombre_inscription;
```

```
};
```



On peut **déclarer les noms des objets (variables)** à utiliser au moment de la **définition** de l'objet lui-même.

Syntaxe:

```
struct nom_type
{
    type variable;
    type variable;
    type variable;
    ...
} nom1, nom2, nom3 ;
```

Exemple:

```
struct Personne
{
    string nom;
    string prenom;
    string date_naissance;
    int numero_social;
} etudiant, enseignant, administrateur,
agent_securite, *liste_etudiant ;

enseignant.nom = "nom quelconque";
enseignant.prenom = "prénom";
etudiant.prenom = "prénom";
```

Définition d'un nouvel objet:

```
struct  Personne
{
    string nom;
    string prenom;
    string date_naissance;
    int numero_social;

    void affichage()
    {
        // afficher des informations
    }

    void changerNumeroSocial(int num)
    {
        numero_social = num;
    }
};
```



Utilisation de l'objet créé:

// déclaration variable statique

Personne etudiant;

//déclaration variable dynamique

Personne* enseignant;

// accéder aux éléments avec point

etudiant.nom = "nom quelconque";

// allouer l'objet créé

enseignant = new Personne;

// accéder aux élément avec ->

enseignant->nom = "nom quelconque";



Reprenons l'exercice N° 5 de la série N° 1, et on propose une solution plus facile.

// décomposition d'un type en bits

```
struct Protocole
{
    unsigned char act_1 : 3;
    unsigned char act_2 : 3;
    unsigned char act_3 : 2;
};
```

On met deux points et on précise le nombre de bits

Syntaxe:
memcpy(ptr dest , ptr src , taille)

// dans la fonction main

```
int temp; // va contenir une valeur codée sur 8 bits (1 octet)
```

```
cin>>temp; // on tape une valeur entre 0-255
```

```
Protocole proto;
```

```
memcpy(&proto , &temp , 1); // fonction spéciale pour copier un contenu
```

```
cout<< proto.act_1 <<endl;
```

```
cout<< proto.act_2 <<endl;
```

```
cout<< proto.act_3 <<endl;
```



Programmation orientée objet

(Les classes ou objets avancés)



Problème

- ✓ On reprend l'exemple précédent de l'étudiant, l'enseignant, l'administrateur et l'agent de sécurité.
- ✓ **Toutes** les catégories précédentes des personnes **partagent quelques propriétés communes** (e.g. nom, prénom, date de naissance, etc.).
- ✓ **Chaque** catégorie a quelques **propriétés uniques**.

Comment peut-on représenter toutes ces catégories ?



Solution archaïque

La solution utilisant les **structures** consiste à créer un **objet** (structure) **indépendant** pour chaque catégorie.

- trop de redondances
- occupation d'espace mémoire important
- maintenance couteuse qui pourra causer des problèmes



// structure pour les étudiants

```
struct Etudiant
{
    string nom;
    string prenom;
    string date_naissance;
    int numero_social;
    ...
};
```

// structure pour les enseignants

```
struct Enseignant
{
    string nom;
    string prenom;
    string date_naissance;
    int numero_social;
    int grade;
    string diplome;
    ...
};
```

Redondance

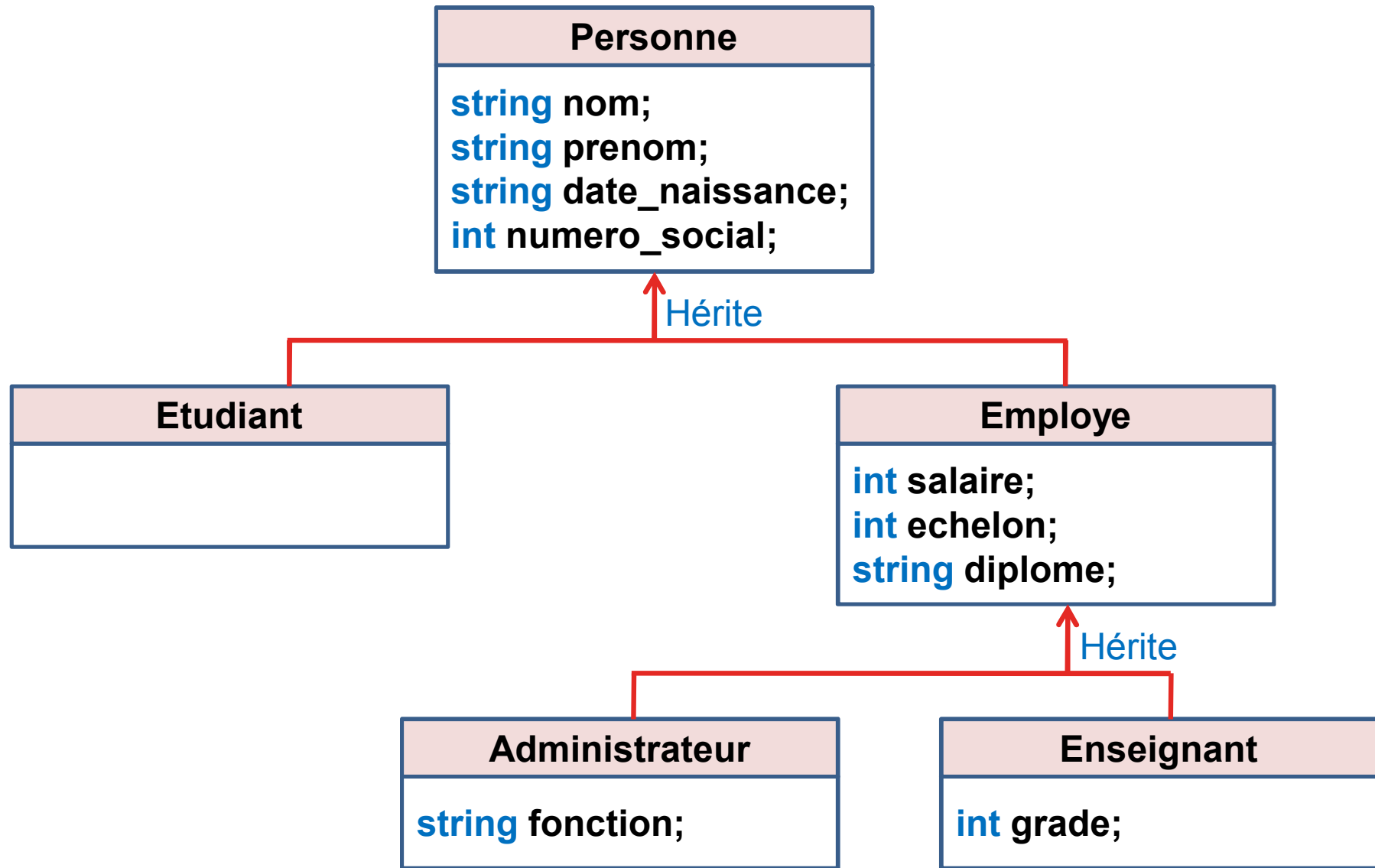


Solution idéale

L'utilisation des **objets plus évolués (classes)** qui permettent de bien **structurer** le code et **faciliter** la conception.

Classe **vs** structure:

- ✓ Encapsulation
- ✓ Héritage
- ✓ Polymorphisme
- ✓ Interface





Pour créer une classe, on utilise le mot clé **class** en suivant la syntaxe suivante:

Syntaxe:

