

## **Cours : Programmation orientée objet C++**

### **Chapitre 4: Notion de Base du langage C++**

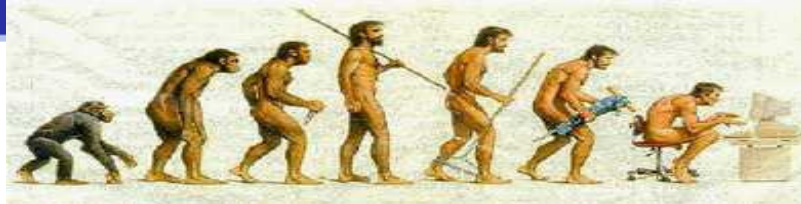
Présenter par : KMIMECH Mourad

## **Cours : Programmation orientée objet C++**

### **Sommaire**

- Historique
- Approche fonctionnelle vs. approche objet
- Différences entre Java et C++
- Notions de base de l'approche objet
- Notions de base du langage C++

# Historique



- L'assembleur : Le langage du processeur.
- 1954 : FORTRAN : Le langage des mathématiciens.
- 1960 : SIMULA : Le premier langage objet à classe.
- 1963 : BASIC : Beginner's All-purpose Symbolic Instruction Code.
- 1972 : C : Programmation système et généraliste.
- 1972 : SMALLTALK : Le premier langage interprété à objet pur.
- 1980 : C++ : Le C objet.
- 1994 : EIFFEL : L'apogée des langages objet à classes.
- 1995 : JAVA : Le langage objet populaire pour Internet.
- 2002 : C#

# Historique

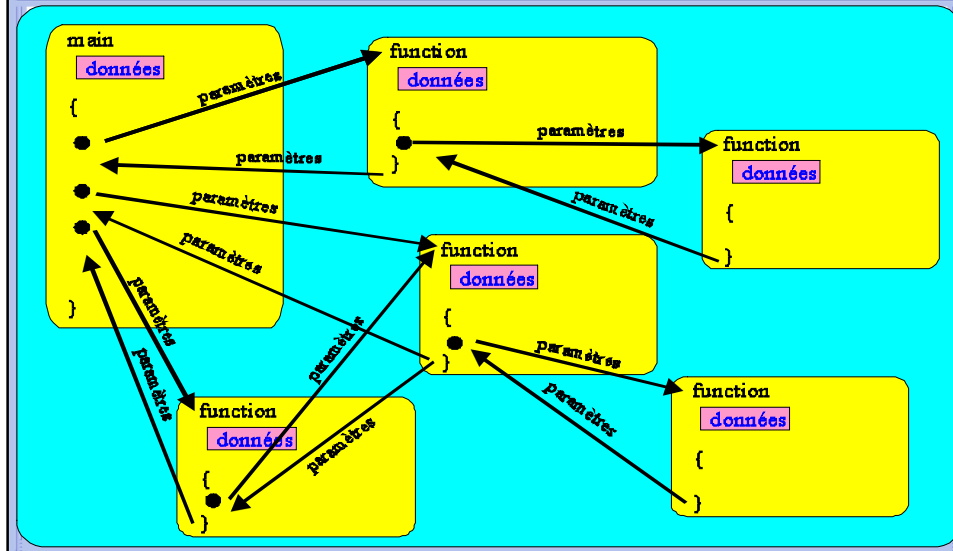
- Créé par B. Stroustrup (Bell Labs. ) à partir de 1979 ("C with classes").
- Initialement: code C++ précompilé → code C
- Devient public en 1985 sous le nom de C++.
- La version normalisée (ANSI) paraît en 1996.

C++ = C +

Vérifications de type + stricte  
Surcharge de fonctions  
Opérateurs  
Références  
Gestion mémoire + facile  
Entrées/sorties + facile  
Classes et héritage  
Programmation générique

## Approche fonctionnelle vs. approche objet

### Approche fonctionnelle : un programme en C



## Approche fonctionnelle vs. approche objet

### Limitations

- Il n'y a pas de méthode ou de cadre pour bien organiser les fonctions.
- Les modifications d'une fonction entraînent d'autres modifications dans autres fonctions, etc. – la portée d'une modification est trop grande et difficile à gérer.
- Redondance dans le code (la même chose est codé plusieurs fois)
- Propagation des erreurs – débogage difficile

## Approche fonctionnelle vs. approche objet

### Paradigme orienté objet comment peut on y arriver?

- Introduction des nouvelles (?) notions
  - objet
  - classe
  - instanciation
  - hiérarchie des classes
  - héritage
  - événement
- On va utiliser ces notions pour introduire le paradigme de programmation orientée objet.

## Approche fonctionnelle vs. approche objet

### Un programme orienté objet

- Modélisation du domaine à l'aide des classes
- Définition des classes
- Création des instances (peut être dynamique)
- Messages entre les objets (appel des méthodes)

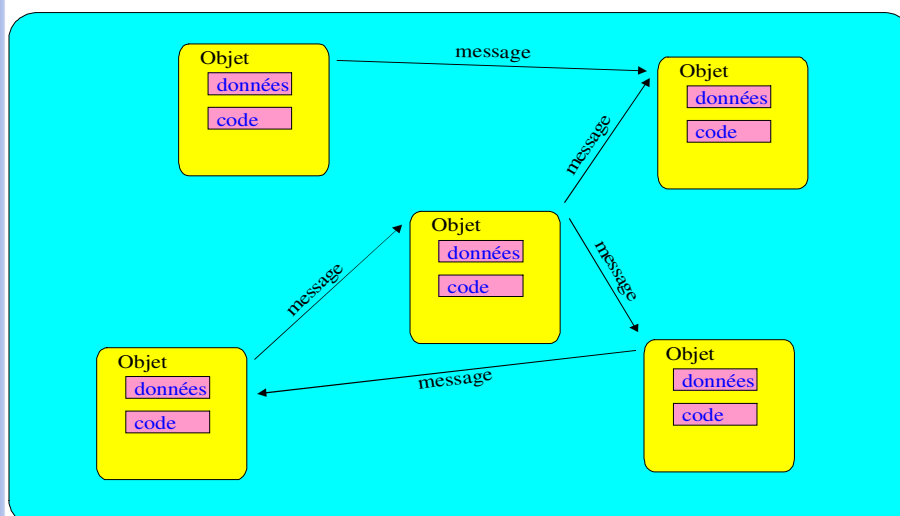
## Approche fonctionnelle vs. approche objet

### Interface vers l'utilisateur

- L'interface vers l'utilisateur est une collection des objets (boutons, champs de texte, menu déroulant, etc.)
- L'utilisateur dispose des actions sur les éléments de l'interface (sélectionner, cliquer, double cliquer, etc.)
- Une action correspond à un événement qui déclenche l'exécution d'une méthode.

