

<b>TD</b>	Initiation au Langage C++
<b>Auditoire</b>	3 <sup>ème</sup> année ingénieur en Informatique Industrielle et Automatique (IIA3)
<b>Etablissement</b>	Institut National des Sciences Appliquées et de Technologies
<b>Responsable du cours</b>	Aymen SELLAOUTI
<b>Années Universitaires</b>	2013-2014

## Objectifs :

Ce module a été spécifiquement conçu pour les étudiants de troisième année ingénieur en Informatique Industrielle et Automatique (IIA3), possédant déjà une formation théorique et pratique approfondie en langage C acquise en première année (MPI). Il traite progressivement à la fois :

(i) Quelques notions fondamentales de la Programmation Orientée Objet, en s'appuyant sur la façon dont elles s'expriment en C++. Nous citons les classes et objets, les méthodes, les constructeurs et les destructeurs ;

(ii) Les spécificités non orientées objet du langage C++, c'est-à-dire celles qui permettent à C++ d'être un C amélioré. Nous citons le passage par référence, l'argument par défaut, la sur-définition des fonctions, les fonctions en ligne (inline), etc.

(iii) Les spécificités orientées objet du C++, telles que les fonctions amies, la sur-définition d'opérateurs, l'héritage multiple et le polymorphisme.

## Pré requis :

Algorithmique et programmation en langage C

## Plan du Cours :

### 1. Introduction au langage C++

- 1.1. Du C au C++
- 1.2. Pointeur
- 1.3. Lien entre les pointeurs et les références
- 1.4. Priorité des opérateurs
- 1.5. Surcharge des fonctions

### 2. Notions Orientée Objet en C++

- 2.1. Classes et Objets en C++
- 2.2. Notions de base de l'objet : Encapsulation/Abstraction
- 2.3. Classe et Instance
- 2.4. Classes et attributs
- 2.5. Classes et méthodes
- 2.6. Portée des membres
- 2.7. Constructeur
- 2.8. Constructeur de copie

- 2.9. Destructeur
- 2.10. Les membres statiques
- 2.11. Surcharge des méthodes
- 2.12. Surcharge des opérateurs

### **3. Héritage**

- 3.1. Introduction
- 3.2. Définition de l'héritage
- 3.3. Transitivité de l'héritage
- 3.4. Droits d'accès sur les membres hérités
- 3.5. Masquage dans la hiérarchie
- 3.6. Accès à une méthode masquée
- 3.7. Héritage et constructeur
- 3.8. Héritage et destructeur
- 3.9. Héritage et constructeur de copie

### **4. Fonctions et Classes Amies en C++**

- 4.1. Introduction
- 4.2. Principe de l'amitié
- 4.3. Mise en œuvre de l'amitié en C++
- 4.4. Exemples illustratifs

### **5. Polymorphisme en C++**

- 5.1. Introduction
- 5.2. Définition
- 5.3. Résolution des liens
- 5.4. Résolution dynamique des liens
- 5.5. Méthodes virtuelles
- 5.6. Masquage, substitution et surcharge
- 5.7. Abstraction et polymorphisme
- 5.8. Classes abstraites

### **6. L'héritage multiple**

- 6.1. Introduction
- 6.2. Définition
- 6.3. Constructeurs/destructeurs dans l'héritage multiple
- 6.4. Accès direct aux membres et méthodes hérités
- 6.5. Les classes virtuelles
- 6.6. Constructeurs et classes virtuelles

## **Bibliographie :**

- [1] C. Delannoy, C++ pour les programmeurs C, Ed. Eyrolles, 2004
- [2] C. Delannoy, Exercices en langages C++, Ed. Eyrolles, 2007
- [3] H. Sutter, Mieux programmer en C++, Ed. Eyrolles, 2000
- [4] V. T'kindt, Programmation en C++ et génie logiciel, Ed. DUNOD, 2007



# Introduction au langage C++

## Plan

- Du C au C++
- Pointeur
- Lien entre les pointeurs et les références
- Priorité des opérateurs
- Surcharge des fonctions

Programmation Orientée Objet en C++ 2

## Introduction à C++

- C++ est un langage orienté-objet. Il est issu des travaux de Bjarne Stroustrup dans les laboratoires d'AT&T Bell à la fin des années 70.
- C++ est étroitement apparenté au C, comme son nom l'indique. Il en reprend d'ailleurs l'ensemble des règles syntaxiques et constitue de fait une extension du C (d'où le terme C++, sémantiquement et lexicalement valide).
- C++ adopte le typage fort et rajoute de nouvelles bibliothèques

**C++ = C + POO**

Programmation Orientée Objet en C++ 3

## Du C à C++ (1)

- **les commentaires**
  - En C, les commentaires sont déterminés par les balises /\* et \*/.
  - Le C++ apporte une nouvelle manière de définir des commentaires à l'aide de la balise //.

`int x; // c'est une déclaration`

Programmation Orientée Objet en C++ 4

## Du C à C++ (2)

### les entrées/sorties

- En C, le programme pouvait communiquer avec l'extérieur sur trois canaux :
  - l'entrée standard : stdin
  - la sortie standard : stdout
  - l'erreur standard : stderr
- En C++, on retrouve cette notion des trois canaux sur lesquels vont travailler trois flux dédiés :
  - le flux en entrée : **cin**
  - le flux en sortie : **cout**
  - le flux d'erreur : **cerr**

## Du C à C++ (3)

### Chaque flux est en fait un objet.

- La sortie d'objets sur les flux de sortie (cout et cerr) se fait avec l'opérateur <<.
- La lecture d'objets sur le flux d'entrée (cin) repose sur l'opérateur >>.

### Pour pouvoir utiliser les flux, il faut inclure la librairie **iostream**

```
#include <iostream.h>
....
int age;
char nom[64];
cin >> nom;
cin >> age;
cout << "Coucou " << nom << ", tu as " << age << " ans" << endl;
```

## Du C à C++ (4)

### Version C

#### Exercice :

Traduire ce programme en introduisant les nouvelles techniques utilisées en C++.

```
#include <stdio.h>
#include <conio.h>
#define pi 3.14

Void main()
{
    float r,surface;
    printf("donner le rayon de cercle\n");
    scanf("%f",&r);
    surface=pi*r*r;
    printf("la surface du cercle est %f\n",
    surface);
    getch();
}
```

### Le qualificatif **const**

- Lorsque const s'applique à des variables locales automatiques, aucune différence n'existe entre C et C++, la portée étant limitée au bloc ou à la fonction concernée par la déclaration.
- lorsque const s'applique à une variable globale, C++ limite la portée du symbole au fichier source contenant la déclaration. Il devient plus facile de remplacer certaines instructions #define par des déclarations
- #define N 8 devient en c++  
**const int N = 8 ;**

## Du C à C++ (5)

### Emplacement libre des déclarations for (int i=0; i<MAX; i++) { ... }

```
int i=5;
int j=2;
for (int i=0; i<10; i++)
{
    j+=i; // on est sur le i local
}
cout << i << endl; // i vaut 5 !
cout << j << endl; // j vaut 47 !
```

## Du C à C++ (6)

### les nouveaux types : type bool

- Le C ne possède pas de type booléen spécifique.
- C++ introduit le type **bool** pour palier à cette carence. Il s'accompagne des mot-clés **true** et **false**
- La conversion **int-boolean** respecte toujours le formalisme du C.

```
bool flag=true;
....
do
{
....
if (...)
flag=false;
....
} while (flag==true);
....
```

## Du C à C++ (7)

### les nouveaux types : type référence

- En C : impossibilité de réaliser un passage de paramètres par adresse. Tous les paramètres sont passés par valeur aussi, si l'on souhaite modifier dans la fonction la valeur d'un paramètre, il faut transmettre explicitement son adresse (un pointeur sur la variable=une référence).
- Le C++ a apporté d'autres types de données : les références. Ils reposent sur une extension des attributions de l'opérateur **&**.
- Les *références* sont des synonymes d'identificateurs. Elles permettent de manipuler une variable sous un autre nom que celui sous laquelle cette dernière a été déclarée.

## Du C à C++ (8)

### les nouveaux types : type référence

- Par exemple, si « *id* » est le nom d'une variable, il est possible de créer une référence « *ref* » de cette variable. Les deux identificateurs *id* et *ref* représentent alors la même variable, et celle-ci peut être accédée et modifiée à l'aide de ces deux identificateurs indistinctement.
- il est donc impossible de déclarer une référence sans l'initialiser (doit être lié à un identificateur de variable)
- Syntaxe : `type &référence = identificateur;`

## Du C à C++ (9)

### les nouveaux types : type référence

```
int i=0; int &ri=i; // Référence sur la variable i.
ri=ri+i; // Double la valeur de i (et de ri).
```

- Il est possible de faire des références sur des valeurs numériques. Dans ce cas, les références doivent être déclarées comme étant constantes, puisqu'une valeur est une constante :  
`const int &ri=3; // Référence sur 3.`  
`ri=5; //erreur l-value specifies const object`  
`int &error=4; // Erreur ! La référence n'est pas constante.`

## Du C à C++ (10)

- Le C++ pâlie au problème du passage de paramètre par adresse. Il se distingue du passage par valeur par l'utilisation d'une **référence** dans la spécification des paramètres. Ensuite, dans le corps de la fonction, on accède "normalement" à la variable, sans avoir besoin de déréférencer.

### Version C

```
void Permut (int * a, int * b)
{
    int tmp=*a;
    *a=*b;
    *b=tmp;
}
....
int i=2, j=4;
Permut (&i, &j);
/* i vaut 4 et j vaut 2 */
```

### Version C++

```
void Permut (int & a, int & b)
{
    int tmp=a;
    a=b;
    b=tmp;
}
....
int i=2, j=4;
Permut (i, j);
// i vaut 4 et j vaut 2
```

## Du C à C++ (11)

- Suivant la norme ANSI, il existe en C deux façons de définir une fonction. Supposons que nous ayons à définir une fonction nommée `fexple`, fournissant une valeur de retour de type `double` et recevant deux arguments, l'un de type `int`, l'autre de type `double`. Nous pouvons, pour cela, procéder de l'une des deux façons suivantes :

### Version C

```
VER1
double fexple (u, v)
int u ;
double v ;
{
    ... /* corps de la fonction */
}

VER2
double fexple (int u, double v)
{
    ... /* corps de la fonction */
}
```

### Version C++

**C++ n'accepte que la seconde forme**

```
double fexple (int u, double v)
{
    ... /* corps de la fonction */
}
```

## Du C à C++ (12)

### les espaces de nommage

- En C, pour éviter le conflit de noms, on était obligé de modifier les noms de chaque structure.
- C++ propose un moyen simple de résoudre ce problème : les espaces de nommage. Ils permettent de définir une unité cohérente dans laquelle on regroupe les déclarations des différents objets (types, constantes, variables, fonctions). Les identificateurs présents dans l'espace de nommage possèdent alors une portée qui leur est spécifique.
- On définit un espace de nommage par le mot-clé ***namespace***

## Du C à C++ (13)

### Exemple

```
namespace A
{
    typedef unsigned int B;
    ....
}
....
A::B i; // une variable de type B
```

## Du C à C++ (14)

### l'allocation dynamique

- En C, l'allocation/désallocation dynamique étaient gérées par les routines malloc/free (calloc, realloc, etc...).
- Le mécanisme d'allocation dynamique a été complètement repensé dans le C++ de manière à apporter davantage de robustesse. Il repose sur deux nouveaux opérateurs :
  - **new** pour l'allocation : **new type**;  
 ○ **int \*ipt=new int**;
  - **delete** pour la désallocation : **delete ipt**;

### Pour les tableaux, elle est légèrement différente

- allocation : **tab=new int[10]**;
- désallocation : **delete[] tab**;

## Du C à C++ (15)

### Fonctions en ligne (*inline*)

- mot clé inline ; fonctions expansées en ligne (idem macros)  
`inline void Nomfonction(){};`

### Arguments par défaut

`float fct(char, int=10, float=0.0);`

- l'appel `fct('a')` est équivalent à `fct('a',10,0.0)`
- l'appel `fct('a',12)` est équivalent à `fct('a',12,0.0)`
- l'appel `fct()` est illégal

## Du C à C++ (16)

- Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste.

### Exemple :

- `float exemple (int = 1, long, int = 3) ;` est interdite.

### Les interprétations possibles de `fexple (10, 20) :`

- `exemple (1, 10, 20) ;`
- `exemple (10, 20, 3) ;`

## Pointeur

- Un pointeur est une amélioration de la notion d'adresse en mémoire
- Un pointeur c'est l'adresse en mémoire d'un objet ainsi que le type exact de l'objet qui se trouve à cette adresse.
- L'idée est que si **pi** est un « pointeur sur **un entier** », alors connaître **pi** permet de trouver cet entier dans la mémoire (**opération d'indirection**) et de le manipuler (addition, etc) puisqu'on sait que c'est un **entier**



## Pointeur...

- Supposons qu'une variable entière *i* existe dans le programme, comment obtenir l'adresse en mémoire de *i* ?
- On utilise l'opérateur **&** (**opérateur de prise d'adresse**, à ne pas confondre avec le **&** de la déclaration de référence !),
- **&i** : signifie « adresse de *i* qui est entier ». Puisque *i* est un entier, alors **&i** est du type « pointeur sur un entier ».
- A partir d'un pointeur sur un entier contenant bien l'adresse d'un entier, comment manipuler cet entier ?
  - Il faut utiliser l'opérateur **\*** (**opérateur d'indirection**)

## Lien entre les pointeurs et les références

- En effet, une variable et ses différentes références ont la même adresse, puisqu'elles permettent d'accéder à un même objet.
- Utiliser une référence pour manipuler un objet revient donc exactement au même que de manipuler un pointeur constant contenant l'adresse de cet objet.
 