```
class nom_classe
{
    type variable;
    type variable;

    type fonction( ) { }
    type fonction( ) { }
    ...
};
```

Exemple:

```
class Personne
{
    string nom;
    string prenom;
    string date_naissance;

    void marcher( )
    { // écrire un code
    }

    int dormir( )
    { // écrire un code
    }
};
```



En **C++**, il y a **trois** manières de créer le contenu (**implémenter**) des classes:

- **Déclarer et définir** les membres à **l'intérieur**.
- **Déclarer** les membres à **l'intérieur** et les **définir** à **l'extérieur**.
- **Déclarer** les membres à **l'intérieur** et les **définir** dans un **autre fichier**.

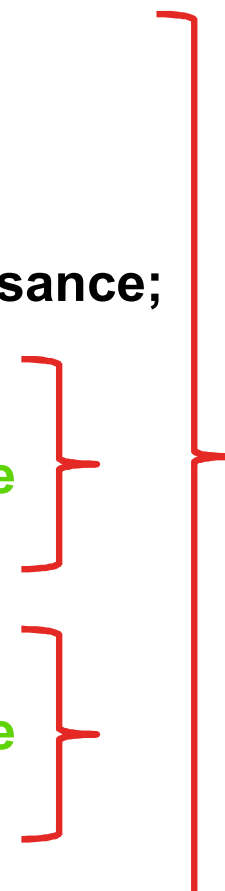
Membre = attribut (variable) ou méthode (fonction)

1) Déclarer et définir les membres à l'intérieur ?

```
class Personne
{
    string nom;
    string prenom;
    string date_naissance;

    void marcher( )
    { // écrire un code
    }

    int dormir( )
    { // écrire un code
    }
};
```



Fortement déconseillé



2) Déclarer les membres à l'intérieur et les définir à l'extérieur?

```
class Personne
{
    string nom;
    string prenom;
    string date_naissance;
    void marcher( );
    int dormir( );
};
```

Ce n'est pas professionnel !

```
void Personne :: marcher( )
{ // écrire un code
}
```

On écrit seulement l'entête des fonctions avec les paramètres s'il y en a

```
int Personne :: dormir( )
{ // écrire un code
}
```

On définit le contenu des méthodes à l'extérieur de la classe



Pour **définir le contenu** des méthodes (fonctions) en **dehors** de la classe on met le **nom de la classe suffixé** par **::** entre le **type** et le **nom de la méthode**.

Nom de la classe

Nom de la méthode

Type de la méthode

```
void Personne :: marcher( )  
{// écrire un code  
}
```

Quatre points
::



3) Déclarer les membres à l'intérieur et les définir dans un autre fichier?

```
class Personne
{
    string nom;
    string prenom;
    string date_naissance;

    void marcher( );
    int dormir( );
};
```

fichier.h

```
#include "fichier.h"

void Personne :: marcher( )
{ // écrire un code
}

int Personne :: dormir( )
{ // écrire un code
}
```

fichier.cpp

Méthode conseillée et professionnelle



- ❑ On fait la même chose que la deuxième manière, mais en **implémentant** les fonctions dans un **fichier indépendant**.

- ❑ On appelle le fichier contenant la déclaration «**un fichier entête** ou **fichier d'interface**» et doit porter **l'extension .h** (abréviation de **header**).

- ❑ **L'implémentation** des méthodes **doit se faire** dans un **fichier source** ayant une **extension .cpp** et de préférence le même **nom** du **header**.

- ❑ Dans le fichier source (**implémentation**), on **inclut** le **fichier header** avec la directive **#include**.



**Cette méthode de programmation
est appelée **D.D.U**
(abréviation de **Déclaration-Définition-Utilisation**)**

***Avec cette méthode on crée des bibliothèques
personnelles.***

***L'utilisateur peut consulter le fichier d'interface
(header) pour savoir le fonctionnement de la classe
(un manuel).***



```
// fichier RobotMobile.h
```

```
class RobotMobile
{
    float position_x;
    float position_y;

    void avancer(float x , float y);
};
```

```
// fichier RobotMobile.cpp
```

```
#include "RobotMobile.h"
```

```
void RobotMobile ::avancer(float x , float y)
{
    position_x += x;
    position_y += y;
}
```

Exercice: on veut créer une classe simple pour représenter un robot mobile.



Déclaration d'une variable de type classe est appelée instance de la classe