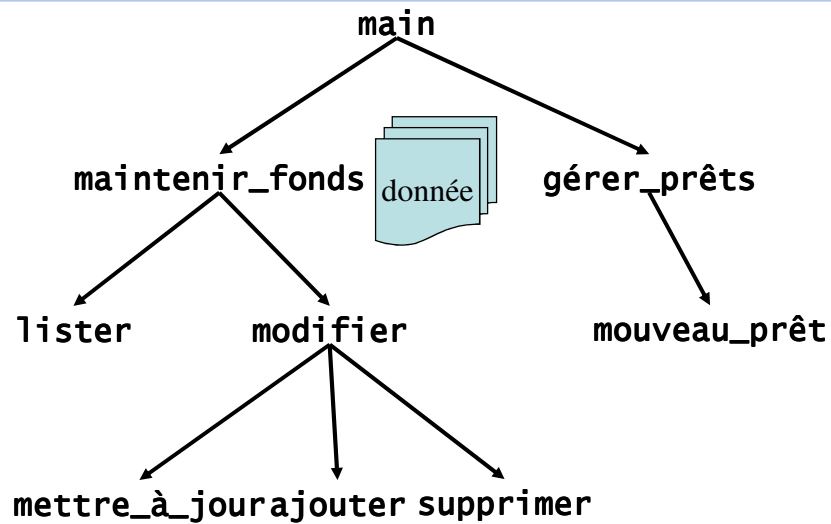
## Approche fonctionnelle vs. approche objet

### Un programme orienté objet

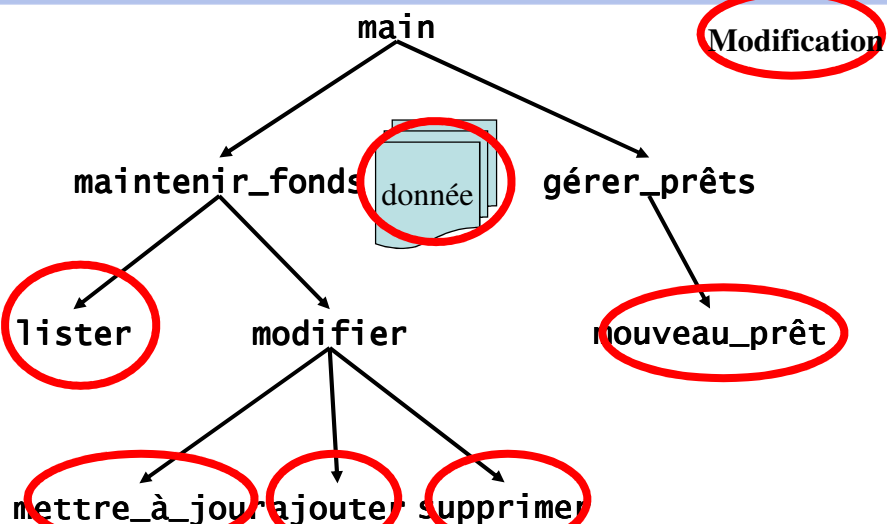


## Approche fonctionnelle vs. approche objet

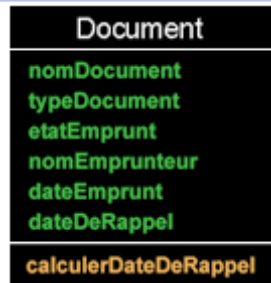
Exemple de découpe fonctionnelle d'un logiciel dédié à la gestion d'une bibliothèque



## Approche fonctionnelle vs. approche objet



## Approche fonctionnelle vs. approche objet



une entité autonome, qui regroupe un ensemble de propriétés cohérentes et de traitements associés. Une telle entité s'appelle... un **objet** !

**Approche fonctionnelle** : Dirigé par les traitements  
**Approche objet** : Dirigé par le type des données

## Approche fonctionnelle vs. approche objet

### Exemple

#### Fonctionnelle

```
void mettre_a_jour(int ref)
{ /* ... */
  switch (DOC[ref].type)
  { case LIVRE:
    maj_livre(DOC[ref]);
    break;
    case CASSETTE_VIDEO:
    maj_k7(DOC[ref]);
    break;
    case CD_ROM:
    maj_cd(DOC[ref]);
    break;
  }
  /* ... */
}
```

#### Objet

```
void mettre_a_jour(int ref)
{
  /* ... */
  DOC[ref].maj();
  /* ... */
}
```

## Approche fonctionnelle vs. approche objet

### Est-ce qu'il faut oublier le C?



**NON!**

Vous allez avoir le  
choix parmi  
plusieurs  
méthodes de  
programmation!