**référence  $\equiv$  pointeur constant contenant l'adresse de l'objet**
- Les références permettent simplement d'obtenir le même résultat que les pointeurs, mais avec une **plus grande facilité d'écriture**.

## Lien entre les pointeurs et les références

```
int i=0;           int i=0;
int *pi=&i;        int &ri=i;
*pi=*pi+1; // Manipulation de i via pi.
ri=ri+1; // Manipulation de i via ri..
```

Nous constatons que la référence *ri* peut être identifiée avec l'expression *\*pi*, qui représente bel et bien la variable *i*.

La référence *ri* **encapsule** la manipulation de l'adresse de la variable *i* et s'utilise comme l'expression *\*pi*.

## Lien entre les pointeurs et les références

- Ecrire un programme C++ qui permet de résoudre l'équation  $ax^2+bx+c=0$ . Ce programme contient :
  - Une fonction prototype `void saisie(...)` qui permet de saisir *a,b,c*.
  - Une fonction prototype `void calcul(...)` qui calcule et affiche les résultats.
  - *a,b,c* sont des variables locales au programme principal.
- Le programme principal se contente d'appeler les fonctions *saisie* et *calcul* en mode référence



## Priorité des opérateurs

+  
 :: (opérateur de résolution de portée)  
 . -> [] () (appel de fonction) () (parenthèses) sizeof()  
 ++ -- ~ ! -(unaire) & (prise d'adresse) \* (indirection) new delete delete[]  
 () (conversion de types, cast)  
 \* / % (multiplication, division, modulo)  
 + - (addition et soustraction)  
 << >> (décalages et envoi sur flots)  
 < <= > >= (comparaisons)  
 == != (comparaisons)  
 & (ET bit à bit)  
 ^ (OU-Exclusif bit à bit)  
 | (OU-Inclusif bit à bit)  
 && (ET logique)  
 || (OU logique)  
 (? :) (expression conditionnelle ou opérateur ternaire)  
 = \*= /= %= += -= <<= >>= &= |= ~=  
 , (mise en séquence d'expressions)  
 -

## L-value

Une Left-value (valeur gauche) est un élément de syntaxe C++ pouvant être écrit à gauche d'une d'affectation.

Exemple : une variable, une case de tableau... mais bien d'autres choses

Une left-value se caractérise par :

- Un type précis (et donc une taille en mémoire) ;
- Un emplacement de stockage connu en mémoire.

## R-Value

Une Right-value (valeur droite) est un élément de syntaxe C++ pouvant être écrit à droite d'une affectation.

Exemple : une valeur, une constante, une variable, une expression, etc...

□

Une right-value se caractérise par :

- Un type précis
- Une right-value n'a pas forcément de zone de stockage en mémoire.

□

Une affectation est encore une right-value... ce qui donne le droit d'écrire : **i=j=10;** ! C'est équivalent à **j=10; i=j;**.

□ Il existe des dizaines de façons de produire des right- et leftvalues...

## Surcharge des fonctions

Chaque méthode possède une signature :

- nom de la méthode
- paramètres admis en entrée (et ordre des paramètres)
- résultat fourni en sortie (facultatif)

Une même classe peut contenir la même méthode dotée de signatures différentes

```
float calculer()
int calculer(int)
```

## Surcharge des fonctions

### ⚡ Règles de recherche d'une fonction surchargée (cas d'un seul argument)

- Correspondance exacte
- Correspondance avec promotions numériques, c'est-à-dire essentiellement :
  - char et short → int
  - float → double
- Conversions dites standard : il s'agit des conversions légales en C++, c'est-à-dire de celles qui peuvent être imposées par une affectation (sans opérateur de cast) ; cette fois, il peut s'agir de conversions dégradantes puisque, notamment, toute conversion d'un type numérique en un autre type numérique est acceptée.

## Surcharge des fonctions

### ⚡ Règles de recherche d'une fonction surchargée (cas de plusieurs arguments)

- Filtrer toutes les fonctions qui peuvent être choisies.
- Extraire les fonctions qui réalisent le moins de conversion
- Si il en résulte une seule fonction qui est meilleure que les autres alors il la choisie
- Sinon s'il y a une ambiguïté (plusieurs fonctions avec le même nombre de conversion) alors il en résulte une erreur.

## Surcharge des fonctions

### Version C

```
#include <iostream.h>
#include <conio.h>
void test(int n=0, float x=2.5)
{
    cout<<"function n°1: ";
    cout<<"n=" <<n<<"x=" <<x<<"\n";
}
void main()
{
    int i=5; float r=3.2;
    test(i,r);
    test(r,i);
    test(i);
    test(r);
    test();
    test(i,i);
    test(r,r);
    getch();
}
```

**Exercice :**  
Préciser quel fonction va être exécuté et pourquoi.

[aymen.sellaouti@gmail.com](mailto:aymen.sellaouti@gmail.com)



## Notions OO en C++

## Plan

- Programmation Orientée Objet : Introduction
- Classes, Objets en C++,
- Surcharge,
- Surcharge des opérateurs

Programmation Orientée Objet en C++

2

## Programmation impérative/procédurale (rappel)

- Dans les programmes que vous avez écrits jusqu'à maintenant, nous avons vu les notions
  - de variables/types de données
  - et de traitement de ces données
- étaient séparées :

Variables/  
constantes

opèrent sur

traitements

données

influencent

Fonctions ← arguments

ORIENTEE  
OBJETS

Séparation entre le traitement et les données

Programmation Orientée Objet en C++

3

## Critique de la programmation procédurale

```
double surface(double largeur, double hauteur);

int main()
{
    double largeur(3.0);
    double hauteur(4.0);

    cout << "La surface du rectangle est : "
          << surface(largeur, hauteur) << endl;

    return 0;
}
```

Ou est le lien entre les variables ???  
Aucune notion sémantique

Produit l h

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

Rectangle

Largeur

Hauteur

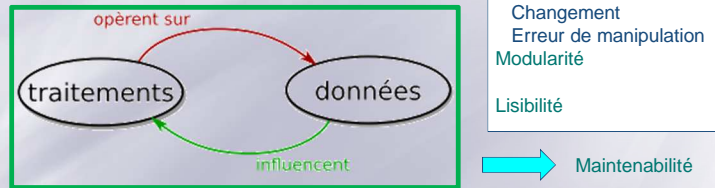
Surface

Programmation Orientée Objet en C++

4

## Programmation impérative/procédurale (rappel)

- Dans les programmes que vous avez écrits jusqu'à maintenant, les notions
  - de variables/types de données
  - et de traitement de ces données
- étaient séparées :



## Objets : quatre concepts de base

Un des objectifs principaux de la notion d'objet :

- organiser des programmes complexes grâce aux notions :

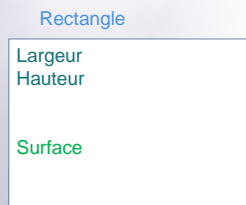
- d'encapsulation
- d'abstraction
- d'héritage
- et de polymorphisme

## Notions d'encapsulation

### Principe d'encapsulation :

- regrouper dans le même objet informatique («concept»), les données et les traitements qui lui sont spécifiques :
- attributs : les données incluses dans un objet
- méthodes : les fonctions (= traitements) définies dans un objet

✓ Les objets sont définis par leurs attributs et leurs méthodes.



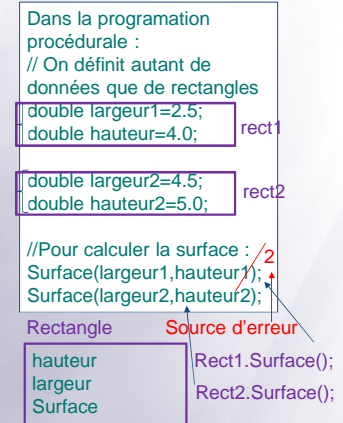
## Notion d'abstraction

- Pour être véritablement intéressant, un objet doit permettre un certain degré d'abstraction.

- Le processus d'abstraction consiste à identifier pour un ensemble d'éléments :

- des caractéristiques communes à tous les éléments
- des mécanismes communs à tous les éléments

✓ description générique de l'ensemble considéré : Se focaliser sur l'essentiel, cacher les détails.



## Notion d'abstraction : exemple

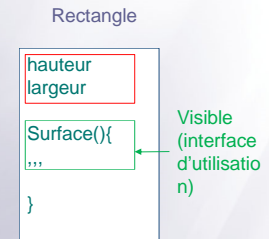
### Exemple : Rectangles

- la notion d'«objet rectangle» n'est intéressante que si l'on peut lui associer des propriétés et/ou mécanismes généraux (valables pour l'ensemble des rectangles)
- Les notions de largeur et hauteur sont des propriétés générales des rectangles (**attributs**),
- Le mécanisme permettant de calculer la surface d'un rectangle (**surface = largeur x hauteur**) est commun à tous les rectangles (**méthodes**)

## Abstraction et Encapsulation

En plus du regroupement des données et des traitements relatifs à une entité, l'encapsulation permet en effet de définir **deux niveaux** de perception des objets :

- **niveau externe** : partie « visible » (par les programmeurs-utilisateurs) :
  - l'**interface** : prototypes de quelques méthodes bien choisies
- ✓ résultat du processus d'abstraction
- **niveau interne** : détails d'implémentation
  - **corps** :
    - méthodes et attributs accessibles uniquement depuis l'intérieur de l'objet (ou d'objets similaires)
    - définition de toutes les méthodes de l'objet



## Exemple de la vie réelle

### ■ L'interface d'une voiture vis-à-vis du conducteur :

- Volant, accélérateur, pédale de frein, 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 »)

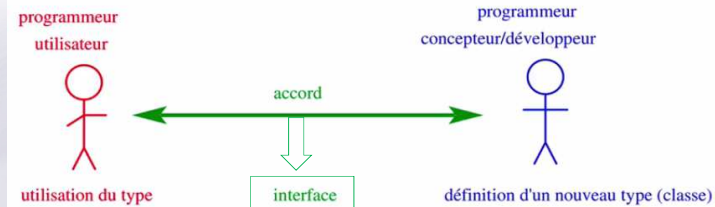
## Encapsulation et Interface

### ■ 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 et les méthodes concernant l'objet (abstraction)



## Les « 3 facettes » d'une classe



## Pourquoi abstraire/encapsuler ?

- L'intérêt de regrouper les traitements et les données conceptuellement reliées est de permettre une meilleure **visibilité** et une meilleure **cohérence** au programme, d'offrir une plus grande **modularité**.

```
double largeur(3.0);
double hauteur(4.0);

cout << "Surface : "
      << surface(largeur, hauteur)
      << endl;
```

```
Rectangle rect(3.0, 4.0);
cout << "Surface : "
      << rect.surface()
      << endl;
```

Si on change la façon d'écrire la fonction surface et qu'elle demande un tableau de valeur au lieu de deux valeurs séparées, que se passe-t-il ?

## Pourquoi abstraire/encapsuler ? (2)

- L'intérêt de séparer les niveaux interne et externe est de donner un **cadre** plus **rigoureux** à l'utilisation des objets utilisés dans un programme

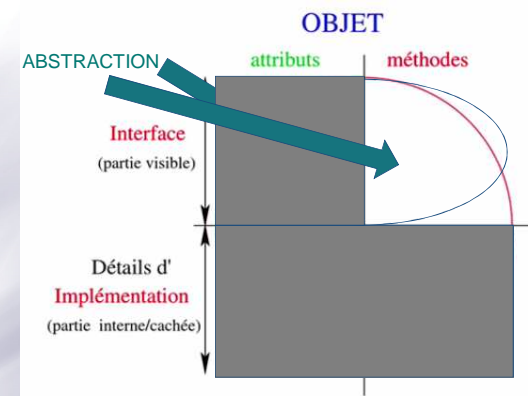
Les objets ne peuvent être utilisés qu'à travers de leurs interfaces (niveau externe) et donc les éventuelles **modifications** de la structure interne restent **invisibles** à l'extérieur (même idée que la séparation prototype/définition

- d'une fonction)

**Une voiture ne peut pas accélérer de 1000 km d'un coup**

**Règle : les attributs d'un objet ne doivent pas être accessibles depuis l'extérieur, mais uniquement par des méthodes.**

## Encapsulation / Abstraction : Résumé



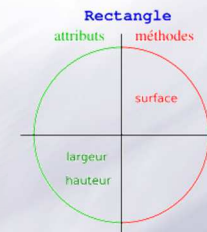
## Classes et Instances, Types et Variables

En programmation Objet :

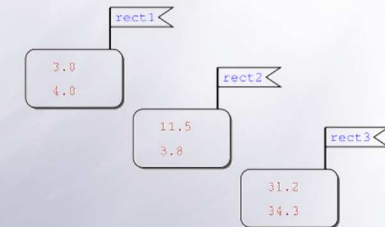
- le résultat des processus d'encapsulation et d'abstraction s'appelle **une classe**
- classe = catégorie d'objets
- une classe définit un **type**
- une réalisation particulière d'une classe s'appelle une **instance**
- instance = objet
- un objet est une **variable**

Reproduisons ces concepts sur notre exemple Rectangle

## Classes et Instances, Types et Variables (illustration)

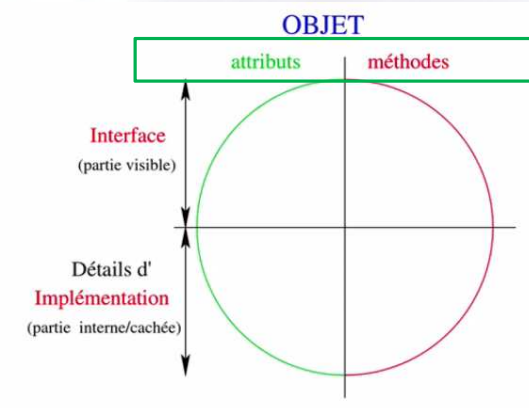


classe  
type (abstraction)  
existence conceptuelle  
(écriture du programme)



objets/instances  
variables en mémoire  
existence concrète  
(exécution du programme)

## Les classes et les méthodes en C++



Nous commençons en premier lieu par l'étude des classe et des méthodes abstraction faite de l'axe permettant de séparer l'interface des détails de l'implémentation

## Classe en C++

■ En C++ une classe se déclare par le mot-clé `class`.