```
// fichier main.cpp
```

```
#include "RobotMobile.h"
```

```
int main() {  
    RobotMobile robot;
```

```
    // initialisation
```

```
    robot.position_x = 10.0;  
    robot.position_y = 0.3;
```

```
    // avancer dans l'espace  
    robot.avancer(1.0 , 1.0);
```

```
    return 0;
```

```
}
```

```
// fichier main.cpp
```

```
#include "RobotMobile.h"
```

```
int main() {  
    RobotMobile *robot;  
    robot = new RobotMobile ;
```

```
    // initialisation
```

```
    robot->position_x = 10.0;  
    robot->position_y = 0.3;
```

```
    // avancer dans l'espace  
    robot->avancer(1.0 , 1.0);
```

```
    if(robot != NULL) {  
        delete robot;  
        robot = NULL;
```

```
    }  
    return 0;
```

```
}
```



Attention

```
// fichier engen.h
```

```
class Engin  
{  
};
```

```
// fichier vehicule.h
```

```
#include "engin.h"
```

```
class Vehicule : Engin  
{  
};
```

```
// fichier bateau.h
```

```
#include "engin.h"
```

```
class Bateau : Engin  
{  
};
```

Erreur:
Redéfinition de la
classe Engin

```
// fichier main.cpp
```

```
#include "vehicule.h"
```

```
#include "bateau.h"
```

```
int main()
```

```
{  
}
```




Solution

```
// fichier engen.h  
#ifndef ENGIN_H  
#define ENGIN_H  
// code source  
#endif
```

Ajouter des
préprocesseurs de test

```
// fichier vehicule.h  
#ifndef VEHICULE_H  
#define VEHICULE_H  
// code source  
#endif
```

```
// fichier bateau.h  
#ifndef BATEAU_H  
#define BATEAU_H  
// code source  
#endif
```

```
// fichier main.cpp  
#include "vehicule.h"  
#include 'bateau.h'  
int main()  
{  
}
```



Constructeur et Destructeur



❑ Les **constructeurs** et **destructeurs** sont des **méthodes** (fonctions) spécifiques **sans types de retour**, mais ils **peuvent avoir des paramètres**.

❑ Le **constructeur** est **appelé automatiquement** lors de la **création de l'objet statique ou dynamique**.

❑ Le **destructeur** est **appelé automatiquement** lors de **suppression de l'objet créé (dynamique)** ou à **la fin du programme** si **l'objet est statique**.

Le **constructeur** porte le **même nom** de la **classe** en respectant le majuscule et le minuscule.

// fichier RobotMobile.h

class RobotMobile

{

float position_x;

float position_y;

void avancer(**float** x , **float** y);

RobotMobile();

~RobotMobile();

};

Le **destructeur** porte le **même** nom de la **classe** en respectant le majuscule et le minuscule et **doit être préfixé** du caractère **~**.



❖ Le **constructeur** sert pour:

✓ **initialiser** les **attributs** (variables) de la **classe**.

✓ **allouer** l'**espace mémoire** des variables **dynamique**.

✓ **appeler** des **fonctions** pour faire un **prétraitement** dans les **programmes avancés**.

❑ Le **constructeur** est appelé **lors** de la **déclaration** d'une **instance statique**.

❑ Il est appelé lors de **l'utilisation** de l'opérateur **new** pour allouer l'espace d'une **instance dynamique**.

❖ Le **destructeur** sert pour **libérer** l'**espace mémoire** alloué par les variables **dynamiques**.

// fichier main.cpp

#include "RobotMobile.h"

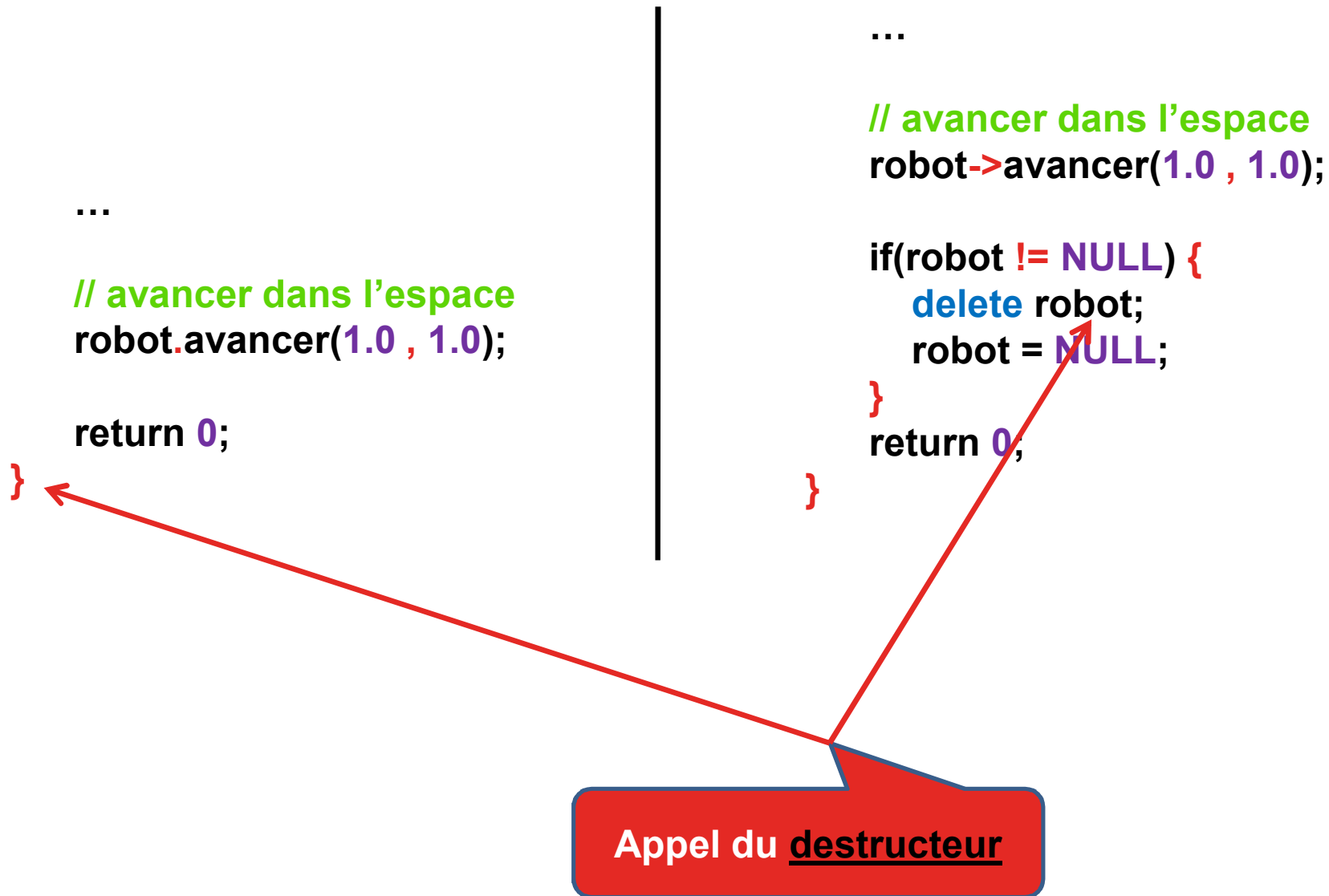
```
int main() {
    RobotMobile robot;
    ...
```

// fichier main.cpp

#include "RobotMobile.h"