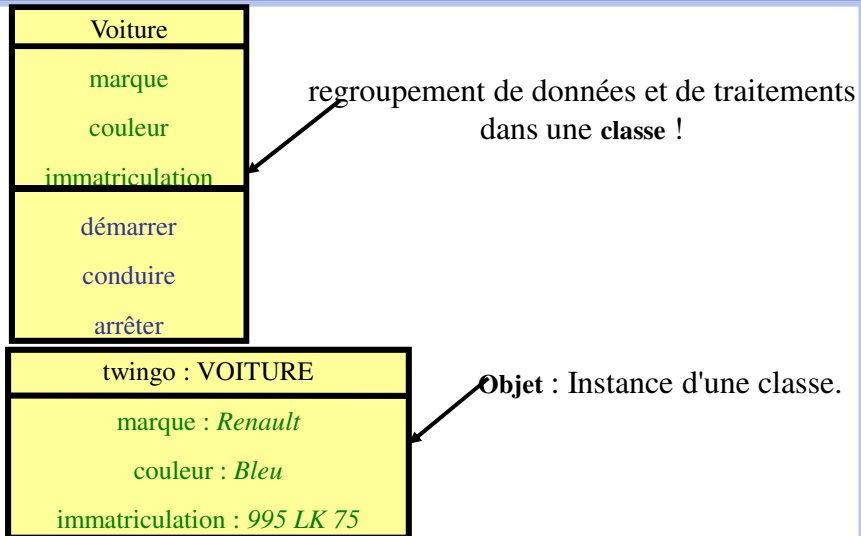
## Approche fonctionnelle vs. approche objet

- ☐ Le C est inclus (à 99%) dans le C++
- ☐ Le C++ rajoute des notions de programmation orientée objet (classe, héritage, polymorphisme... comme en Java), ainsi que des facilités d'écriture (surcharge d'opérateurs...)
- ☐ **Le C++ n'est pas seulement un superset du C, mais aussi un langage complètement différent dans ses concepts et son approche de la programmation !**



## Notions de base de l'approche objet

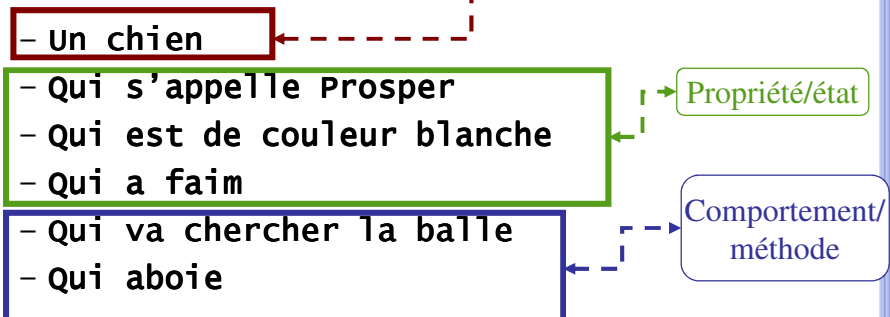
### Classes et objets



## Notions de base de l'approche objet

### Notion d'objet

- Analogie avec le monde réel :



## Notions de base de l'approche objet

### Notion de classe

**Classe** : Représentation abstraite d'une catégorie (=type) d'objets.

**CHIEN**

nom :
couleur :
affamé :
aboyer :
chercher(Balle) :

## Notions de base de l'approche objet

### Notion d'objet

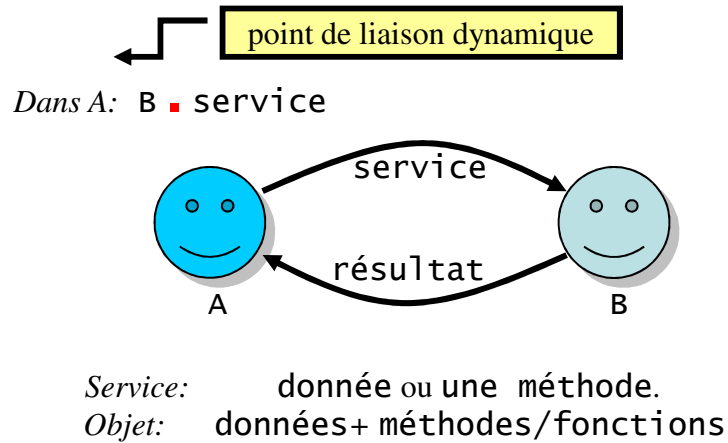
**Objet** : Représentation en mémoire d'une entité physique  
(appartenant à une classe)

**CHIEN**: **monChien**

nom : <b>Prosper</b>
couleur : <b>Blanc</b>
affamé : <b>Oui</b>
aboyer :
chercher(Balle) :

## Notions de base de l'approche objet

### Communication inter-objets



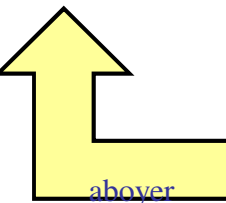
## Notions de base de l'approche objet

### Communication inter-objets

**Communication** : Invocation de message en un point de liaison dynamique

monChien

nom : Prosper



moi

...  
monChien.aboyer()  
...

## Notions de base de l'approche objet

### Notion d'encapsulation

- Masquer les détails d'implémentation d'un objet, en définissant une **interface**.
- **L'interface** : vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.
- L'encapsulation **facilite l'évolution** d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface.
- L'encapsulation **garantit l'intégrité des données**, car elle permet d'interdire l'accès direct aux attributs des objets (utilisation d'accesseurs).

## Différences Java et C++

### ☐ Java

- Création des objets par allocation dynamique(*new*)
- Accès aux objets par références
- Destruction automatique des objets par le ramasse miettes

### ☐ C++

- Allocation des objets en mémoire statique(variables globales), dans la **pile(variables automatiques)** ou dans le **tas(allocation dynamique)**,
- Accès direct aux objets ou par pointeur ou par référence
- Libération de la mémoire à la charge du programmeur dans le cas de l'allocation dynamique.**

- ☐Autres possibilités offertes par le C++: Variables globales, compilation conditionnelle (préprocesseur), pointeurs, surcharge des opérateurs, patrons de classe *template* et *héritage multiple*