■ Exemple : `class Rectangle { ... };`  
 ● Ceci définit un nouveau type du langage.

■ La déclaration d'une instance d'une classe se fait de façon similaire à la déclaration d'une variable :  
 ● `nom_classe nom_instance ;`

■ Exemple :  
 ● `Rectangle rect1;`  
 déclare une instance `rect1` de la classe `Rectangle`.



## Déclaration des attributs

La syntaxe de la déclaration des attributs est la même que celle des champs d'une structure :

`type nom_attribut ;`

Exemple :

- les attributs `hauteur` et `largeur`, de type `double`, de la classe `Rectangle` pourront être déclarés par :

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

## Accès aux attributs

- L'accès aux valeurs des attributs d'une instance de nom `nom_instance` se fait comme pour accéder aux champs d'une structure :

`nom_instance.nom_attribut`

Exemple :

- la valeur de l'attribut `hauteur` d'une instance `rect1` de la classe `Rectangle` sera référencée par l'expression :

`rect1.hauteur`

## Déclaration des méthodes d'une classe

- La Déclaration des fonctions (méthodes) d'une classe peut se faire soit :

- en une étape : déclaration + définition: Un trop grand nombre de définitions de fonctions peuvent encombrer la définition d'une classe ( fonction sera donc *inline*)
- en deux étapes : déclaration puis définition: définir les fonctions à l'extérieur de leur classe donc il faut définir les prototypes des fonctions dans la classe.

## Définition des méthodes interne à la classe

- La syntaxe de la définition des méthodes d'une classe est la syntaxe normale de définition des fonctions :

`type_retour nom_methode (type_param1 nom_param1, ...)`  
`{ // corps de la méthode }`

- mais elles sont simplement mises dans la classe elle-même.

Exemple : la méthode `surface()` de la classe `Rectangle` :

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

mais où sont passés les paramètres ?

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

## Portée des attributs

- Les attributs d'une classe constituent des variables directement accessibles dans toutes les méthodes de la classe (i.e. des « variables globales à la classe »).
- On parle de « portée de classe ».
- Il n'est donc **pas nécessaire** de les passer comme arguments des méthodes.
- Par exemple, dans toutes les méthodes de la classe Rectangle, l'identificateur hauteur (resp. largeur) fait a priori référence à la valeur de l'attribut hauteur (resp. largeur) de l'instance concernée (par l'appel de la méthode en question).

## Déclaration des méthodes

- Les méthodes sont donc :
  - des fonctions propres à la classe
  - qui ont donc accès aux attributs de la classe
- Il **ne faut donc pas** passer les attributs comme arguments aux méthodes de la classe !

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

## Paramètres des méthodes

- Mais ce n'est pas parce qu'on n'a pas besoin de passer les attributs de la classe comme arguments aux méthodes de cette classe, que les méthodes n'ont jamais de paramètres.
- Les méthodes **peuvent avoir des paramètres** : ceux qui sont nécessaires (et donc extérieurs à l'instance) pour exécuter la méthode en question !

```
class FigureColoree {
    // ...
    void colorie(Couleur c) { /* ... */ }
    // ...
};

FigureColoree une_figure;
Couleur rouge;
// ...
une_figure.colorie(rouge);
// ...
```

## Quizz

Pour la classe suivante :

```
class Personne { double taille; double poids; };
```

comment définir la méthode calculant l'indice de masse corporelle (IMC) ?

- ☐ `double imc(double poids, double taille)`  
`{ return poids / (taille * taille) ; }`
- ☐ `double imc()`  
`{ return poids / (taille * taille) ; }`
- ☐ `double imc(double taille, double poids)`  
`{ return poids / (taille * taille) ; }`
- ☐ `double imc(double taille)`  
`{ return poids / (taille * taille) ; }`
- ☐ `double imc(double poids)`  
`{ return poids / (taille * taille) ; }`

## Quizz

Pour la classe suivante :

```
class Contribuable { double fortune; };
```

comment définir la méthode calculant l'impôt à payer pour un certain taux d'imposition donné ?

- ☐ double impot(double taux) { return taux \* fortune; }
- ☐ double impot(double taux, double fortune) { return taux \* fortune; }
- ☐ double impot(double fortune, double taux) { return taux \* fortune; }
- ☐ double impot(double fortune) { return taux \* fortune; }
- ☐ double impot() { return taux \* fortune; }

## Définition externe des méthodes

Il est possible d'écrire les définitions des méthodes à l'extérieur de la déclaration de la classe

➡ meilleure lisibilité du code, modularisation

Pour relier la définition d'une méthode à la classe pour laquelle elle est définie, il suffit d'utiliser l'opérateur :: de résolution de portée :

- La déclaration de la classe contient les prototypes des méthodes
- les définitions correspondantes spécifiées à l'extérieur de la déclaration de la classe se font sous la forme :

```
typeRetour NomClasse::nomFonction(type1 param1, type2 param2, ...)
{ ... }
```

## Déclaration d'une classe en deux étapes (1)

Exemple de déclaration d'une classe Vecteur en deux étapes :

- 1ère étape : Écrire la définition de la classe

```
class vecteur
{
//définition des attributs
float x,y,z;
//déclaration des méthodes
void saisir(void);
void afficher(void);
float norme(void);
void produit(float);
void additionner(vecteur);
};
```

## Déclaration d'une classe en deux étapes (2)

La Déclaration d'une classe en deux étapes :

- 2ème étape : Implantation des méthodes

```
float vecteur::norme(void)
{float r;
r=sqrt(x*x+y*y+z*z);
return r;
}
```

```
void vecteur::afficher(void)
{
cout<<"["<<x<<" "<<y<<" "<<z<<"]";
}
void vecteur::saisir(void)
{
cout<<"x?";cin>>x;
cout<<"y?";cin>>y;
cout<<"z?";cin>>z;
}
```

## Appels aux méthodes

- L'appel aux méthodes définies pour une instance de nom `nom_instance` se fait à l'aide d'expressions de la forme :  
`nom_instance.nom_methode(val_arg1, ...)`
- Exemple : la méthode  
`void surface() const;`  
définie pour la classe `Rectangle` peut être appelée pour une instance `rect1` de cette classe par :  
`rect1.surface()`
- Autres exemples :  
`une_figure.colorie(rouge);`  
`vect.saisir();`

## Actions et Prédicats

- En C++, on peut distinguer les méthodes qui modifient l'état de l'objet (« **actions** ») de celles qui ne changent rien à l'objet (« **prédicats** »).
- On peut pour cela ajouter le mot `const` après la liste des paramètres de la méthode :

`type_retour nom_methode (typ_para1, nom_para1, ...) const`

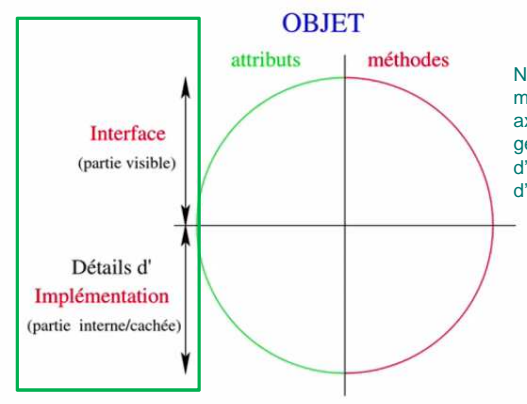
```
class Rectangle {
    // ...
    double surface() const;
};

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

Si vous déclarez une action en tant que prédicat (`const`), vous aurez à la compilation le message d'erreur :

assignment of data-member '...' in read-only structure

## Les classes et la portée des membres en C++



Nous nous intéressons maintenant au deuxième axe permettant la gestion de la notion d'encapsulation et d'abstraction

## Portée des membres (1)

- Tout ce qui n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé **private** : c'est la notion de **portée**
- Donc les données et fonctions **private** sont utilisables par les objets de la classe seulement.

```
class Rectangle {
    double surface() const;
private:
    double hauteur;
    double largeur;
};
```

- Note : Si aucun droit d'accès n'est précisé, c'est **private** par défaut.
- Donc, dans notre exemple quel est la portée de la fonction `surface()` ?

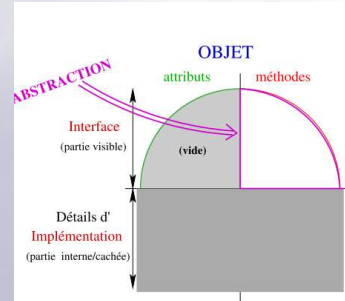
## Portée des membres (2)

- À l'inverse, l'interface, qui est accessible de l'extérieur, se déclare avec le mot-clé **public**.
- Public** : Données et fonctions utilisables par d'autres objets et fonctions.

```
class Rectangle {
public:
    double surface() const;
private:
    // ...
};
```

- Dans la plupart des cas :

- **Privé** :
  - Tous les attributs
  - La plupart des méthodes
- **Public** :
  - Quelques méthodes bien choisies (interface)



## Constructeur d'une classe (1)

- Un **constructeur** est une méthode ayant le **même nom** que la classe n'ayant **pas de type de retour**.
- Si, on n'écrit pas un constructeur, un constructeur est fourni automatiquement (**constructeur par défaut**). Ce dernier ne prend **aucun argument** et son **corps est vide**.
- Il est important de réaliser que si vous ajoutez une déclaration de **constructeur** comportant des arguments à une classe qui n'avait pas de constructeur explicite auparavant ; vous perdez le **constructeur par défaut**.
- Le constructeur sera appelé lors de l'**instanciation** simple d'un objet ou d'une **création dynamique** (utilisation de **new**)

## Constructeur d'une classe (2)

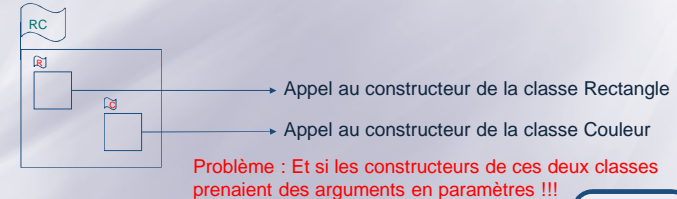
```
class vecteur
{
private:
    ...
public:
    vecteur();
    vecteur(float,float,float);
    ...
};
```

## Constructeur d'une classe contenant des objets comme attributs (1)

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

```
class RectangleCouleur {
private:
    Rectangle rectangle;
    Couleur couleur;
    //...
};
```

On a ici deux objets qui sont des attributs de la classe **RectangleCouleur**. Lors de l'instanciation d'un objet de la classe **RectangleCouleur** qu'est ce qui se passe ???  
**RectangleCouleur RC;**





## Constructeur d'une classe contenant des objets comme attributs (2)

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

...ainsi que l'initialisation des attributs de type de base.

### Solution proposée par C++ :

« liste d'initialisation » du constructeur.

Syntaxe générale :

NomClasse(liste\_paramètres)

// liste d'initialisation

: attribut1(...), // appel au constructeur de attribut1

...

attributN(...) // appel au constructeur de attributN

{ // autres opérations }

## Constructeur d'une classe contenant des objets comme attributs (3)

### Exemple

```
class Rectangle {
    Rectangle(double h, double L);
    // ...
};

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

private:
    Rectangle rectangle;
    Couleur couleur;
};
```

## Liste d'initialisation

■ Cette section introduite par « : » est **optionnelle lorsqu'on n'a pas d'objet qui nécessite un constructeur avec paramètres** mais elle est **recommandée**.

■ les attributs non-initialisés dans cette section

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

■ les attributs initialisés dans cette section 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 de recopie (1)

■ C++ offre la possibilité de créer une instance à partir d'une instance déjà existante : **la copie d'une instance**

### le constructeur de recopie (ou de copie)