```
int main() {
    RobotMobile *robot;
    robot = new RobotMobile ;
    ...
```

Appel du constructeur



// fichier RobotMobile.h

```
class RobotMobile
{
    float position_x;
    float position_y;

    void avancer(float x , float y);

    RobotMobile(float x , float y);
    ~RobotMobile();
};
```

// fichier RobotMobile.cpp

```
#include "RobotMobile.h"
```

```
RobotMobile :: RobotMobile(float x , float y)
{
    position_x = x;
    position_y = y;
}
```

```
RobotMobile :: ~RobotMobile()
{
    // on n'a rien à libérer
}
```

```
void RobotMobile ::avancer(float x , float y)
{
    position_x += x;
    position_y += y;
}
```


Appel du constructeur pour initialiser la position

```
// fichier main.cpp
#include "RobotMobile.h"

int main() {
    RobotMobile robot(10.0 , 0.3);

    // avancer dans l'espace
    robot.avancer(1.0 , 1.0);

    return 0;
}
```

Appel du destructeur

```
// fichier main.cpp
#include "RobotMobile.h"

int main() {
    RobotMobile *robot;
    robot = new RobotMobile(10.0 , 0.3);

    // avancer dans l'espace
    robot->avancer(1.0 , 1.0);

    if(robot != NULL) {
        delete robot;
        robot = NULL;
    }
    return 0;
}
```



Encapsulation



L'encapsulation sert pour:

- **donner des droits d'accès** aux membres des classes.
- **protéger les attributs** contre la **modification** de l'extérieur.

Il existe **trois modificateurs** d'accès:

- **public** = membres accessibles **même** par **d'autres objets**
- **private** = membres accessibles par la **classe seulement**
- **protected** = membres accessibles par la **classe et ses enfants**

Tous les membres sont par défaut en mode private.



Scénario (1)

On utilise quotidiennement un instrument électronique comme la télévision (objet) sans savoir sa constitution.

Le **public effectue **seulement** des actions de base tel que:**

- **Allumer la télé**
- **Eteindre la télé**
- **Changer les chaînes**
- **Régler les paramètres software**
- **Brancher des périphériques externes**



Scénario (2)

La télé est dotée de quelques caractéristiques internes:

- Voltage de fonctionnement
- Puissance consommée
- Résistance des boutons
- Microcontrôleur
- Enceinte du son
- Etc.

L'utilisateur normal **ne peut pas toucher** ces caractéristiques, car elles **sont protégées** par le fabricant.

C'est dans cet aspect que l'encapsulation dans POO a été conçue.



Pour désigner le **mode d'accès** à certains membres (attributs ou méthodes), on met le **modificateur suffixé par deux points (:)**.

Remarque: *tous les membres qui succèdent le modificateur suivront le même mode d'accès. Donc, il faut faire attention aux autres membres.*

// exemple

```
class RobotMobile
```

```
{
```

```
private:
```

```
float position_x;
```

```
float position_y;
```

```
public:
```

```
void avancer(float x , float y);
```

```
RobotMobile(float x , float y);
```

```
~RobotMobile();
```

```
};
```

Accessible seulement
par les méthodes de la
classe

Accessible par n'importe
quelle méthode ou fonction



Attention ! Erreur !

```
// fichier main.cpp
```

```
#include "RobotMobile.h"
```

```
int main() {  
    RobotMobile robot;  
  
    // initialisation  
    robot.position_x = 10.0;  
    robot.position_y = 0.3;  
  
    // avancer dans l'espace  
    robot.avancer(1.0 , 1.0);  
  
    return 0;  
}
```

On ne peut pas accéder à un membre privé (private) en dehors de la classe.

Attention ! Erreur !

// fichier RobotMobile.h

```
class RobotMobile
{
    private:
        float position_x;
        float position_y;

        void avancer(float x , float y);

        RobotMobile(float x , float y);
        ~RobotMobile();
};
```

Oublier de mettre le modificateur
pour les méthodes publiques



Résultat = tous les attributs et toutes les méthodes seront considérés
comme **private**.



Getter et Setter

Vu qu'on **ne peut pas accéder** directement aux membres **private** et **protected**, il y a toujours un moyen de le faire.

- On utilise les **Getters (méthodes)** pour **lire** les valeurs
- On utilise les **Setters (méthodes)** pour **modifier** les valeurs

// fichier RobotMobile.h

private:

```
float position_x;  
float position_y;
```

public:

```
void setPosX(float x);  
float getPosX();
```

// fichier RobotMobile.cpp

public:

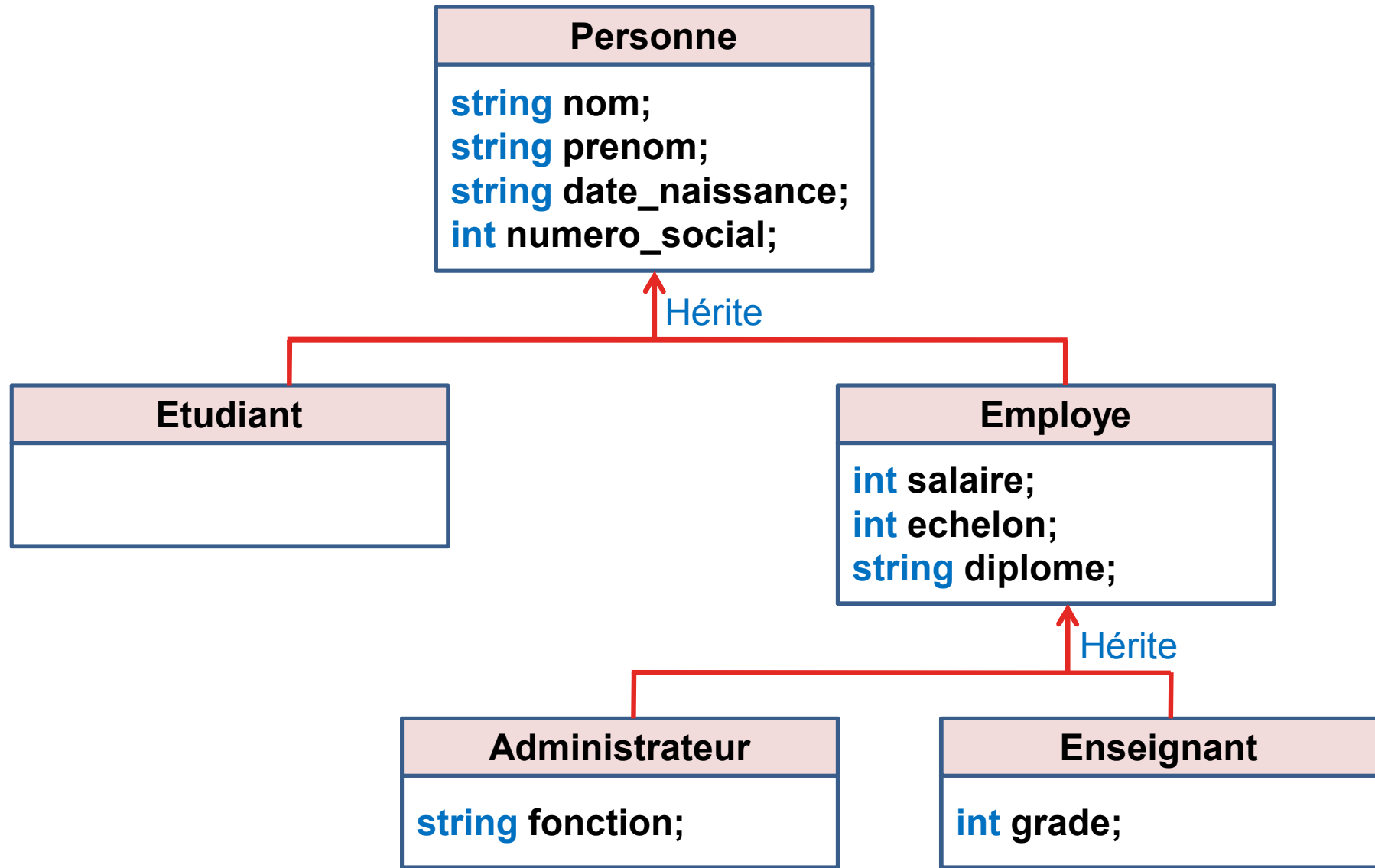
```
void RobotMobile :: setPosX(float x)  
{  
    position_x = x;  
}  
float RobotMobile :: getPosX()  
{  
    return position_x;  
}
```



Héritage en POO

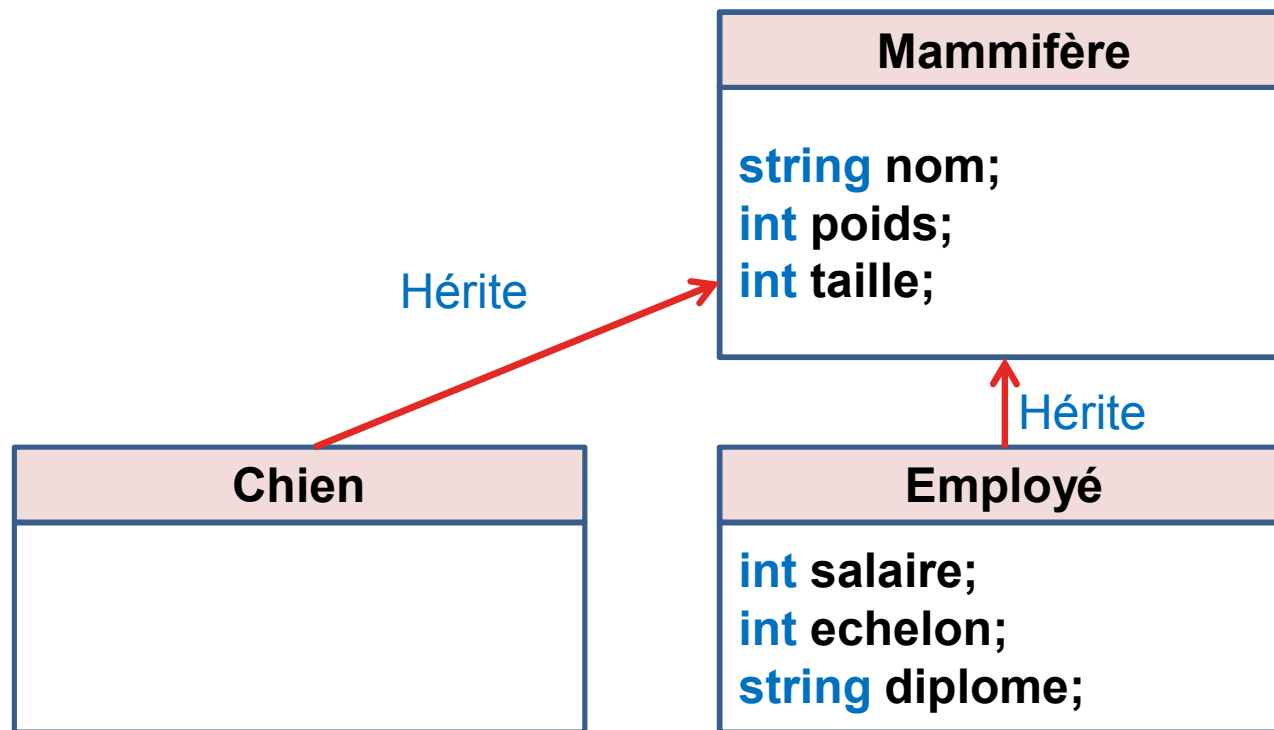


- Le concept d'héritage consiste à **éliminer la redondance** dans l'abstraction des données.
- L'héritage consiste à **unifier** des variables et fonctions **partagés** entre plusieurs objets dans un seul objet.
- Les objets enfants héritent **tous** ou **quelques** membres (attributs et méthodes) de l'objet parent.
- L'objet **parent** et **enfant** doivent **garder la sémantique**.



Erreur sémantique !

Employé ne doit pas hériter de Mammifère quelque soit, car ils ne sont pas sémantiquement liés.





Héritage en C++ se fait en ajoutant deux points (:) après le nom de classe enfante + le nom de la classe parente.

Exemple:

// classe parente