24

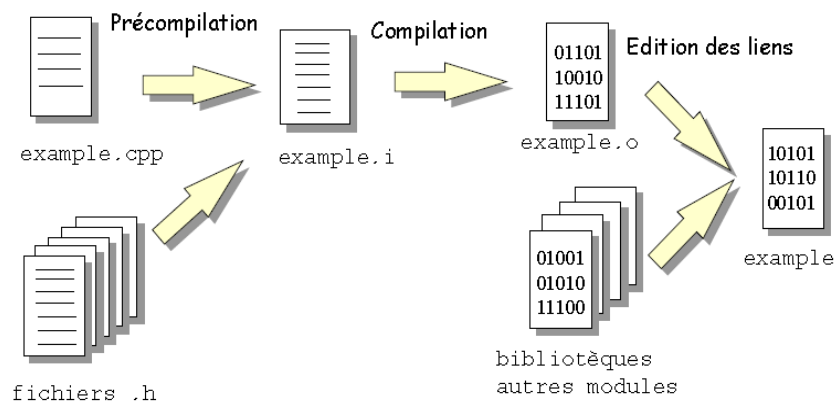
## Notions de base

### Fichiers Sources

- ❑ Un programme est généralement constitué de plusieurs modules
- ❑ Chaque module est composé de deux fichiers sources:
  - ❑ un fichier contenant la description de l'interface du module
  - ❑ un fichier contenant l'implémentation proprement dite du module
- ❑ Un fichier utilisation comportant un **void main()**
- ❑ Un suffixe est utilisé pour déterminer le type de fichier
  - **.h .H .hpp .hxx** pour les fichiers de description d'interface (header files ou include files)
  - **.C .cc .cxx .cpp .c++** pour les fichiers d'implémentation

## Notions de base

### Fichiers Sources



## Notions de base

### Commentaire

#### ☐ C et C++

*/\* Un commentaire en une seule ligne \*/*

*/\**

*\* Un commentaire sur plusieurs*

*\* lignes*

*\*/*

#### ☐ C++ uniquement

*// Un commentaire jusqu'à la fin de cette ligne*

*//*

*// Un commentaire sur plusieurs*

*// lignes*

*//*

## Notions de base

### Types fondamentaux

#### ☐ Booléen

- o **bool**

- o deux valeurs constantes de type bool: true et false

#### ☐ Entiers de tailles différentes

- o **char** (1 octet)

- o **short** int ou short (2 octets)

- o **int** (2 ou 4 octets)

- o **long int** ou long (4 ou 8 octets)

#### ☐ Valeurs en virgule flottante

- o **float** (4 octets)

- o **double** (8 octets)

- o **long double** (10 ou plus octets)

## Notions de base

### Types fondamentaux

- ❑ Entiers positifs
  - **unsigned char**
  - **unsigned short**
  - **unsigned int**
  - **unsigned long**
- ❑ Entiers avec signe
  - **signed char**
  - **signed short  $\cong$  short**
  - **signed int  $\cong$  int**
  - **signed long  $\cong$  long**
- ❑ Pour avoir la plage des valeurs pour chaque type, voir les fichiers
  - **limits.h** pour les types entiers
  - **float.h** pour les types en virgule flottante

## Notions de base

### Définition de variables

- ❑ Tout objet(variable) en C++ a une classe(type)
- ❑ Valide mais à éviter
  - float price, discount;**
- ❑ Plutôt
  - float price;**
  - float discount;**
- ❑ Majuscules et minuscules sont différenciées
  - float distance;**
  - int Distance; // "Distance" et "distance" sont deux variables différentes**
- ❑ C++ permet de déclarer des variables n'importe où et de les initialiser.
- ❑ On peut donner une valeur initiale lors de la définition
  - unsigned int minDistance = 15;**
  - double angle = 245.38;**

## Notions de base

### Définition de variables

file1.cpp

```
float maxCapacity;
```

file2.cpp

```
extern float maxCapacity;  
float limit = maxCapacity*0.90;
```

- Si la variable est définie dans un autre module il faut la déclarer avant de l'utiliser

```
extern float maxCapacity; // declaration
```

```
float limit = maxCapacity*0.90; // usage
```

## Notions de base

### Conversion de type

- ❑ Le C++ autorise les conversions de type entre variable de type :

**char <--> int <--> float <--> double**

Exemples :

conversion simple	lors de l'appel d'une méthode
<pre>int x=2 ; float y=x ;</pre>	<pre>void f(int, int) ; float a=2.2 ; int b=2 ; f(a, b) ;</pre>



## Notions de base

### Les arguments par défaut

En C++ on peut préciser la valeur prise par défaut par un argument d'une fonction. Lors de l'appel à cette fonction, si on ne met pas l'argument, il prendra la valeur par défaut, dont le cas contraire, la valeur par défaut est ignorée.

Exemple :

```
void f1(int n=3){...}
void f2 (int n, float x=2.3) {...}
void f3(char a, int b=21, float c=5){...}
void main(){
    char a='x'; int i=2; float r=3.2;
    f1(i); //appel de f1 avec n=2
    f1(); //appel de f1 avec n=3
    f2(i,r); //appel de f2 avec n=2 et x=3.2
    f2(i); //appel de f2 avec n=2 et x=2.3
    f3(a,i,r); //appel de f3 avec ...
    f3(a); //appel de f3 avec ...
    f3(); //erreur
```

## Notions de base

### Les arguments par défaut

Les arguments dont les valeurs sont définies par défaut doivent obligatoirement être situés à la fin de la liste des arguments.