```
Rectangle r1(12.3, 24.5);
```

```
Rectangle r2(r1);
```

r2 sera la copie de r1 mais de point de vue logique pas physique, i.e., une copie des valeurs des attributs

Lors de la création, r1 et r2 sont deux instances distinctes mais ayant les mêmes valeurs pour leurs attributs

## Constructeur de recopie (2)

### Syntaxe :

- Question avant d'aborder la syntaxe du constructeur :

Quand se produit le déclenchement du constructeur de recopie ?

A chaque création d'une instance à partir d'une instance existante

Qu'est ce qui se passe lors du passage d'un argument par valeur à une fonction ?

Une copie de la variable est créée et les changements à l'intérieur de la fonction se font sur la copie



La syntaxe suivante est-elle correcte ?

`NomClasse(NomClasse nomObjetSource) { ... }`

Si non quel est le problème rencontré par cette syntaxe

## Constructeur de recopie (3)

- Supposons que nous ayons définis le constructeur de recopie suivant :

• `Rectangle(Rectangle a) { ... }`

Suivons l'exécution de l'instanciation suivante :

`Rectangle r2(1.0,2.0);`

`Rectangle r1(r2);`

Passage de r2' par valeur : Création d'une copie de r2' DONC appel d'un constructeur de recopie : `Rectangle r2'(r2')`

Passage de r2 par valeur : Création d'une copie de r2 DONC appel d'un constructeur de recopie : `Rectangle r2'(r2)`

Passage de r2'' par valeur : Création d'une copie de r2'' DONC appel d'un constructeur de recopie : `Rectangle r2'''(r2'')`

Aaaaaaaaah  
ça ne finira  
jamais !!!!!



## Constructeur de recopie (3)

### Solution

`NomClasse (NomClasse const & nomObjetSource) { ... }`

Avec const, on interdit la modification de l'objet source.

Afin de supprimer la boucle infinie  
Question : Le constructeur de recopie n'est pas sensé modifier l'objet source, comment faire alors pour le protéger?

## Constructeur de recopie (4)

- Un constructeur de copie est **automatiquement généré** par le compilateur s'il **n'est pas explicitement défini** (**constructeur de copie par défaut**)

- Ce constructeur opère une **initialisation membre à membre des attributs** (si l'attribut est un objet le constructeur de cet objet est invoqué)

• **copie de surface**

- Cette copie de surface est suffisante dans la plupart des cas, cependant, il est parfois nécessaire de redéfinir le constructeur de copie, en particulier lorsque certains attributs sont des pointeurs !

- Discutons l'exemple d'une classe chaîne ayant pour attribut un pointeur `char*` comme attribut.



## Destructeur d'une classe (1)

Libérer des attributs dynamiques ou fermer de fichiers

Le destructeur est **automatiquement** lancée à l'exécution de l'ordre de libération mémoire de l'instance (donc à la fin du cycle de vie de l'objet)

Syntaxe

• `~nomdelaclass(void)`

`delete nomobjet;`

```
class vecteur
{
private:
...
public:
....
~vecteur(){...};
...
};
```

Un destructeur peut-il contenir des paramètres ?

Un destructeur possède-il une valeur de retour ?

**NON**

## Destructeur d'une classe (2)

**Besoin** : Si l'initialisation des attributs d'une instance implique la mobilisation de ressources : fichiers, périphériques, portions de mémoire (pointeurs), etc.

● Il est alors important de libérer ces ressources après usage !

**Donc, existe-il d'autres pour les quelles on aurait besoin d'un destructeur?**

**Est-ce que le destructeur peut avoir d'autres rôles à part la libération de ressources?**

Qu'aurez vous fait si vous deviez compter les instances actives d'une classes?

## Rappel de l'utilisation des objets (1)

A tout endroit où l'on peut déclarer une variable, on peut désormais déclarer un objet.

`Nomclasse Nomobjet(...);`

`Nomclasse Nomobjet; //Nomclasse Nomobjet();`

Déclaration dynamique (Déclaration de pointeurs sur objets) :  
`vecteur *v1= new vecteur();`

L'accès à un membre d'une classe se fait à l'aide de l'opérateur .

`NomObjet.membre`

## Rappel de l'utilisation des objets (2)

Déclaration

`vecteur v1(12,0.5,4);`

`vecteur v2;`

`vecteur *pv1 = new vecteur(12,0.5,4);`

`vecteur *pv2 = new vecteur();`

Accès

`v1.afficher();`

`(*pv1).afficher(); // ou pv1-> afficher();`

## Pointeur this

comment les fonctions membres, qui appartiennent à la classe, peuvent accéder aux données d'un objet, qui est une instance de cette classe ?

- À chaque appel d'une fonction membre, le compilateur passe implicitement un pointeur sur les données de l'objet en paramètre.
- Le pointeur sur l'objet est accessible à l'intérieur de la fonction membre. Il porte le nom « **this** »
- \*this** représente l'objet lui-même. Fait référence à l'objet pour lequel une fonction membre a été appelé
- Dans une fonction **non-static**, le mot clé **this** est un pointeur sur l'objet pour lequel la fonction a été appelée.

## Les membres données statiques (1)

A chaque déclaration d'une instance, celle-ci possède ses propres membres données.

Exemple :

```
class exple1
{   int n ;
    float x ;
    .... } ;
```

une déclaration telle que : **exple1 a, b ;** Conduira au schéma suivant :



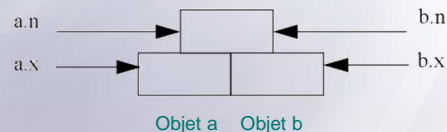
## Les membres données statiques (2)

**Question : Est-ce qu'il existe des cas où on a besoin d'une variable membre commune à tous les objets ?**

- Un membre **static** est un membre commun à tous les objets de la classe.
- Le qualificatif **static** permet de définir un membre de donnée static:

Exemple :

```
class exple2
{   static int n ;
    float x ;
    ... } ;
```



Avec la déclaration (exple2 a, b ;)

Un membre statique est accessible via la classe : **class.varstat**; Exemple : **exple2::n**

## Initialisation des membres données statiques

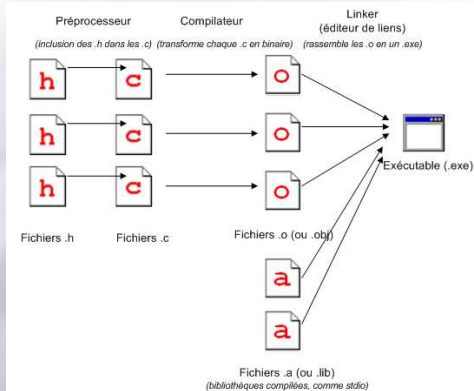
Peut-on initialiser un membre static à l'aide d'un constructeur ?

Peut-on le faire lors de la déclaration ?

**NON**

- Problème de multi initialisation dans le constructeur.**
- Problème de compilation séparée dans la déclaration dans la classe.**
- Un membre statique doit donc être initialisé explicitement (à l'extérieur de la déclaration de la classe) par une instruction telle que : **int exple2::n = 5 ;**
- Cette démarche est utilisable aussi bien pour les membres statiques privés que publics.

## Rappel sur la compilation séparée



## Surcharge des méthodes

Chaque méthode possède une signature :

- nom de la méthode
- paramètres admis en entrée (et ordre des paramètres)
- résultat fourni en sortie (facultatif)

Une même classe peut contenir la même méthode dotée de signatures différentes

```
float calculer()
int calculer(int)
```

## Exercice (1)

Définir une classe vecteur avec les données membres x et y et les fonctions membres :

- « init\_vecteur » : initialise un vecteur
- « afficher » : affiche les composantes d'un vecteur
- « prod\_scal » : calcule le produit scalaire de deux vecteurs et retourne le résultat
- Ecrire les fonctions membres associées
- Ecrire la fonction main ( ) permettant d'initialiser deux vecteurs v1 et v 2, de calculer et d'afficher leurs produits scalaires.

## Exercice(2)

On souhaite pouvoir déclarer un vecteur soit en fournissant explicitement ces 2 composantes, soit en ne fournissant aucune, auquel cas le vecteur crée possèdera deux composantes nulles

Ecrire le ou le(s) constructeur(s) correspondant(s) :

- En utilisant des fonctions membres surdéfinies
- En utilisant une seule fonction membre

### Exercice(3)

Introduire une fonction membre nommée `coincide` permettant de savoir si deux vecteurs ont même composantes :

- En utilisant une transmission par valeur
- En utilisant une transmission par adresse
- En utilisant une transmission par référence

Si `v1` et `v2` désignent deux vecteurs de type `vecteur` comment s'écrit le test de coïncidence de ces 2 vecteurs dans chacun des 3 cas considérés.

### Surcharge des opérateurs (1)

C'est quoi la **surcharge** des **opérateurs**

Surcharger une fonction :  
deux fonctions ayant le même nom mais pas les mêmes paramètres



un opérateur est une opération sur un ou entre deux opérande(s)

Redéfinir les opérateurs afin de les adapter aux besoins de la classe ou des opérandes traitées

### Surcharge des opérateurs (2)

Syntaxe :

Valuer\_De\_Retour **Operator****opérateur**(operand) ;

Exemple

```
class complex{
    double re,im;
public :
    complex(double r,double i):re(r),im(i){}
    void affichage () {cout <<"résultat est : " <<c1.im<<" " <<c1.re<<"\n";
    }
    complex operator+(complex);
    complex operator*(complex);};
```

Le programmeur définit ces deux opérateurs à l'aide de `complex::operator+(...)` et `complex::operator*(...)`.

Si `b` et `c` sont de type complexe alors,

**`b+c`** signifie **`b.operator+(c)`**

### Surcharge des opérateurs (3)

`a + b` correspond à `operator+(a, b)` ou `a.operator+(b)`

`b + a` `operator+(b, a)` `b.operator+(a)`

`-a` `operator-(a)` `a.operator-()`

`cout << a` `operator<<(cout, a)` `cout.operator<<(a)`

`a = b` `a.operator=(b)`

`a += b` `operator+=(a, b)` `a.operator+=(b)`

`++a` `operator++(a)` `a.operator++()`

`not a` `operator not(a)` `a.operator not()` ou `operator!(a)`  
`a.operator!()`

## Opérateur surchargeable

La majorité des opérateurs est surchargeable à part quelques uns qui sont :

- ::

- .

- ?:

### Remarque :

On ne peut utiliser que les opérateurs déjà existant (c'est normal, c'est de la surcharge)

Il faut conserver la pluralité (unaire, binaire) de l'opérateur initial.

Ainsi, vous pourrez surcharger un opérateur + unaire ou un opérateur + binaire, mais vous ne pourrez pas définir de = unaire ou de ++ binaire.

## Mise en œuvre (1)

Les 2 possibilités de mise en oeuvre en C++ sont:

- **Surcharge des opérateurs externes** : L'opérateur est une fonction indépendante

- Ainsi, si op est une opération binaire,

la notation **a op b** est équivalente à **operator op(a,b)**

- **Surcharge des opérateurs internes** : L'opérateur est une fonction membre d'une classe.

- Ainsi, si op est une opération binaire,

la notation **a op b** est équivalente à **a.operator op(b)**

## Mise en œuvre (2)

Les 2 possibilités de mise en oeuvre en C++ sont:

- **Surcharge interne** : (déclarés à l'intérieur de la classe) Une première méthode pour surcharger les opérateurs consiste à les considérer comme des méthodes normales membres de la classe sur laquelle ils s'appliquent. Le nom de ces méthodes est donné par le mot clé *operator*, suivi de l'opérateur à surcharger.

Syntaxe : **type operatorOp(paramètres)**

l'écriture **A Op B** se traduisant par : **A.operator Op(B)**

Avec cette syntaxe, le premier opérande est toujours l'objet auquel cette fonction s'applique

Les paramètres de la fonction opérateur sont alors le deuxième opérande et les suivants.

## Mise en œuvre (3)

Les 2 possibilités de mise en oeuvre en C++ sont:

- **Surcharge externe** : Une deuxième possibilité nous est offerte par le langage pour surcharger les opérateurs. La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci, par surcharge d'un opérateur de l'espace de nommage global. Il s'agit donc d'opérateurs externes cette fois.

- La surcharge des opérateurs externes se fait donc exactement comme on surcharge les fonctions normales. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres : il n'y aura pas de paramètre implicite

Dans le cas où les attributs à utiliser sont privés, on peut utiliser soit :

- Les accesseurs (getter) qui permettent d'accéder à ces attributs
- La notion d'amitié d'une ou de plusieurs classes (mais ça risque de briser l'encapsulation)



### Exercice(1/2)

Définir une classe tab contenant :

- Un constructeur contenant la taille max et un constructeur par recopie. Le premier constructeur allouera un tableau de taille tmax.
- Un membre privé représentant la taille maximale du tableau.
- Un membre privé gérant le nombre d'élément du tableau.
- Un pointeur sur un tableau.
- Une fonction membre afficheProp permettant d'afficher la taille maximale du tableau ainsi que le nombre d'éléments courant.

### Exercice(2/2)

- Une fonction affiche permettant d'afficher le tableau.
- Redéfinir l'opérateur d'affectation = en utilisant une fonction membre.
- Redéfinir l'opérateur + de telles sortes qu'il permette de retourner l'union de deux objets de type tab.
- Redéfinir l'opérateur - de telles sortes qu'il permette de retourner un objet de type tab privés des entiers en commun entre les deux objets tab en entrée.
- Un destructeur qui affiche le nombre d'éléments du tableau et libère l'espace mémoire alloué au tableau.



# Héritage en C++


## Plan

- Introduction
- Définition de l'héritage
- Transitivité de l'héritage
- Droits d'accès sur les membres hérités
- Masquage dans la hiérarchie
- Accès à une méthode masquée
- Héritage et constructeur
- Héritage et destructeur
- Héritage et constructeur de copie

Héritage en C++ 2

## Introduction

Commençons par un exemple concret : Les personnages de jeux vidéo



Héritage en C++ 3

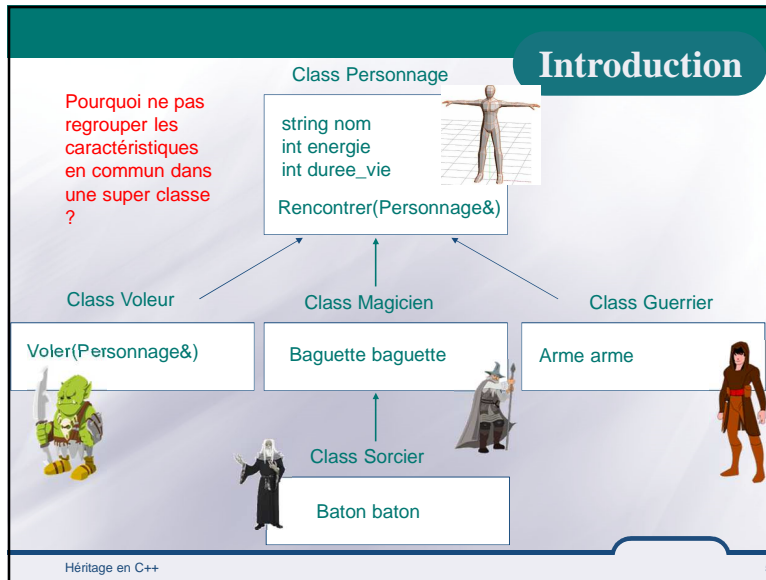
## Introduction

<b>Class Guerrier</b> string nom int energie int duree_vie Arme arme Rencontrer(Personnage&)	<b>Class Voleur</b> string nom int energie int duree_vie Rencontrer(Personnage&) Voler(Personnage&)	<b>PROBLEMES ?</b> ✓ Duplication de codes ✓ Problèmes de maintenance : Supposons qu'on veuille changer le nom ou le type d'un attribut, il faudra le faire pour chacun des classes !!!!!
<b>Class Magicien</b> string nom int energie int duree_vie Baguette baguette Rencontrer(Personnage&)	<b>Class Sorcier</b> string nom int energie int duree_vie Baguette baguette Baton baton Rencontrer(Personnage&)	

**Solution : Héritage**

Héritage en C++ 4

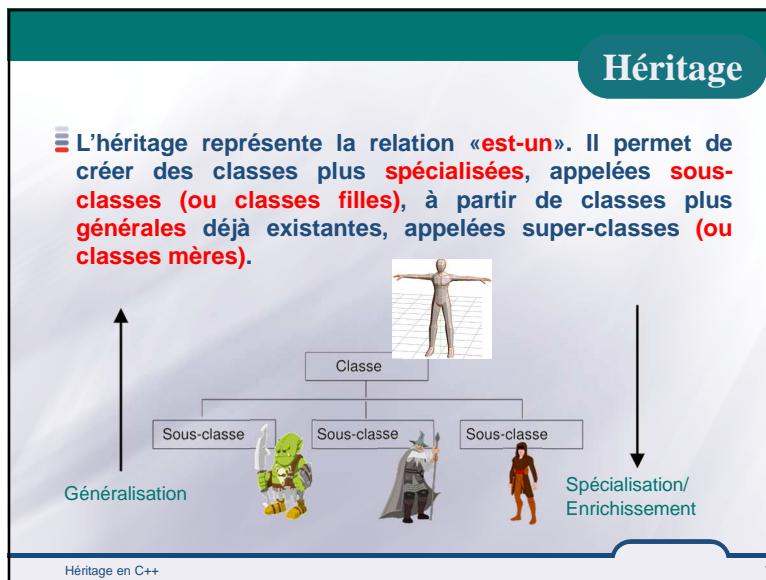




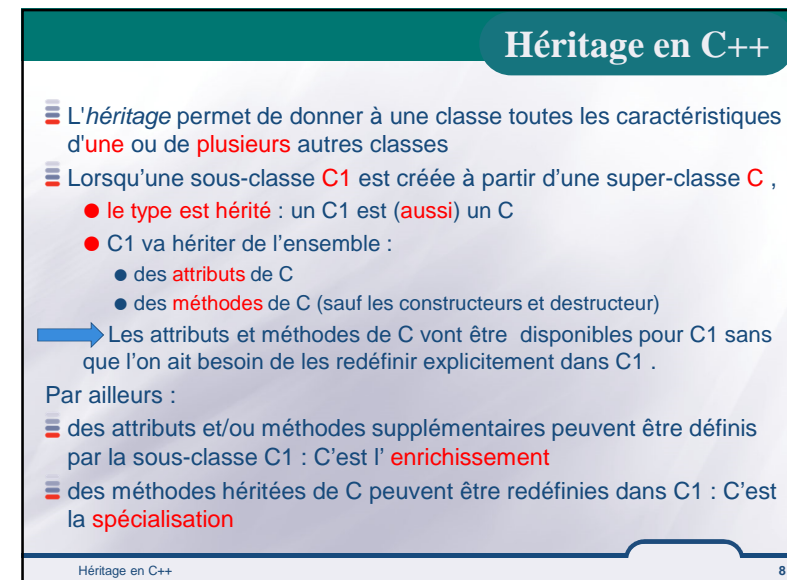
5



6



7



8

## 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);
```