```
class RobotMobile
{
    protected:
        float position_x;
        float position_y;

    public:
        void avancer(float x , float y);

        RobotMobile(float x , float y);
        ~RobotMobile();
};
```

// classe enfante héritée de RobotMobile

```
class RobotBiped : RobotMobile
{
    protected:
        Moteur moteur_1, moteur_2;
        Moteur moteur_3, moteur_4;

    public:
        RobotBiped();
        ~ RobotBiped();
};
```



On peut **spécifier** le **mode** d'héritage soit **public** ou **private**.

❖ Héritage **public** = membres hérités **conservent** les **mêmes** droits d'accès du parent.

❖ Héritage **private** = membres hérités **deviennent** **privés** dans la classe dérivée.

Si on **ne spécifie pas** le **mode** d'héritage, l'héritage **privé** est effectué **par défaut**.



On reprend l'exemple précédent, où la classe dérivée devient (*le code ci-dessous est illustratif*):

```
// class RobotBiped : private RobotMobile
class RobotBiped : RobotMobile
{
    protected:
        Moteur moteur_1, moteur_2;
        Moteur moteur_3, moteur_4;
    private:
        float position_x;
        float position_y;

    public:
        RobotBiped();
        ~ RobotBiped();

    private:
        void avancer(float x , float y);
        RobotMobile(float x , float y);
        ~RobotMobile();
};
```

```
class RobotBiped : public RobotMobile
{
    protected:
        Moteur moteur_1, moteur_2;
        Moteur moteur_3, moteur_4;
    protected :
        float position_x;
        float position_y;

    public:
        RobotBiped();
        ~ RobotBiped();

    public:
        void avancer(float x , float y);
        RobotMobile(float x , float y);
        ~RobotMobile();
};
```



Il est **conseillé** de faire un **héritage public** en cas d'un autre héritage de la classe enfante, car la classe enfante de la classe enfante **doit hériter les propriétés** de classe parente (classe de base).

Les constructeurs et destructeurs de tous les parents d'une classe enfante sont appelés avant d'exécuter ceux de la classe enfante.

```
class A {
    public:
    A() {
        cout<<"classe A"<<endl;
    }
};

class B : public A {
    public:
    B() {
        cout<<"classe B"<<endl;
    }
};

class C : public B {
    public:
    C() {
        cout<<"classe C"<<endl;
    }
};
```

```
int main( )
{
    cout<<"affichage..."<<endl;
    C enfant; // constructeur appelé

    return 0;
}
```

```
Affichage...
classe A
classe B
classe C
```



C++ permet de faire un héritage multiple contrairement aux autres langages de programmation

Après les deux points (:), on met les noms des classes (avec les modificateurs) parentes séparés par des virgules

```
class Enfant : public Parent1 , public Parent2 , public Parent3
```

Attention !

Lorsque la **même méthode existe** dans **différentes classes parentes** et est appelée dans la classe enfant, le compilateur détecte une ambiguïté.

```
class A {  
    public:  
    void fonct() {}  
};
```

```
class B {  
    public:  
    void fonct() {}  
};
```

```
class C : public A, public B {  
    public:  
    C () {  
        fonct();  
    }  
};
```

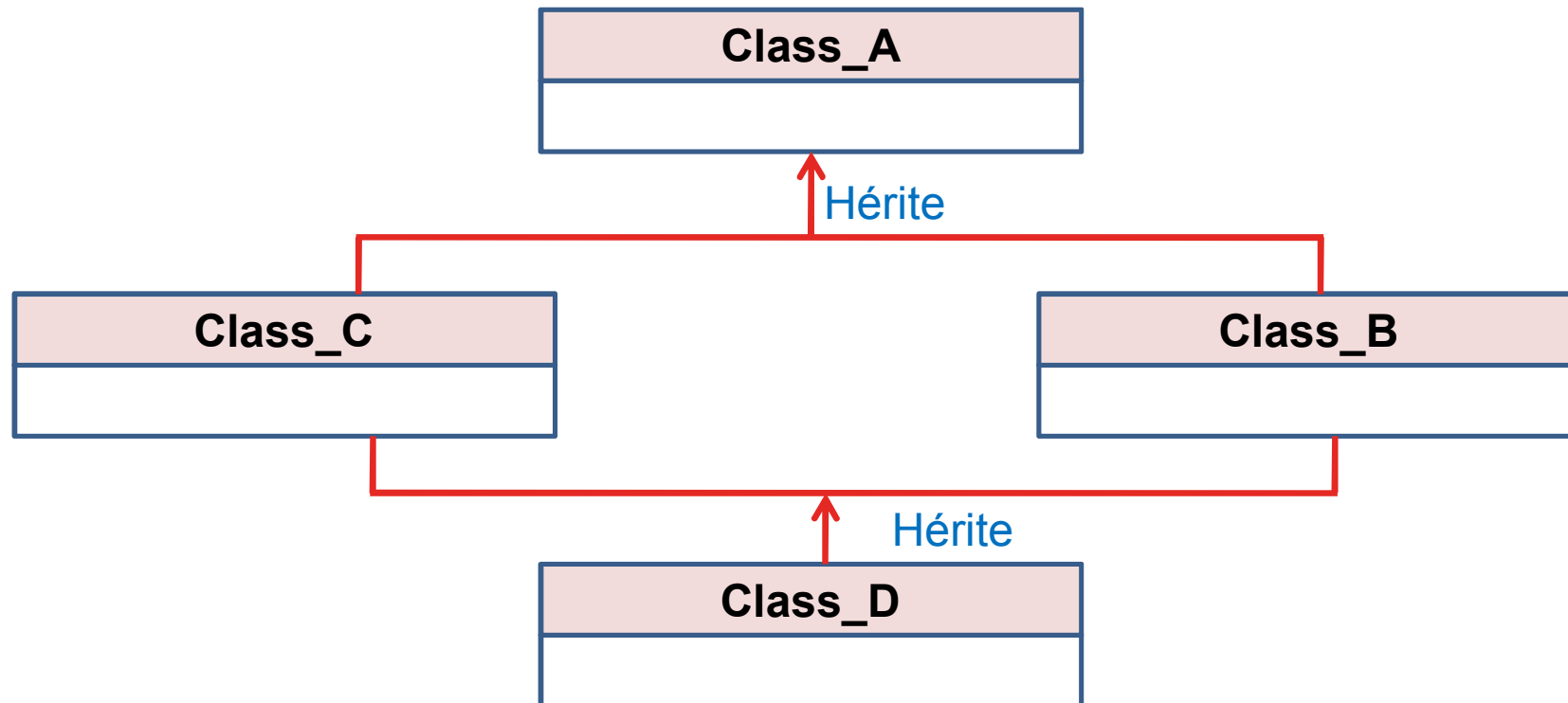
Le compilateur ne sait pas de quelle fonction s'agit-elle.



Remarque: on ne peut pas hériter de la même classe deux fois.