Exemple :

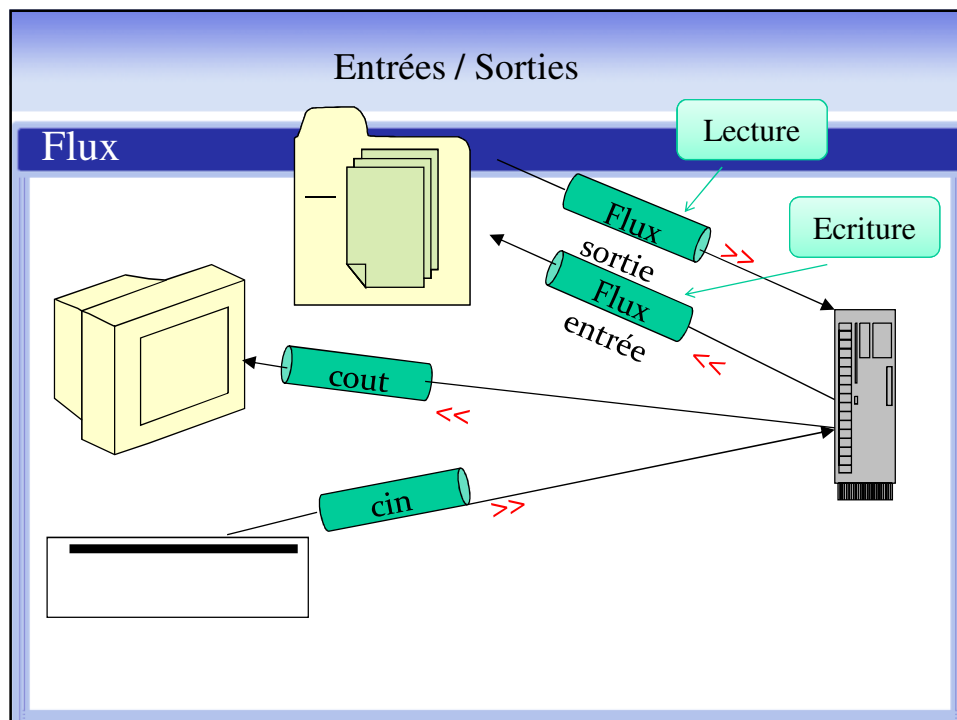
```
void f1(int x, int n=3){...} //ok
void f2 (int n=2, float x) {...} //erreur
void f3(char a, int b=2, float c){...} //erreur

//correction de f2 et f3
void f2 (float x, int n=2) {...} //ok
void f3(char a, float c, int b=2){...} //ok
```

## Notions de base

### Standard Template Library

- **S**tandard **T**emplate **L**ibrary: STD
- Bibliothèque standard (elle fait partie de la norme C++) qui offre des types de variables et des fonctions utiles
- Toutes les noms sont précédées de std::
- std::string, std::vector, std::list, std::sort...



## Entrées / Sorties

C	<pre>#include &lt;stdio.h&gt; int value = 10; printf( "value = %d\n", value ); printf( "New value = ??\n" ); scanf( "%d", &amp;value );</pre>
C++	<pre>#include &lt;iostream&gt; using namespace std; int value = 10; cout &lt;&lt; "Value = " &lt;&lt; value &lt;&lt; endl; cout &lt;&lt; "New value = ?? " &lt;&lt; endl; cin &gt;&gt; value;</pre>

Les opérateurs '<<' et '>>' sont surchargeables.

## Exercice 1

Écrire un programme C++ qui permet la saisie de deux entiers A et B. Calculer puis afficher leur somme et leur produit.

**Solution :**

```
#include <iostream.h>
using namespace std;
void main()
{
    int A, B;
    cout<<"donner A= ";
    cin>> A;
    cout<<"donner B= ";
    cin>> B;

    cout<< "la somme de A et B = "<<A+B<< endl;
    cout<< "le produit de A et B = "<<A*B<< endl;
}
```

## Notions de base

### La sur définition d'une fonction : surcharge

C++ autorise la définition de fonctions différents et portant le même nom à condition de les différencier par le type des arguments.

#### Exemple :

```
void test (int n=0, float x=2.3)
{ cout<<"fonction 1 avec n = "<<n<<"et x= "<<x ; }
void test (float x=3.4, int n=5)
{ cout<<"fonction 2 avec n = "<<n<<"et x= "<<x ; }
void main(){
    int i=3 ; float n=3.3 ;
    test(i,n) ; //appel f1
    test(n,i) ; //appel f2
    test(i) ; //appel f1
    test(n) ; //appel f2
    test(); // erreur dans ce cas
}
```

## Notions de base

### Les opérateurs new et delete

Les 2 opérateurs **new** et **delete** remplacent les fonctions de gestion dynamique de la mémoire malloc et free. Ils permettent donc de réserver et de libérer une place mémoire.

#### Exemple 1 en C/C++ :

```
int *p ;
long nb=12;
p=(int *) malloc (nb*sizeof(int));
...
free(p) ;
```

#### Exemple 2 en C/C++ :

```
int *pi; float *pr ;
pi=new int;//allocation d'une seule valeur
pr= new float[50];
...
delete pi;
delete pr;
```

**Remarque :!! il ne faut pas utiliser conjointement (malloc et delete) ou (new et free)**

## Notions de base

### Notion de référence

En C l'opérateur & désigne l'adresse, en C++ il peut désigner soit l'adresse soit une référence selon le contexte. Seul le contexte de programme permet de déterminer s'il s'agit d'une référence ou d'une adresse.

#### Exemple :

```
int n=3 ;  
int &p=n ; //p et n ont la même @ mémoire  
cout<< p ; // affiche 3
```

## Notions de base

### Passage de paramètres par référence

En C un sous-programme ne peut modifier la valeur d'une variable locale passée en argument d'une fonction que si on passe l'adresse de cette variable.

#### Exemple 1 en C/C++ :

```
//passage par valeur  
void permutation( int a, int b){  
    int c ;  
    c=a ; a=b ; b=c ;}  
void main()  
{  
    int x=2 ;int y=3 ;  
    permutation(x,y) ;  
    //après l'exécution le changement ne se fait pas : x=2 et y=3  
}
```

## Notions de base

### Passage de paramètres par référence

En C un sous-programme ne peut modifier la valeur d'une variable local passée en argument d'une fonction que si on passe l'adresse de cette variable.