Question :  
 Quel est le type de g?  
 C'est un Guerrier  
 l'opération p=g  
 Personnage avec  
 Argument p=g;

Réponse :  
 NOOONNNNNNNNN  
 Vu que les propriétés d'un guerrier  
 ne sont pas les mêmes que celles d'un  
 personnage qui est copié  
 Déjà du guerrier passe que tout  
 personnage est un guerrier

## Exemple

- Lorsqu'une sous-classe C1 (ici Guerrier) est créée à partir d'une super-classe C (ici Personnage ) :

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

L'héritage permet donc :

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



L'héritage décrit la relation « est-un » et non la relation « a-un »

Quel est alors la notion de l'orienté objet qui décrit la relation « a-un » ?

## L'ENCAPSULATION

## Transitivité de l'héritage

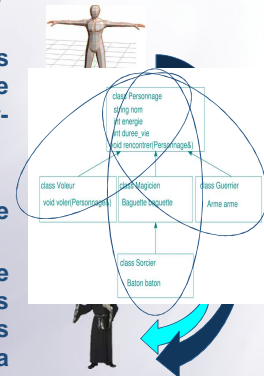
Par **transitivité**, les instances d'une sous-classe possèdent :

- 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



## L'héritage : résumons ce qu'on a vu

### 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 (plus loin !!)**)

#### On **évite** ainsi la **duplication de code**

## Syntaxe

### Comment définir une classe fille C++

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

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

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

## Droits d'accès sur les membres hérités

### Question :

- Est-ce qu'une classe fille peut accéder à tous les attributs et méthodes de la classe mère qu'ils soit public ou privé

Si la réponse est oui  
quel problème peut se  
poser ?



A chaque fois que je voudrais briser l'interface et accéder à tous les attributs de la classe même les privés, je n'ai qu'à hériter de la classe et comme ça j'aurai un accès total à tous les attributs et méthodes.

Si la réponse est non ?

Je suis la classe fille et j'ai réellement besoin d'accéder aux attributs et méthodes privés que faire ?



## Droits d'accès sur les membres hérités

Rappel : Jusqu'à maintenant, l'accès aux membres (attributs et méthodes) d'une classe pouvait être :

- soit public : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé **public**)
- soit privé : visibilité uniquement à l'intérieur de la classe (mot-clé **private**)

Un **troisième type d'accès** régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes :

- l'accès protégé : assure la visibilité des membres d'une classe dans les classes de sa descendance
- Le mot clé est « **protected** ».

## Accès protégé

- Le niveau d'accès protégé correspond à une extension du niveau privé permettant l'accès aux sous-classes.

```
class Personnage {
// ...
protected:
    int energie;
};
class Guerrier : public Personnage {
public:
// ...
    void frapper(Personnage& le_pauvre) {
        if (energie > 0) {
            // frapper le perso
        }
    }
};
```

Que se passera-t-il si on changeait protected par private?

## Accès protégé : portée

- 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

```
class A {
// ...
protected: int a;
private: int prive;
};

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 ! : this n'est pas de la même
                        // portée que autreA
    }
};
```

C'est quoi la portée de B ici?!!

## Droits d'accès sur les membres hérités (récap)

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

		mot clé utilisé pour l'héritage		
Accès aux données		<i>public</i>	<i>protected</i>	<i>private</i>
mot clé utilisé	<i>public</i>	<i>public</i>	<i>protected</i>	<i>private</i>
pour les champs et les méthodes	<i>protected</i>	<i>protected</i>	<i>protected</i>	<i>private</i>
	<i>private</i>	<i>interdit</i>	<i>interdit</i>	<i>interdit</i>

## Masquage dans la hiérarchie

Supposons qu'on ait le scénario suivant : A la rencontre d'un personnage tous les personnages saluent le personnage sauf le guerrier qui frappe tous ce qu'il voit



- Pour un personnage non- Guerrier :

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

- Pour un Guerrier

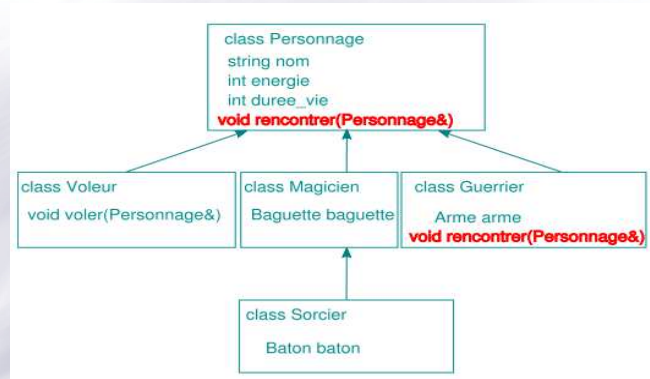
```
● void rencontrer(Personnage& le_pauvre) const {
    frapper(le_pauvre); }
```

- Faut-il re-concevoir toute la hiérarchie ?

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



## Masquage dans la hiérarchie



## Masquage dans la hiérarchie

- Donc finalement le **masquage** est : **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

## Accès à une méthode masquée

- Supposons que lorsqu'on va masquer une méthode, on a besoin de lui accéder que faire ?!!

Prenant l'exemple suivant :

- Même toujours énervé, le **Guerrier** reste toujours poli et commence par saluer le personnage rencontré avant de le frapper. **Doit-on reprendre tous le code de la méthode masquée?**
- Le nouveau fonctionnement
  - Personnage non- Guerrier :
    - Méthode générale (rencontrer de Personnage)
  - Personnage Guerrier :
    - Méthode spécialisée (rencontrer de Guerrier)
    - Appel à la méthode générale depuis la méthode spécialisée

## Accès à une méthode masquée

D'après vous quelle solution peut nous permettre d'utiliser la méthode masquée?

**Astuce :** pensez à la façon avec la quelle on définit une méthode à l'extérieur de la classe et comment on accède aux attributs et méthodes static.

Pour accéder aux attributs/méthodes masqué(e)s

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



```

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

## Héritage et constructeur

- Question : Que se passe t-il lors de l'instanciation d'un objet?

Le constructeur est appelé et généralement il construit notre objet en le préparant et en initialisant ces attributs.

- Question : Est-ce que c'est toujours le cas pour l'héritage sachant que la classe fille hérite des propriétés de la classe mère? Si oui qui fait quoi !!!!

## Héritage et constructeur

- Lors de l'instanciation d'une sous-classe, il faut initialiser
  - les attributs propres à la sous-classe
  - les attributs hérités des super-classes
- ...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.
- Comment faire alors?!!!!
- Indice : pensez à la solution que nous avons trouvée pour instancier des attributs objet d'une classe
- Solution : l'initialisation des attributs hérités doit se faire en invoquant les constructeurs des super-classes.



## Héritage et constructeur

- 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

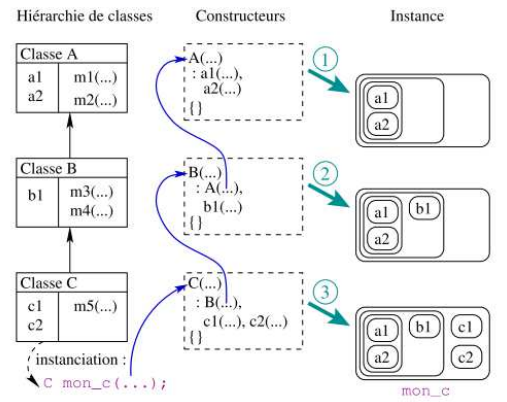
## Héritage et constructeur

- 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 {
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) {}
    // ...
};
```

## 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 !

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

Que se passera t-il si on enlève l'appel au constructeur par recopie de FigureGemetrique?

## Exercice d'application

Soit les deux classes suivantes :

```
class point
{
  Private
  int x,y ;
  public :
  point (int, int) ;
  void affiche();
  void affiche(int);
};
```

```
class pointcol : public point
{
  Private :
  int color ;
  Public :
  pointcol (int, int, int) ;
  void affiche();
  //qui affiche l'entier int et le point
};
```

- Développez les fonctions membres des deux classes.
- Comment fonctionne le passage des paramètres entre les deux constructeurs. A quoi ressemble ce mode de passage ?
- Quels problème pose les appels suivant pointcol p(1,2,3) ; p.affiche(2) ;



*[aymen.sellaouti@gmail.com](mailto:aymen.sellaouti@gmail.com)*



# Fonctions et classes amies C++

## Plan

- Introduction
- Principe de l'amitié
- Mise en oeuvre de l'amitié en C++
- Exemples illustratifs

Fonctions et classes amies 2

## Introduction

- Les membres privés ne sont accessibles qu'aux fonctions membres. Les membres publics sont accessibles de l'extérieur
- L'unité de protection est la classe
- Donc, ce principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des données privées d'une autre classe
- Ce qui impose une contrainte gênante dans certains cas

Fonctions et classes amies 3

## Introduction...

- Exemple :
  - classe vecteur
  - classe matrice
  - On veut définir une méthode permettant de calculer le produit d'une matrice par un vecteur
- Or, on ne peut définir cette fonction ni comme fonction membre de la classe vecteur, ni comme fonction membre de la classe matrice et non aussi comme fonction indépendante.