```
class Enfant : public Parent1 , public Parent1
```

Par contre, il y a l'héritage en diamant !





Polymorphisme



Le concept de **polymorphisme** est dédié pour les méthodes des classes (fonctions), où elles peuvent prendre différentes formes dans la même classe.

Il y a **quatre types** de polymorphisme:

- ✓ Polymorphisme **ad-hoc**
 - ❖ Surcharge
 - ❖ Coercition

- ✓ Polymorphisme **universel**
 - ❖ Paramétrique
 - ❖ D'inclusion



❖ Le polymorphisme de **surcharge** consiste à définir plusieurs méthodes portant le **même nom**, mais avec différents types de retour et **différents paramètres** (nombres et types).

❖ Le **nombre** de paramètres et les **types** des paramètres représentent la **signature** des méthodes (distinction).

// class RobotMobile implémentant différentes fonctions 'avancer'

```
class RobotMobile
{
    protected:
        float position_x;
        float position_y;

    public:
        void avancer(float x , float y);
        bool avancer(int x , int y);
        bool avancer(ObjetPosition x , ObjetPosition y);
        RobotMobile(float x , float y);
        ~RobotMobile();
};
```



Le polymorphisme de **coercition** consiste à surcharger des opérateurs pour permettre la conversion implicite du type (convertir la classe en autre type).

// surcharge du type int pour pouvoir convertir la classe en int

```
class MaClasse
```

```
{
```

```
    public:
```

```
        operator int ()
```

```
        {
```

```
            return 2; // on peut effectuer une opération arithmétique  
                    // ou afficher le contenu d'une variable
```

```
        }
```

```
};
```

On parlera
prochainement sur
cette syntaxe

// dans la fonction main

```
int entier;
```

```
MaClasse ma_classe;
```

```
entier = ma_classe; // si la classe n'est surchargée de int, ça renvoie une erreur
```

```
cout<< entier << endl; // le résultat affiché est 2
```



Le polymorphisme **paramétrique** consiste de rendre les méthodes plus **génériques** et adaptées à **plusieurs paramètres** au lieu de **définir les différentes formes**. En particulier, on bénéficie du **paradigme de template** qui est propre au c++ (**une technique avancée**).

// généraliser les méthodes d'une classe avec les templates

```
class MaClasse
```

```
{
```

```
    public:
```

```
        template <class T> void fonction (T variable)
```

```
        {
```

```
        }
```

```
        template <class T1 , class T2> int fonction (T1 variable , T2 variable2)
```

```
        {
```

```
        }
```

```
};
```

// dans la fonction main

```
double var;
```

```
MaClasse ma_classe;
```

```
ma_classe.fonction <double> (var);
```

On parlera
ultérieurement sur
cette technique



❖ Le polymorphisme **d'inclusion** constitue **l'abstraction** des classes. En d'autres termes, il consiste de déclarer les **prototypes** dans les classes de base et les **redéfinir** dans les classes enfants.

❖ On met le mot **virtual** avant le type de la méthode pour rendre cette dernière **abstraite**.

```
class MaClasse
{
    public:
        virtual void fonction () { }
};
```

```
class classeEnfante : public MaClasse
{
    public:
        void fonction ()
        {
            // implémenter la fonction différemment
        }
};
```



Classes abstraites et Interfaces



Une **classe abstraite** contient au moins une **méthode abstraite**

Une **méthode abstraite (purement)** est une méthode **non implémentée** dans la **classe parente** et **implémentée et redéfinie** différemment par les **sous-classes**.

But



Traiter des classes liées de manière **uniforme** sans prendre en considération leurs détails

Imposer une **spécification** d'implémentation des sous-classes



- ❖ Une méthode purement abstraite est précédée par le mot clé *virtual* dans la classe parente et on lui **affecte 0**
- ❖ La **redéfinition** est **précédée** par le mot clé *override* ou *final* dans les sous-classes et a la même signature (c++11)

```
class Engin
{
    float m_speed , m_weight;
    Engin();
    ~Engin();
    virtual void move() = 0;
    virtual void stop() = 0;
};

class Vehicule
{
    int m_nb_wheels;
    override void move() { //code }
    override void stop() { //code }
};
```

Classe ne peut pas être instanciée

Méthode ne peut pas être implémentée ici

En utilisant **final**, ça veut dire que la méthode ne peut plus être redéfinie par les sous-classes



Une **interface** est un cas particulier des **classes abstraites** et caractérisée par:

- Pas de variables et constructeurs
- Toutes les méthodes sont **abstraites**

```
class Engin
{
public:
    virtual void move() = 0;
    virtual void stop() = 0;
    virtual void open() = 0;
    virtual void close() = 0;
};
```