**Exemple 2 en C++ :**

```
//passage par adresse
void permutation( int *a, int *b) { /*a représente le contenu de a
int c ;
c=*a ; *a=*b ; *b=c ;}
void main(){
int x=2 ;int y=3 ;
permutation(&x,&y) ;// après l'exécution x=3 et y=2
}
```

## Notions de base

### Passage de paramètres par référence

**Remarque !!!** En C++ on préfère d'utiliser le passage par référence qui passe par adresse.

**Exemple en C++ :**

```
void permutation ( int &a, int &b){
    int c ;
    c=a ; a=b ; b=c ;}
void main(){
int x=2 ;int y=3 ;
permutation(x,y) ;// après l'exécution x=3 et y=2
}
```

## Notions de base

### Notion de référence

#### Remarques :

a- Une référence doit être toujours initialisée :

`int &p ;//incorrecte`

b- On ne peut pas référencer une constante

`int &p=3 ; //incorrecte`

c- On ne peut pas référencer une expression

`int &p=n+2;// incorrecte`

d- Une référence ne doit pas modifier

`int &p=n ;`

`p=q ;//correcte`  `n=q`

`&p=q ;//incorrecte`

## Notions de base

### Utilisation d'une référence comme valeur de retour d'une fonction

Le mécanisme de passage par référence peut être appliqué à la valeur de retour d'une fonction

#### Exemple 1 :

```
int &f()// f retourne une référence sur entier
int *f()// f retourne un pointeur sur entier
```

#### Exemple 2 :

```
int &f(){
    int n =3;
    return n ;
}
void main(){
    int p;
    p=f() ;
    /* cette appelle affecte à p la valeur (3) située à l'emplacement mémoire référencé par f()*/
    cout<<"p=" <<&p <<endl;
    /*le programme affiche l'adresse de p (ex: p=0x7fff4200aaac */
```

## Notions de base

### Les classes

#### Introduction

- ❑ Le mécanisme de classe en C++ permet aux programmeurs de définir des nouveaux types propres à leurs applications.
- ❑ La classe est une extension très forte à la notion de structure de données du langage C : au lieu d'une classe, nous pouvons définir des données et des méthodes. Ces entités constituent les membres de classe.

## Notions de base

### Les classes

#### Définition d'une classe

- ❑ En C++ la syntaxe d'une classe est la suivante :

```
class NomClasse {  
    // Modificateur d'accès pour les données membres  
    //Données membres  
    // Modificateur d'accès pour les fonctions membres  
    //Fonctions membres  
};
```

**Remarque :!!** Ne pas oublier le ; à la fin de la classe

- ❑ Le code d'une fonction membre peut se faire dans ou en dehors de la classe. Dans le second cas, le nom de la méthode sera préfixé par le nom de la classe suivi de l'opérateur ::



## Notions de base

### Les classes

#### Définition d'une classe

##### Protection des membres d'une classe

❑ Il est possible d'empêcher l'accès des champs ou de certaines méthodes à toute fonction autre que celles de la classe. Cette opération s'appelle l'encapsulation. Pour la réaliser, il faut utiliser les mots clés suivants :

▪ **public** : les accès sont libres ;

▪ **private** : les accès sont autorisés dans les fonctions de la classe seulement ;

▪ **protected** : les accès sont autorisés dans les fonctions de la classe et de ses descendantes (voir le chapitre d'héritage) seulement. Le mot clé *protected* n'est utilisé que dans le cadre de l'héritage des classes

## Notions de base

### Les classes

#### Objet et instance d'une classe

```
Class StructureClasse{  
    private: // attributs et méthodes privées :  
    public: // attributs et méthodes publiques :  
    protected: // attributs et méthodes protégées :  
};
```

## Notions de base

### Les classes

#### Objet et instance d'une classe

Par défaut, tous les éléments d'une classe sont privés.

##### Exemple :

```
class Client{  
    // private est à présent inutile.  
    char Nom[21], Prenom[21];  
    unsigned int Date_Entree;  
    int Solde;  
    public: // Les données et les méthodes publiques.  
    bool dans_le_rouge(void);  
    bool bon_client(void);  
};  
//...Client A, B ;  
int s= A.Solde ; //erreur car Solde est privé  
bool b= A.dans_le_rouge() ; //ok car cette méthode est publique
```

## Notions de base

### Les classes

#### Définition d'une classe

```
Class Point{  
    private : // Déclaration de 2 données membres de type réel  
        float abs ;  
        float ord;  
  
    public: // Déclaration d'une fonction définie à l'intérieur de la classe  
    void initialise (float a, float b)  
    {   abs=a;  
        ord=b;  
    }  
    // Déclaration d'une fonction définie à l'extérieur de la classe  
    void deplacer(float dx, float dy);  
};  
  
void Point::deplacer(float dx, float dy)  
{   abs+=dx;  
    ord+=dy;  
}
```

## Notions de base

### Les classes

#### Objet et instance d'une classe

□ Un objet correspond à la localisation physique d'un exemplaire (ou instance) de la classe. Chaque instance de la classe (ou objet) possède ses propre exemplaire de données (non statiques) de la classe.

□ Les données membres (non statiques) d'une classe seront appelées généralement les composants d'un objet. Exemple :

```
class Point{  
    static int nb ; int abs ; int ord ; /*nb est une donnée membre  
    statique alors que abs et ord ne sont pas statique*/  
}
```

□ Les fonctions membres (non statiques) d'un classe, quand-à elles, sont définies pour être appliquées sur des instances de la classe. Par exemple : P1.affiche() ;

## Notions de base

### Les classes

#### Objet et instance d'une classe

La création d'une instance sous-entend un mécanisme d'allocation mémoire. À chaque allocation mémoire correspond une durée de vie particulière :

- **Durée de vie statique** : les objets sont alloués une fois pour toute au moment du chargement, leur durée de vie est celle du programme. Les objets statiques sont ceux créés par une déclaration située en dehors de toute fonction.

- **Durée de vie automatique** : ces objets sont alloués dans la pile de la mémoire avec une durée de vie correspondant à l'exécution du bloc englobant. Les objets automatiques sont ceux créés par une déclaration dans une fonction ou dans un bloc.

- **Durée de vie explicite** : ces objets sont créés dynamiquement par une requête explicite (en utilisant malloc et new). Leurs destruction est à la charge de programmeur.