*(sauf, si on rend publiques les données de nos deux classes ou à travers des getter et des setters)*

Fonctions et classes amies 4

## Principe

- Solution : compromis entre encapsulation des données privées et des données publiques
- Lors de la déclaration d'une classe, il est possible de déclarer qu'une ou plusieurs fonctions (extérieures de la classes) sont des « **amies** »
- Une telle déclaration d'amitié les autorise alors à accéder aux données privées (au même titre que n'importe quelle fonction membre)

## Principe...

■ **L'amitié est une relation qui doit permettre à une classe d'autoriser à d'autres fonctions ou classes, dites amies, d'accéder à ses membres privés**

### ■ Avantage

- s'affranchir facilement des règles strictes d'accès aux membres privés
- permettre le contrôle des accès au niveau de la classe concernée

### ■ Mais !!!

- Protection peu être moins efficace : une fonction peut parfois se faire passer pour une autre
- Viol du principe strict d'encapsulation de données

## Principe...

■ Il existe plusieurs situations d'amitiés :

- fonction indépendante, amie d'une classe,
- fonction membre d'une classe A, amie d'une autre classe B,
- fonction amie de plusieurs classes,
- toutes les fonctions membres d'une classe, amies d'une autre classe (**classe amie**)

## Mise en oeuvre

■ **L'amitié est une relation déclarative**: Une classe doit déclarer ses amis

■ Utiliser le mot clé **friend**

## Fonction indépendante amie d'une classe

- Les fonctions amies se déclarent en faisant précéder la déclaration classique de la fonction du mot clé **friend** à l'intérieur de la déclaration de la classe cible.

### Syntaxe

#### class A

```
{ ...
friend ... fct(...);
...
};
```

- Généralement la fonction fct() possède un argument ou une valeur de retour de type A (ce qui justifie sa déclaration d'amitié).

## Exemple

/\* Fichier point.h\*/

class Point

{ private :

int x, y;

public :

Point (int a, int b) {x=a; y=b;}

**friend void Afficher(const &Point);**

/\* Fichier point.cpp\*/

#include <iostream.h>

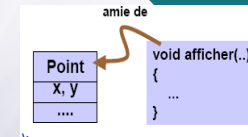
#include "Point.h"

// Nécessaire pour compiler

//afficher()

**void afficher(const Point & p)**

{ cout<< " Coordonnées" << p.x << "  
et " <<p.y << " \n ";}



```
void main()
{ Point a(1,5);
afficher(a);
Point *adp ;
adp = new Point(2,12);
afficher(*adp);
```

Il n'est pas nécessaire de déclarer le prototype de la fonction indépendante afficher() car elle figure dans la déclaration de la classe Point.

## Fonction membre d'une classe B, amie d'une autre classe A

- Même principe que le cas précédent.
- Il suffit de préciser dans la déclaration d'amitié, la classe à laquelle appartient la fonction concernée (à l'aide de l'opérateur :: )

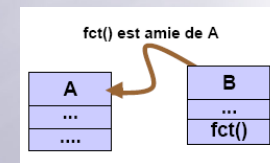
```
class A
{ ...
friend ... B::fct(...);
...
};

class B
{ ...
... fct(...); //membre
...
};

...B::fct(...){
/* On a accès aux membres
privés de tout objet de type A*/
```

## Fonction membre d'une classe B, amie d'une autre classe A...

- Pour compiler les déclarations d'une classe A, le compilateur a besoin de connaître les caractéristiques de B. **La déclaration de B devra avoir été compilée avant celle de A**
- Généralement la fonction fct() possède un argument ou une valeur de retour de type A (ce qui justifie sa déclaration d'amitié)



## Fonction membre d'une classe B, amie d'une autre classe A...

```
class A; //déclarer A pour compiler fct(...)
/* La compilation de la définition de fct(...)
nécessite la connaissance des caractéristiques
des classes A et B */
class B
{ ...
... fct(int, A);
...
};

class A
{ ...
friend ... B::fct(...);
...
};
...B::fct(int a, A a){...}
```

Fonctions et classes amies

13

## Exemple Complet

```
class PointA;
class PointB
{ private :
int x, y;
public :
PointB (int a, int b) {x=a; y=b;}
int calculer(PointA );
};

class PointA
{ private :
int x, y;
public :
PointA (int a, int b) {x=a; y=b;}
friend int PointB::calculer(PointA);
};

int PointB::calculer(PointA p){
//privés accessibles
return p.x+p.y;}

void main(){
PointA p1(15,28);
PointB p2(5,8);
cout<<p2.calculer(p1)<<endl;
}
```

Fonctions et classes amies

14

## Fonctions amies de plusieurs classes

- Une fonction (membre ou indépendante) peut faire l'objet de déclarations d'amitié dans différentes classes

```
class A
{ ...
friend void fct(A,B);
...
};

class B
{ ...
friend void fct(A,B);
...
};

void fct(A ...,B ...){/* accès aux
membres privés de n'importe quel
objet de type A ou B*/}
```

Fonctions et classes amies

15

## Fonctions amies de plusieurs classes...

- La déclaration de A peut être compilée sans celle de B, en la faisant précéder de la déclaration : **class B;**
- De même pour B
- La compilation de la fonction fct(...) nécessitera les déclarations de A et de B

Fonctions et classes amies

16



### Exemple complet

```
#include <iostream.h>
class PointA;
class PointB
{ private :
  int x, y;
public :
  PointB (int a, int b) {x=a; y=b;}
  friend int calculer(PointA, PointB );
};

class PointA
{ private :
  int x, y;
public :
  PointA (int a, int b) {x=a; y=b;}
  friend int calculer(PointA, PointB);
};

int calculer(PointA p, PointB p1){
  return p.x+p.y+p1.x+p1.y;}

void main(){
  PointA p1(15,28);
  PointB p2(5,8);
  cout<<calculer(p1,p2)<<endl;
}
```

Fonctions et classes amies

17

### Toutes Fonctions d'une classe sont amies d'une autre classe

☰ C'est une généralisation du cas 2 : où toutes les fonctions membres sont amies d'une autre classe

☰ Il est plus simple d'effectuer une déclaration globale: **classe amie**

☰ Syntaxe

- Pour dire que toutes les fonctions membres de la classe B sont amies de la classe A, on placera dans la classe A, la déclaration suivante

**friend class B**

Fonctions et classes amies

18

### Toutes Fonctions d'une classe sont amies d'une autre classe...

☰ La déclaration de A peut être compilée sans celle de B : Il suffira de la faire précéder de **class B**

☰ Ce type de déclaration d'amitié évite d'avoir à fournir les en-têtes des fonctions concernées

Fonctions et classes amies

19

### Exemple Complet

```
#include <iostream.h>
class PointA;
class PointB
{ private :
  int x, y;
public :
  PointA (int a, int b) {x=a; y=b;}
  friend class PointB;
  // toutes les méthodes de PointB
  //sont amies
};

int PointB:: calculer(PointA p1){
  return p1.x+p1.y;}

void main(){
  PointA p1(15,28);
  PointB p2(5,8);
  cout<<p2.calculer(p1)<<endl;
}
```

Fonctions et classes amies

20

## Exemple

## Calcul du produit d'une matrice par un vecteur

```

/* fichier vecteur.h */
class matrice; //pour compiler vecteur
class vecteur
{ private: double v[3];
public :
vecteur(double v1=0, double v2=0, double v3=0)
{ v[0]=v1; v[1]=v2; v[2]=v3; }
friend vecteur prod (const matrice &, const vect &);
void affiche();
};
//Déclaration de la classe mat
/* fichier mat1.h */
class vecteur; //pour compiler matrice
class matrice
{ private: double mat[3][3];
public :
matrice ();
matrice (double t[3][3]);
friend vecteur prod (const matrice &, const vect &);

```

```

//La fonction prod()
#include "vecteur.h"
#include "mat1.h"
vecteur prod (const matrice & m,
const vecteur & v)
{ int i, j;
double som;
vecteur res;
for( i=0; i<3; i++)
{ for (j=0,som=0; j<3; j++)
som+=m.mat[i][j] * v.v[j] ;
res.v[i] = som;
}
return res ;
};

```

## Exemple...

## Calcul du produit d'une matrice par un vecteur

## //programme de test

```

#include "vect1.h"
#include "mat1.h"

void main()
{ vect w(1,2,3);
vect res;
double tb[3][3]= {1,2,3,4,5,6,7,8,9};
matrice a= tb;
res = prod(a, w);
res.affiche();
}

```



# Polymorphisme en C++

## Plan

- Introduction
- Définition
- Résolution des liens
- Résolution dynamique des liens
- Méthodes virtuelles
- Masquage, substitution et surcharge
- Abstraction et polymorphisme
- Classes abstraites

Héritage multiple en C++

2

## Introduction

Reprenons notre exemple des personnages de jeux vidéo



Héritage multiple en C++

3

## Introduction

- Supposons qu'on veuille écrire un code pour un personnage qui va rencontrer des magiciens, des guerrier,... (supposons que l'on ai une collection de personnages qu'il va rencontrer)
- Que faire sachant que la façon dont un Personnage en rencontre un autre peut prendre plusieurs formes : le saluer ( Magicien ), le frapper ( Guerrier ), le voler ( Voleur )... ??!!!!



Une fonction qui va chercher le type du personnage et appeler sa fonction !!!!!!!!!!!!!

Héritage multiple en C++

4

## Introduction

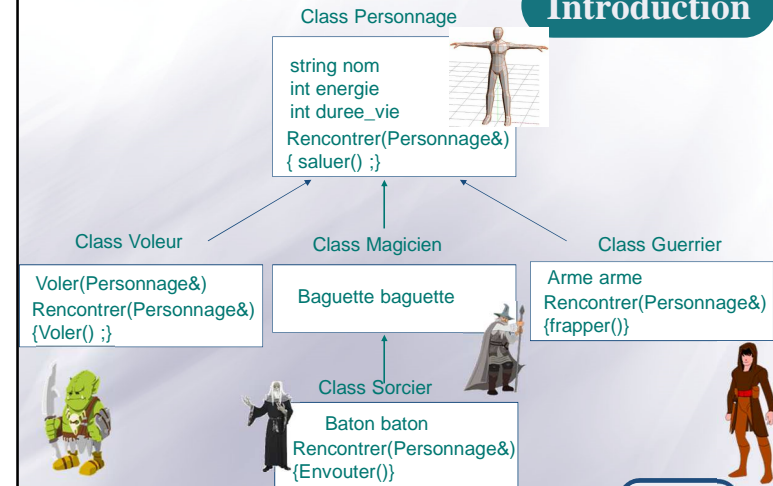
- Grâce à l'héritage, le même code pourra être appliqué à un Magicien, un Guerrier, ... qui sont des Personnage.
- Si grâce à l'héritage on peut avoir des fonctions différentes selon le personnage pourquoi ne pas avoir un mécanisme qui permet d'avoir une fonction **générique** appliquée à tous les personnages et qui s'adaptera au type du personnage en ayant un comportement différent, propre à chacun

# POLYMORPHISME

Héritage multiple en C++

5

## Introduction



Héritage multiple en C++

6

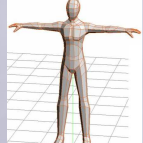
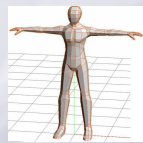
## Introduction

Etudions le pseudo code d'un joueur qui rencontrera des personnes

```

RecontrerPersos()
{
    Personnage Joueur;
    vector<Personnage> autres ;
    for (int i=0; i<autres.size();i++)
        Autres.rencontrer(joueur);
}
  
```

Si le personnage possède sa propre fonction rencontrer il l'utilise sinon il utilisera la fonction de la super classe



Pas de fonction rencontrer donc : Saluer



Envouter

Voler

Frapper

Héritage multiple en C++

7

## Introduction

### Rappel de notions d'héritage à ne pas oublier

Dans une hiérarchie de classes, la sous-classe hérite de la super-classe :

- tous les attributs/méthodes (sauf constructeurs et destructeur)
- le type : on peut affecter un objet de type sous-classe à une variable de type super-classe :



```

Personnage p;
Guerrier g;
// ...
p = g;
  
```

L'héritage est **transitif** : un Sorcier est un Magicien qui est un Personnage

Héritage multiple en C++

8

## Objets : quatre concepts de base

Un des objectifs principaux de la notion d'objet :

organiser des programmes complexes grâce aux notions :

● d'encapsulation

● d'abstraction

● d'héritage

● de polymorphisme

Héritage multiple en C++

9

## Définition

En POO, le polymorphisme est le fait que les instances d'une sous-classe, lesquelles sont **substituables** aux instances des classes de leur **ascendance** (en argument d'une méthode, lors d'affectations), **gardent** leurs **propriétés propres**.

Le choix des méthodes à invoquer se fait lors de l'exécution du programme en fonction de la **nature réelle** des instances concernées.

La mise en œuvre se fait au travers de :

- l'héritage (hiérarchies de classes) ;
- la résolution dynamique des liens.

Héritage multiple en C++

10

## La résolution des liens

Une instance de sous-classe B est substituable à une instance de super-classe A .

Que se passe-t-il lorsque B redéfinit une méthode de A ?

```
class Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};
```

```
void faire_rencontrer(
    Personnage const& un,
    Personnage const& autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;
    faire_rencontrer(g, v);
}
```

Quel fonction rencontrer sera exécutée ??  
Celle de Personnage ou celle de Guerrier

Autrement dit, est ce que c'est le type de la variable qui prime dans l'affectation de la fonction à exécuter ou c'est le type de l'objet effectivement contenu dans la variable ??