```
class Boat : Engin
{
protected:
    int m_nb_motors;

public:
    override void move() {}
    override void stop() {}
    override void open() {}
    override void close() {}
};
```

```
class Vehicule : Engin
{
protected:
    int m_posX, m_posY;
    bool m_state;

public:
    override void move() {}
    override void stop() {}
    override void open() {}
    override void close() {}
};
```




Traitement uniforme

```
class Shape
{
Protected:
    int m_posX, m_posY;
public:
    virtual void draw() = 0;
};
```

```
class Circle : Shape
{
protected:
    int m_radius;
public:
    override void draw()
    {
        // afficher cercle
    }
};
```

```
class Rect : Shape
{
protected:
    int m_posX;
public:
    override void draw()
    {
        // afficher rectangle
    }
};
```

```
int main()
{
    Shape** tab = new Shape* [2];
    tab[0] = new Circle;
    tab[0].draw();
    tab[1] = new Rect;
    tab[1].draw();
};
```

Tableau
de Shape*





Fonctions amies (*friend*)



- ❖ Une **fonction amie** d'une classe est une fonction **indépendante**, où elle pourra être **définie** dans une autre classe ou en dehors.
- ❖ Une **fonction amie** peut **accéder** à **tous les membres** (attributs ou méthodes) d'une classe **quelque soit** le mode d'accès (public, private ou protected).
- ❖ Pour désigner une **fonction amie**, on met le mot **friend** succéder par la **déclaration du prototype** de la fonction (sans implémentation).

```
class MaClasse
{
    public:
        friend void fonction (/* paramètres */);
};
```

Il faut informer le compilateur qu'il y a une classe A

```
class Class_A;
```

```
class Class_B
{
    public:
        friend void Class_A::fonction (/* paramètres */);
};

class Class_A
{
    public:
        void fonction (/* paramètres */)
        {
            // implémentation
        }
};
```

Renvoie une erreur, car la classe A non déclarée à l'avance



```
class Class_A;
```

```
class Class_B
```

```
{
```

```
public:
```

```
friend class Class_A;
```

```
};
```

```
class Class_A
```

```
{
```

```
public:
```

```
void fonction (/* paramètres */)
```

```
{
```

```
    // implémentation
```

```
}
```

```
void fonction_2 (/* paramètres */)
```

```
{
```

```
    // implémentation
```

```
}
```

```
};
```

Toutes les méthodes de la classe A
sont amies de la classe B

Si la **classe A** est **amie** de la **classe B**, ça ne veut pas dire que la **classe B** est **amie** de la **classe A**.

Si la **classe A** est **amie** de la **classe B** et la **classe B** est **amie** de la **classe C**, ça ne veut pas dire que la **classe B** est **amie** de la **classe A**.

L'amitié n'est pas symétrique.



Utilité des fonctions amies ?

Lorsqu'on veut manipuler deux ou plus objets à la fois.



Surcharge des opérateurs



✓ Technique permettant de réaliser des **opérations mathématiques** intelligentes sur les classes

✓ Lisibilité = code **court + clair**

Personne A, B, C; // **Personne est une classe définie**

A.poids = 70;

A.taille = 180;

B.poids = 90;

B.taille = 190;

// **simple méthode pour calculer la moyenne**

C.poids = (A.poids + B.poids) / 2;

C.taille = (A.taille + B.taille) / 2;

// **ou on définit une fonction pour faire la moyenne**

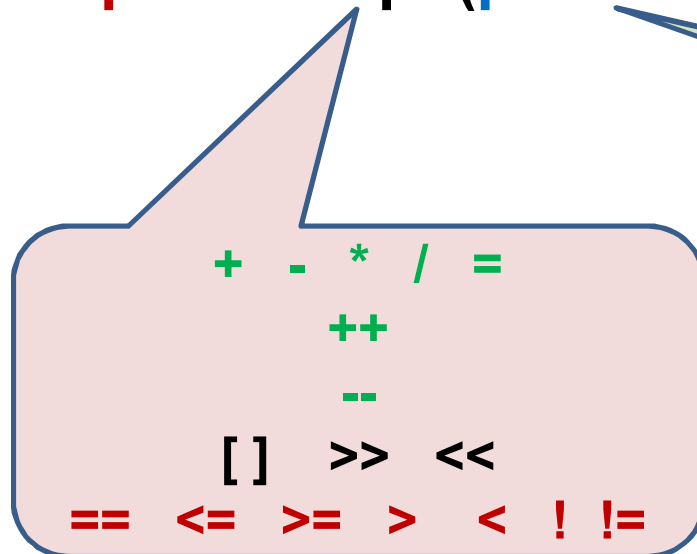
C = moyenne (A, B);

Solution optimale

$C = (A + B) / 2;$

Syntaxe:

type **operator** **opt** (**params**)



Un ou **plusieurs**
paramètres de
différents types

- Opérateur **logique** = type de renvoi **booléen**
- Opérateur **arithmétique** = type de renvoi **numérique/objet**
- Opérateur **index** = type de renvoi **numérique/objet**
- Opérateur **streaming** = type de renvoi **streaming**



Exemple:

```
Personne operator + (Personne const & A, Personne const & B)
{
    Personne C;
    C.poids = A.poids + B.poids;
    C.taille = A.taille + B.taille;
    return C;
}
```

Paramètre en
référence pour le
protéger contre
modification

```
Personne operator / (Personne const & A, int const & val)
{
    Personne C;
    C.poids = A.poids / val;
    C.taille = A.taille / val;
    return C;
}
```

Exemple:

```
bool operator > (Personne const & A, Personne const & B)
{
    if(A.poids > B.poids && A.taille > B.taille) return true;
    return false;
}
```

```
ostream operator << (Personne const & A)
{
    cout<<"poids = "<<A.poids<<" taille = "<<A.taille;
}
```



Classes imbriquées

Déclaration des sous-classes à l'intérieur des classes

```
class Class_A
{
    class Sous_class_A
    {
        // code source
    };

private:
    Sous_class_A variable;

public:
    void method (/* paramètres */)
    {
        // implémentation
    }
};
```

Classe à l'intérieur d'une autre

Instanciation de la nouvelle sous-classe



Prochain cours

Conteneurs, itérateurs et foncteurs