## Notions de base

### Les classes

#### Objet et instance d'une classe

##### Exemple :

```
Point M ; //objet global (durée statique)
void main(){
    Point A, B ; //objet automatique ;
    Point *P= new point ;//création dans la pile explicite
    static Point C ; //objet statique
    //....
    delete p ; //destruction explicite
}
```

## Notions de base

### Les classes

#### Les fonctions membres non statiques d'une classe : Fonctions d'objets

Les fonctions d'objets sont les fonctions membres d'une classe qui ne sont pas déclarées statiques.

#### Déclaration d'une fonction membre

La déclaration d'une fonction d'objet est identique à la définition usuelle, cependant, déclarée dans le bloc de déclaration de la classe, elle est liée à cette classe particulière.

## Notions de base

### Les classes

#### Déclaration d'une fonction membre: Exemple :

```
class Compte{  
    int montant;  
    public:  
    void init (int); // fonction d'objet  
    int solde(); // fonction d'objet  
    void depot (int); // fonction d'objet  
};  
void affiche (Compte); //fonction usuelle (non membre)
```

fonctions membre

## Notions de base

### Les classes

#### Déclaration d'une fonction membre

- ❑ La différence entre une fonction d'objet et une fonction usuelle est que la fonction membre possède un argument implicite qui est l'objet sur lequel elle est appliquée.
- ❑ La fonction sera appliquée à un objet, instance de classe.
- ❑ L'utilisation de ces fonctions se fait avec la notation classique « . » ou « → » suivant que la fonction est appliquée à un objet ou à un pointeur sur un objet.

#### Exemple :

```
void main(){  
    Compte c1 ;  
    Compte *p=&c1 ;  
    c1.initialise(30) ;  
    p→depot (500) ;// c1.depot(500) ;  
};
```

## Notions de base

### Les classes

#### Définition d'une fonction membre:

La définition d'une fonction d'objet **peut se faire** en dehors du bloc de définition de son classe d'appartenance. Lors de la définition d'une fonction d'objet, son nom est préfixé par le nom de la classe d'appartenance suivie de l'opérateur de résolution de portée ::

#### Exemple :

```
void Compte :: initialise(int m )
{
    montant =m ;
}
void Point :: affiche()
{
    cout<< «"ce point est de coordonnées "<<x<< "et" <<y <<endl ;
}
```

## Notions de base

### Les classes

#### fonction inline

Le mécanisme de fonctions en ligne permet de supprimer le coût de l'appel d'une fonction en substitution le code de cette fonction au niveau de l'instruction d'appel de cette fonction. De même une fonction membre pourra être définie en ligne.

#### Exemple :

Soit d'une façon <b>explicite</b> à l'aide du mot-clé <b>inline</b>	Soit d'une façon <b>implicite</b> dans le corps de la définition de la classe
<pre>class Compte{     int montant; public:     int solde(); }; inline int Compte::solde(){     return montant; } }</pre>	<pre>class Compte{     int montant; public :     int solde()     {         return montant;     } };</pre>

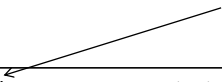
## Notions de base

### Les classes

#### fonction inline

En C :

macro



```
#define CARRE(x) x*x
int main()
{
    long a=CARRE(2+4);
    printf("%ld",a);
}
```

Le programme affiche : **36**

## Notions de base

### Les classes

#### fonction inline

En C++ :

```
inline long carre(long);
int main()
{
    cout << carre(2+4) << endl;
}

long carre(long t)
{
    return t*t;
}
```

Le programme affiche **36**

## Notions de base

### Les classes

#### Les données et fonctions membres statiques d'une classe

❑ L'emploi du mot clé **static** dans les classe intervient pour caractériser les données membres statiques des classes, **les données et les fonctions membres statiques** d'une classe.

##### ▪Données membres statiques

- Une donnée membre statique (variable de classe) est une variable globale de la classe. Elle n'est pas un composant d'objet.
- La définition d'une variable de classe est utilisée pour conserver une information sur l'ensemble des instances de la classe.

## Notions de base

### Les classes

#### Les données et fonctions membres statiques d'une classe : exemple

```
class Point{  
  //x et y sont 2 données membre privées non statique  
  int x, y ;  
  //nb est une donnée membre statique  
  static int nb ;  
};  
//initialisation explicite int Point :: nb=0 ;
```

- ❑ La variable Point :: nb sera partagée par tous les objets de classe Point, et sa valeur initiale est zéro.
- ❑ Donnée non statique : liée à un objet
- ❑ Donnée statique : liée à la classe (fournie une information sur la classe)
- ❑ La définition des données membres statiques suit les mêmes règles que la définition des variables globales. Autrement dit, elles se comportent comme des variables déclarées externes.



## Notions de base

### Les classes

#### Les données et fonctions membres statiques d'une classe

❑ Les variables statiques des fonctions membres doivent être initialisées à l'intérieur des fonctions membres. Elles appartiennent également à la classe, et non pas aux objets. De plus, leur portée est réduite à celle du bloc dans lequel elles ont été déclarées.

#### Exemple :

```
#include <iostream>
using namespace std;
class Test
{ public: static int i;    // Déclaration dans la classe.
};
int Test::i=0;           // Initialisation en dehors de la classe.
void main()
{ Test t1,t2;
  t1.i++;
  t2.i++;
  cout <<"le nombre d'incréméntation de i est:"<<Test::i<<endl;
}
```

Affichera 2, parce que la variable statique *i* est un (attribut en commun) pour les deux objets t1 et t2

## Notions de base

### Les classes

#### Les données et fonctions membres statiques d'une classe : exemple

❑ Les classes peuvent également contenir des fonctions membres statiques. Cela peut surprendre à première vue, puisque les fonctions membres appartiennent déjà à la classe, c'est-à-dire à tous les objets.

❑ Les fonctions membres ne recevront pas le pointeur sur l'objet *this*, comme c'est le cas pour les autres fonctions membres. Par conséquent, elles ne pourront accéder qu'aux données statiques de l'objet.