Héritage multiple en C++

11

## La résolution des liens

En C++, c'est le type de la variable qui détermine la méthode à exécuter :

● résolution statique des liens

```
class Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};
```

```
void faire_rencontrer(
    Personnage const& un,
    Personnage const& autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;
    faire_rencontrer(g, v);
}
```

Problème ?!!!! Que faire ?

Résolution dynamique des liens

Héritage multiple en C++

12



## Résolution dynamique des liens

- Il pourrait dans certains cas sembler plus naturel de choisir la méthode correspondant à la nature réelle de l'instance.
- Dans ces cas, il faut permettre la **résolution dynamique des liens** :
  - Le choix de la méthode à exécuter se fait à l'exécution, en fonction de la nature réelle des instances
- 2 ingrédients pour cela :

références/pointeurs et méthodes virtuelles



Héritage multiple en C++

13

## Déclaration des méthodes virtuelles

- En C++, on indique au compilateur qu'une méthode peut faire l'objet d'une résolution dynamique des liens en la déclarant comme **virtuelle** (mot clé **virtual**)
- Cette déclaration doit se faire dans la **classe la plus générale qui admet cette méthode** (c'est-à-dire lors du **prototypage d'origine**)
- Les redéfinitions éventuelles dans les sous-classes seront aussi **considérées comme virtuelles par transitivité**.
- Syntaxe :  
**virtual** Type nom\_fonction(liste de paramètres) [const];

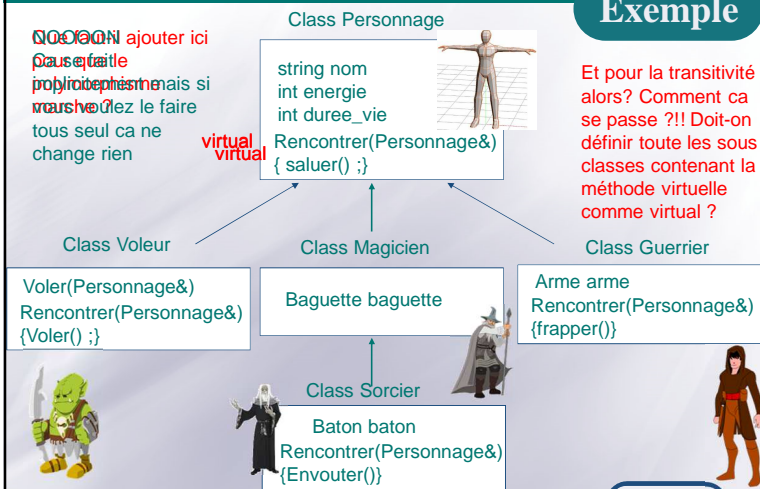
**Exemple :**

```
class Personnage {
    virtual void rencontrer(Personnage& autre) const
    { cout << "Bonjour !" << endl; }
};
```

Héritage multiple en C++

14

## Exemple



Héritage multiple en C++

15

## Exemple

Est-ce que cet exemple est correct ?

```

class Personnage {
public:
    // ...
    virtual void rencontrer(
        Personnage& p) const
    { cout << "Bonjour !" << endl; }
};

class Guerrier : public Personnage {
public:
    // ...
    void rencontrer(Personnage& p) const
    { cout << "Boum !" << endl; }
};

void faire_rencontrer(Personnage& un,
    Personnage& autre) {
    cout << un.get_nom() << " rencontre "
        << autre.get_nom() << " : ";
    un.rencontrer(autre);
}

int main() {
    Guerrier g;
    Voleur v;
    faire_rencontrer(g, v);
}
  
```

« sangoku »  
200  
kamehameha  
Guerrier

Passage par valeur donc une copie d'un objet de type Personnage « un=g »

« sangoku »  
200  
Personnage



Plus de Polymorphisme, Que faire !!!!!

Héritage multiple en C++

16

## N'oubliez jamais

- Il pourrait dans certains cas sembler plus naturel de choisir la méthode correspondant à la nature réelle de l'instance.
- Dans ces cas, il faut permettre la **résolution dynamique des liens** :
  - Le choix de la méthode à exécuter se fait à l'exécution, en fonction de la nature réelle des instances
- 2 ingrédients pour cela :

références/pointeurs et méthodes virtuelles



## Méthodes virtuelles : résumé et compléments

### En résumé :

Lorsqu'une **méthode virtuelle** est invoquée à partir d'une **référence** ou d'un **pointeur vers une instance**, c'est la **méthode du type réel** de l'instance qui sera **exécutée**.

### Attention

- Il est conseillé de toujours définir les destructeurs comme virtuels
- Un constructeur ne peut pas être virtuel
- L'aspect virtuel des méthodes est ignoré dans les constructeurs.

## Masquage, substitution et surcharge

- Nous avons rencontré trois concepts différents :
  - la **surcharge (overloading)** de fonctions et de méthodes ;
  - le **masquage (shadowing)** (en particulier de méthodes) ;
  - la **substitution (un nouveau concept (ou redéfinition, overriding))**, dans les sous-classes, de nouvelles versions de méthodes virtuelles.

Pour les méthodes virtuelles, on pourrait donc avoir les trois ?!  
Mais qui est quoi exactement ?



## Masquage, substitution et surcharge : définitions

- surcharge** : même nom, mais **paramètres différents**, dans la même portée (Note : en C++, il ne peut y avoir surcharge que **dans la même portée**).
- masquage** : entités de **mêmes noms** mais de **portées différentes**, masqués par les règles de résolution de portée. Pour les méthodes :
  - Attention aux subtilités : une seule méthode de même nom suffit à les masquer toutes, indépendamment des paramètres ! (ca n'existe pas en JAVA)
- substitution/redéfinition des méthodes virtuelles**
  - résolution dynamique** : c'est la méthode de l'instance qui est appelée (si pointeur ou référence)
  - Si l'on redéfinit qu'une seule méthode (virtuelle) surchargée, alors les autres sont masquées.

## Masquage, substitution et surcharge : exemple

```
class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};

int main() {
    B b;
    //b.m1(2); // NON : no matching function for call to 'B::m1(int)'
    b.A::m1(2); // ... mais elle est bien là
    b.m1("2");

    C c;
    c.m1(2); // Attention ici : c'est celle avec double !!
    //c.m1("2"); // NON : no matching function
    c.A::m1("2"); // OK
    c.A::m1(2); // OK, et là c'est celle avec int

    return 0;
}
```

Héritage multiple en C++

21

## Masquage, substitution et surcharge : exemple

```
class A {
public:
    virtual void m1(int i) const { cout << "A::m1(int) : " << i << endl; }
    // surcharge :
    virtual void m1(string const& s) const { cout << "A::m1(string) : " << s << endl; }
};

class B : public A {
public:
    // substitution de l'une des deux, l'autre devient hors de portée (masquage)
    virtual void m1(string const& s) const { cout << "B::m1(string)" << endl; }
};

class C : public A {
public:
    // introduction d'une 3e => masquage des 2 autres
    virtual void m1(double x) const { cout << "C::m1(double) : " << x << endl; }
};

int main() {
    B b;
    C c;
    A* pa ;

    pa = &b;
    pa->m1("2"); // OK (nous sommes dans A::)
    pa->m1(2);

    pa = &c;
    pa->m1(2.1); // Attention ici : c'est celle avec int !!
    // Nous sommes dans A::
    // pa->C::m1(2.1); // Impossible ! A n'hérite pas de C !!

    return 0;
}
```

Lorsqu'il n'y a pas de polymorphisme c'est une résolution statique des liens

B::m1(string)  
A::m1(int) N'oubliez pas que pa est un pointeur sur A  
A::m1(int)

Héritage multiple en C++

22

## Abstraction et polymorphisme

### Contexte :

Au sommet d'une hiérarchie de classe, il n'est pas toujours possible de :

- donner une définition générale de certaines méthodes, compatibles avec toutes les sous-classes,
- même si l'on sait que toutes ces sous-classes vont effectivement implémenter ces méthodes

Héritage multiple en C++

23

## Abstraction et polymorphisme

### Exemple

```
class FigureFermee {
// ...

// difficile à définir à ce niveau !..
virtual double surface(...) const { ??? }

// ...pourtant la méthode suivante en aurait besoin !
double volume(double hauteur) const {
    return hauteur * surface();
}
};
```

Peut-on définir la fonction surface de n'importe quelle forme fermée d'une manière générique ?



**Solution : déclarer la méthode surface comme virtuelle pure**

Héritage multiple en C++

24

## Méthodes virtuelles pures : définition et syntaxe

Une méthode **virtuelle pure**, ou **abstraite** :

- sert à **imposer aux sous-classes (non abstraites)** qu'elles **doivent redéfinir la méthode virtuelle héritée**
- est signalée par un **= 0** en fin de prototype,
- est, en général, **incomplètement spécifiée** : il n'y a très souvent pas de définition dans la classe où elle est introduite (pas de corps).

Syntaxe :

- virtual** Type nom\_methode(liste de paramètres) **= 0;**

```
class FigureFermee {
public:
    virtual double surface() const = 0;
    virtual double perimetre() const = 0;
};

// On peut utiliser une méthode virtuelle pure :
double volume (double hauteur) const {
    return hauteur * surface();
}
```

Exemple :

## Classes abstraites : Définition

- Une classe **abstraite** est une classe **contenant au moins une méthode virtuelle pure**.
- Elle **ne peut être instanciée**
- Ses sous-classes **restent abstraites** tant qu'elles ne fournissent pas les définitions de toutes les méthodes **virtuelles pures** dont elles **héritent**. (En toute rigueur : tant qu'elles ne suppriment pas l'aspect virtuel pur (le « =0 »).)

## Classes abstraites : exemple

- Supposons que l'on est définit la classe **personnage** comme suit :

```
class Personnage {
// ...
    virtual void afficher() const = 0;
// ...
};
```

- Et qu'une équipe de développeur crée la sous classe **Guerrier de Personnage** de la façon suivante :

```
Jeu jeu;
jeu.ajouter_personnage(new Guerrier(...));
```

- Que se passera t-il d'après vous s'ils ont oublié de définir la méthode **afficher** ?

Erreur de compilation avec le message : cannot allocate an object of abstract type 'Guerrier' because the following virtual functions are pure within 'Guerrier':  
virtual void Guerrier::afficher()

## Classes abstraites : exemple

```
class Cercle: public FigureFermee {
public:
    double surface() const override {
        return M_PI * rayon * rayon;
    }
    double perimetre() const override {
        return 2.0 * M_PI * rayon;
    }
protected:
    double rayon;
};
```

```
class Polygone: public FigureFermee {
public:
    double perimetre() const override {
        double p(0.0);
        for (auto cote : cotes) {
            p += cote;
        }
        return p;
    }
protected:
    vector<double> cotes;
};
```

- Que se passe t-il avec les classes **Cercle** et **Polygone** ?
- Cercle** définit la méthode **surface**, elle n'est plus abstraite
- Polygone** ne définit pas la méthode **surface**, elle reste abstraite







# Héritage multiple en C++

## Plan

- Introduction
- Définition
- Constructeurs/destructeurs dans l'héritage multiple
- Accès direct aux membres et méthodes hérités
- Les classes virtuelles
- Constructeurs et classes virtuelles

Héritage multiple en C++

2

## Récapitulatif

### Introduction

Nous connaissons maintenant les aspects essentiels de la POO :

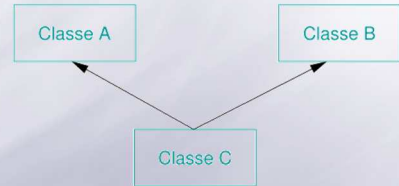
- Encapsulation et abstraction
  - regroupement traitements et données : `class Rectangle { ... };`
  - séparation entre interface et détails d'implémentation : `public`, `protected`, `private`
- Héritage : relation est-un
  - `class Guerrier : public Personnage { ... };`
- Polymorphisme
  - pouvoir être vu de plusieurs façons, abstraction, généricité
  - 2 ingrédients nécessaires : pointeurs et méthodes virtuelles
  - classes abstraites et méthodes virtuelles pures

Héritage multiple en C++

3

## Qu'est ce que l'héritage multiple

- En C++, une sous-classe peut hériter de plusieurs super-classes :



```

graph BT
    C[Classe C] --> A[Classe A]
    C --> B[Classe B]
  
```

Comme pour l'héritage simple, la sous-classe hérite des super-classes :

- tous leurs attributs et méthodes (sauf les constructeurs/destructeurs)
- leur type

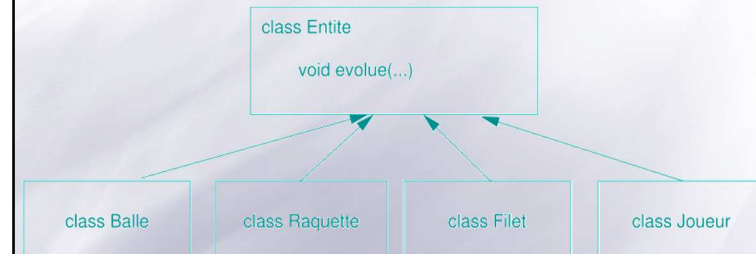
Héritage multiple en C++

4

## Exemple illustratif

- Supposons qu'on veuille réaliser un petit jeu de tennis.
- Les entités qui peuvent constituer notre jeu peuvent être les suivantes :
  - Balle
  - Raquette
  - Filet
  - Joueur
- Supposons qu'on veuille suivre l'évolution de chaque entité dans le jeu. Pour cela on utilisera la méthode `evolue`.

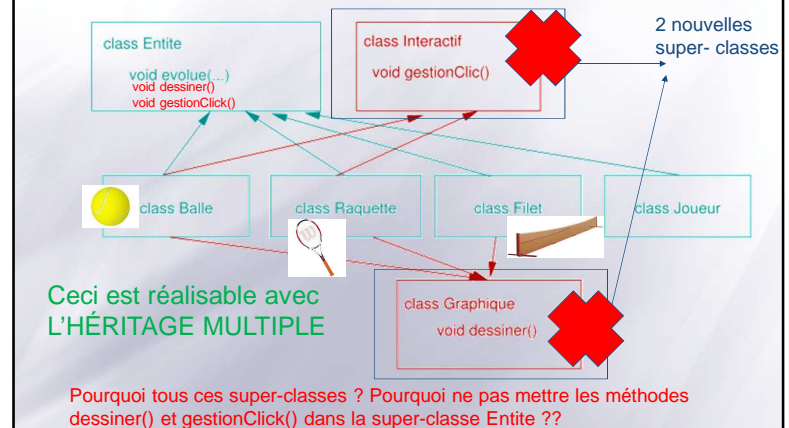
## Exemple illustratif



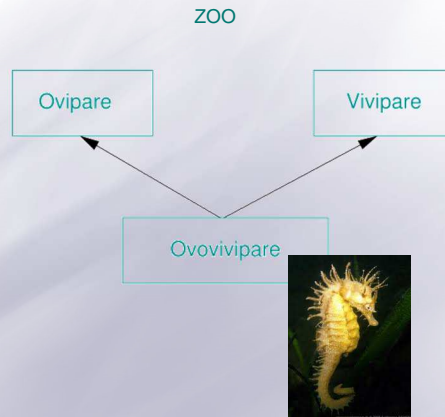
## Exemple illustratif

- Pour simplifier notre jeux on veut illustrer la vision suivante
- certaines entités doivent avoir une représentation graphique
    - ( Balle , Raquette , Filet )
  - et d'autres non
    - ( Joueur )
  - certaines entités doivent être interactives (Contrôlable par la souris) :
    - Balle , Raquette
  - et d'autres non :
    - Joueur , Filet
  - Comment doit-on organiser notre hiérarchie?

## Exemple illustratif



## Autre exemples

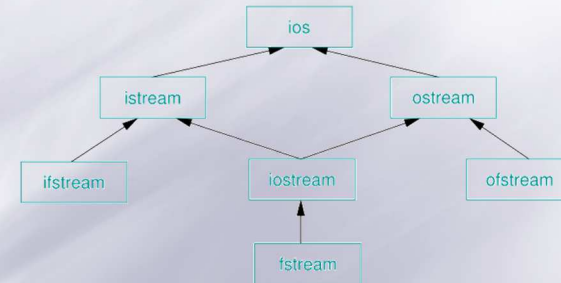


Héritage multiple en C++

9

## Autre exemples

### Bibliothèques Entrées/Sorties en C++



Héritage multiple en C++

10

## Héritage multiple

### Syntaxe :

```

class nomSousClasse: public nomSuperClasse1, ...
public nomSuperClasseN {
    //...
};
  
```

### Exemple :

```

class Ovovivipare: public Ovipare, public Vivipare {
public:
    Ovovivipare(unsigned int, unsigned int);
    virtual ~Ovovivipare();
protected:
    bool espece_rare;
};
  
```

■ L'ordre de déclaration des super-classes est pris en compte lors de l'invocation des constructeurs/destructeurs

Héritage multiple en C++

11

## Constructeurs/destructeurs dans l'héritage multiple

■ L'initialisation des attributs hérités doit être faite par invocation des constructeurs des super-classes :

### Syntaxe :

SousClasse(liste de paramètres)

```

: SuperClasse1(arguments1),
...
SuperClasseN(argumentsN),
attribut1(valeur1),
...
attributK(valeurK)
{}
  
```

■ Lorsque l'une des super-classes admet un constructeur par défaut, il n'est pas nécessaire de l'invoquer explicitement.

Héritage multiple en C++

12

## Constructeurs/destructeurs dans l'héritage multiple

```
class Ovovivipare : public Ovipare, public Vivipare {
public:
    Ovovivipare(unsigned int nb_oeufs,
                unsigned int duree_gestation,
                bool rarete = false);
    virtual ~Ovovivipare();
protected:
    bool espece_rare;

Ovovivipare::Ovovivipare(unsigned int nb_oeufs,
                        unsigned int duree_gestation,
                        bool rarete)
    : Vivipare(duree_gestation), // Mauvais ordre !!
      Ovipare(nb_oeufs),
      espece_rare(rarete)
{}
}
```

**Attention !**  
L'exécution des constructeurs des super-classes se fait selon l'ordre de la déclaration d'héritage, et non selon l'ordre des appels dans le constructeur !

L'ordre des appels des destructeurs de super-classes est l'inverse de celui des appels de constructeurs

## Accès direct aux membres et méthodes hérités

Comme dans le cas de l'héritage simple, une sous-classe peut accéder directement aux attributs et méthodes protégés de ses super-classes.

```
class Ovipare {
// ...
void afficher() const;
};
```

```
class Vivipare {
// ...
void afficher() const;
};
```

```
class Ovovivipare : public Ovipare,
                  public Vivipare
{
// ...
};
```

```
int main()
{
    Ovovivipare o(...);
    o.afficher();
}
```



Quel méthode afficher va être appelée!!! Celle de l'ovipare ou celle du vivipare !!!!!!!!!!!!!

## Accès direct aux membres et méthodes hérités

```
class Ovipare {
// ...
void afficher() const;
};
```

```
class Vivipare {
// ...
void afficher(string const& entete) const;
};
```

```
class Ovovivipare : public Ovipare,
                  public Vivipare
{
// ...
};

int main()
{
    Ovovivipare o(...);
    o.afficher("Un orvet : ");
    return 0;
}
```



**L'accès o.afficher provoquera une erreur à la compilation même si la méthode afficher n'avait pas les mêmes paramètres dans les deux classes Ovipare et Vivipare !!!**

On a deux fonction avec des signatures différentes est ce que ça résout le problème ou pas ??

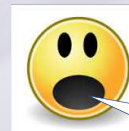
(La raison est qu'en C++, il n'y a surcharge que dans la même portée. Ici ce n'est pas une problème de surcharge, mais un problème de masquage [résolution de portée].)

## Accès direct aux membres et méthodes hérités

Quels solutions proposez vous à ce problème ?

Quel est d'après vous la solution la plus intuitive

**Solution 1 : l'opérateur de portée**



```
int main()
{
    Ovovivipare o(...);
    o.Vivipare::afficher("Un orvet : ");

    return 0;
}
```

Et depuis quand c'est l'utilisateur de la classe qui décide de son bon fonctionnement ??? C'est le rôle du concepteur de la classe NON !!!!

## Accès direct aux membres et méthodes hérités

- Solution 2 :** lever l'ambiguïté en indiquant explicitement dans la sous-classe quelle(s) méthode(s) on veut invoquer :
- ajouter à la sous-classe, une déclaration spéciale indiquant quel(s) méthode(s)/attribut(s) seront invoqué(s) exactement.
- Syntaxe:** `using NomSuperClasse::NomAttributOuMethodeAmbigu ;`
- Exemple :**

```
class Ovovivipare : public Ovipare, public Vivipare {
public:
    using Vivipare::afficher;
    // ...
};
```



Pas de parenthèse (ni prototype) derrière le nom de la méthode !

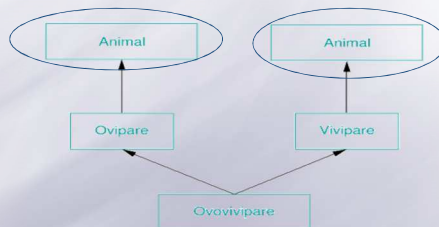
## Accès direct aux membres et méthodes hérités

- Solution 3 :**
- La meilleure solution consiste à ajouter dans la sous-classe une méthode définissant la bonne interprétation de l'invocation ambiguë.
- Exemple :**

```
class Ovovivipare: public Ovipare, public Vivipare {
public:
    // ...
    void afficher() const {
        Ovipare::afficher();
        Vivipare::afficher(" mais aussi pour sa partie"
                           " vivipare : ");
    }
    // ...
};
```

## Les classes virtuels : la problématique

- Il peut se produire qu'une super-classe soit incluse plusieurs fois dans une hiérarchie à héritage multiple :



- Les attributs/méthodes de la super-classe seront inclus plusieurs fois !
- Chaque objet de la classe Ovovivipare possèdera **deux copies** des attributs de la classe **Animal**.

## Les classes virtuels : la problématique

```
class Animal {
public:
    Animal(string const& description) : tete(description) {}
protected:
    string tete;
};

class Ovipare : public Animal { public:
    Ovipare() : Animal("à cornes") {} };
class Vivipare : public Animal { public:
    Vivipare() : Animal("de poisson") {} };

class Ovovivipare : public Ovipare, public Vivipare {
public:
    void affiche() const { cout << "j'ai une tête " << Ovipare::tete
                             << " et une tête " << Vivipare::tete << " !" << endl;
    }
    // ...
    Ovovivipare x;
    x.affiche();
};
```

- j'ai une tête à cornes et une tête de poisson !



Et depuis quand les ovovivipare ont deux têtes!!!!



## Les classes virtuels : la solution

■ Pour éviter la **duplication** des attributs d'une super-classe plusieurs fois incluse lors d'héritages multiples, il faut **déclarer son lien d'héritage avec toutes ses sous-classes comme virtuel**.

■ Cette super-classe sera alors dite « **virtuelle** » (à **ne pas confondre avec classe abstraite !!**)

■ Syntaxe :

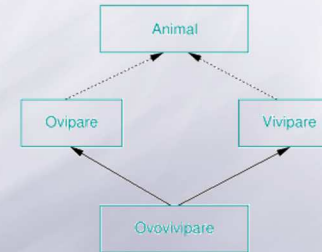
class NomSousClasse : public **virtual** NomSuperClasseVirtuelle

■ Exemple :

```
class Ovipare : public virtual Animal { // ...
// ...
class Vivipare: public virtual Animal { // ...
```

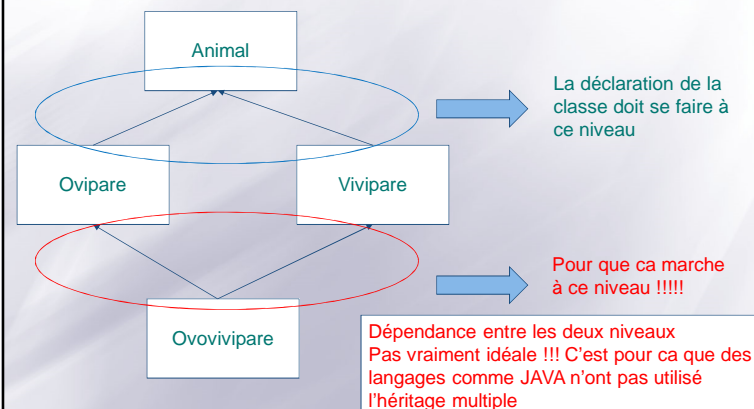
A noter que c'est la **classe pouvant être héritée plusieurs fois qui est virtuelle** (i.e. ici la super-super-classe) et non pas directement les classes utilisées dans l'héritage multiple (i.e. les super-classes).

## Les classes virtuels : la solution



■ Un **seul objet de la super-classe Animal est hérité** par l'héritage commun des sous-classes Ovipare et Vivipare .

## Les classes virtuels : la solution



## Constructeurs et Classes virtuelles

■ Rappel : dans un héritage usuel, le constructeur d'une sous-classe ne fait appel qu'aux constructeurs de ses super-classes immédiates (et ceci récursivement)

■ Dans une dérivation virtuelle, la super-classe virtuelle est initialisée directement par la sous(-sous-...)-classe instanciée

■ Toutes ses sous(-sous-...)-classes instanciables (= non-abstraites) doivent donc faire directement appel au constructeur de la super(-super-...)-classe virtuelle

## Constructeurs et Classes virtuelles : Exemple

```
Ovovivipare::Ovovivipare(string nom, Habitat habitat, Regime regime,
                        unsigned int nb_oeufs,
                        unsigned int gestation,
                        bool rarete = false)
: Animal(nom, habitat, regime),
  Ovipare(nb_oeufs),
  Vivipare(gestation),
  espece_rare(rarete)
{ }
```

## Classes virtuelles : appel des constructeurs

- Lors de la création d'un objet d'une classe plus dérivée, **son constructeur invoque directement le constructeur de la super-classe virtuelle**. Les appels au constructeur de la super-classe virtuelle dans les classes **intermédiaires** sont **ignorés**.
- Si la super-classe virtuelle a un constructeur par défaut, il n'est pas nécessaire de faire appel à ce constructeur explicitement. Il sera directement appelé par le constructeur de la classe dont on crée une instance.
- S'il n'y a pas d'appel explicite au constructeur de la super-classe virtuelle et si celle-ci n'a pas de constructeur par défaut, la compilation signalera une erreur

## Ordre des constructeurs/destructeurs

Dans une hiérarchie de classes où il existe des super-classes virtuelles :

- le soin **d'initialiser** les super-classes **virtuelles** incombe à la **sous-classe la plus dérivée**
- les constructeurs des super-classes virtuelles sont invoqués en **premier**
- les **appels explicites** au constructeur de la super-classe virtuelle dans les classes intermédiaires sont **ignorés**.
- ceux des classes non-virtuelles le sont ensuite dans l'ordre de déclaration de l'héritage
- l'ordre d'appel des constructeurs de copie est identique
- l'ordre d'appel des destructeurs est l'inverse de celui des appels de constructeurs

*[aymen.sellaouti@gmail.com](mailto:aymen.sellaouti@gmail.com)*