#### Exemple :

```
class Entier {
  int i; static int j;
  public: static int get_value(void);
};
int Entier::j=0;
int Entier::get_value(void){
  j=1;    // Légal.
  return i; // ERREUR ! get_value ne peut pas accéder à i.
}
```

## Notions de base

### Les classes

#### Les données et fonctions membres statiques d'une classe : exemple

- ❑ La fonction **get\_value** de l'exemple précédent ne peut pas accéder à la donnée membre non statique *i*, parce qu'elle ne travaille sur aucun objet. Son champ d'action est uniquement la classe Entier. En revanche, elle peut modifier la variable statique *j*, puisque celle-ci appartient à la classe Entier et non aux objets de cette classe.
- ❑ L'appel des fonctions membre statiques se fait exactement comme celui des fonctions membres non statiques, en spécifiant l'identificateur d'un des objets de la classe et le nom de la fonction membre, séparés par un point.
- ❑ Vue que les fonctions membres ne travaillent pas sur les objets des classes mais plutôt sur les classes elles-mêmes, la présence de l'objet lors de l'appel est facultatif. On peut donc se contenter d'appeler une fonction statique en qualifiant son nom du nom de la classe à laquelle elle appartient à l'aide de l'opérateur de résolution de portée ::

## Notions de base

### Les classes

#### Les données et fonctions membres statiques d'une classe

##### Exemple 1 :

```
class Entier {
    static int i;
    public: static int get_value(void);
};
int Entier::i=3;
int Entier::get_value(void){
    return i;
}
void main(){
    // Appelle la fonction statique get_value
    int resultat;
    resultat = Entier::get_value();
}
```

## Notions de base

### Les classes

#### Les données et fonctions membres statiques d'une classe

Exemple 2:

```
#include <iostream>
using namespace std;
class Point {
    static int nb;// pour compter les instances d'une classe
    // indéfinie
public:
    static int count() {return ++nb ;}
};
int Point::nb=0;
void main()
{ int n1=Point::count();
  int n2=Point::count();
  int n3=Point::count();
  cout <<< " le nombre d'appel est:" <<n3 << endl;
}
```

Affichera 3, parce que la variable statique *compte* est la **même (attribut en commun)** pour les deux objets.

## Notions de base

### Les classes

#### Fonctions et classes amies

- ❑ Il est parfois nécessaire d'avoir des fonctions qui ont un accès illimité aux champs d'une classe. En général, l'emploi de telles fonctions traduit un manque d'analyse dans la hiérarchie des classes, mais pas toujours. Elles restent donc nécessaires malgré tout.
- ❑ De telles fonctions sont appelées des fonctions amies. Pour qu'une fonction soit amie d'une classe, il faut qu'elle soit déclarée dans la classe avec le mot clé **friend**.
- ❑ Il est également possible de faire une classe amie d'une autre classe, mais dans ce cas, cette classe devrait peut-être être une classe fille. L'utilisation des classes amies peut traduire un défaut de conception.

## Notions de base

### Les classes

#### Fonctions et classes amies

- ❑ 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.
- ❑ Les fonctions amies ne sont pas des méthodes de la classe cependant (cela n'aurait pas de sens puisque les méthodes ont déjà accès aux membres de la classe).

## Notions de base

### Les classes

#### Fonctions et classes amies

##### Exemple :

```
class Point {  
    int x,y ; // Une donnée privée.  
  
    friend void init(int, int); // La fonction init est amie.  
};  
  
Point A;  
  
void init(int abs, int ord){  
    A.x=abs ; // accède à des données privées à l'objet A  
    A.y=ord ;  
}
```

## Notions de base

### Les classes

#### Fonctions et classes amies

##### Exemple :

```
#include<stdio.h>
```

```
class Hote
{
    friend class Amie; /*Toutes les méthodes
                        de Amie sont amies.*/

    int i; // Donnée privée de la classe Hote.

public:
    Hote(void)
    {
        i=0;
        return ;
    }
};
```

```
Hote h;
class Amie
{
public:
    void print_hote(void)
    {
        printf("%d\n", h.i); /* Accède à
                               la donnée privée de h */
    }
};

int main(void)
{
    Amie a;
    a.print_hote();
    return 0;
}
```

## Notions de base

### Les classes

#### Fonctions et classes amies

- ❑ L'amitié n'est pas transitive. Cela signifie que les amis des amis ne sont pas des amis. Une classe A amie d'une classe B, elle-même amie d'une classe C, n'est pas amie de la classe C par défaut. Il faut la déclarer amie explicitement si on désire qu'elle le soit.
- ❑ Les amis ne sont pas hérités. Ainsi, si une classe A est amie d'une classe B et que la classe C est une classe fille de la classe B, alors A n'est pas amie de la classe C par défaut. Encore une fois, il faut la déclarer amie explicitement.
- ❑ Ces remarques s'appliquent également aux fonctions amies (une fonction amie d'une classe A amie d'une classe B n'est pas amie de la classe B, ni des classes dérivées de A)

## Notions de base

### Les classes

#### Constructeur et Destructeur :utilisation

- juste après une création d'objet. Il porte le même nom que la classe, et n'a pas de type de retour.
- Un constructeur **sert à initialiser** l'objet au moment de sa création
- Le destructeur est appelé lorsque **l'objet est supprimé**

75

## Notions de base

### Les classes

#### Constructeur et Destructeur

- ☐ En général, il est nécessaire de faire appel à une méthode pour initialiser les valeurs des attributs (ex : **creation(int, string);**)
- ☐ Le constructeur permet de traiter ce problème à la déclaration de l'objet
- ☐ Un constructeur est une méthode spéciale qui est appelée automatiquement

76

## Notions de base

### Les classes

#### Constructeur

- ❑ Méthode appelée automatiquement à chaque création d'objet
- ❑ Le constructeur porte le même nom que la classe

#### Exemple :

```
class Compte
{
    int solde;
    int numero;
    string proprietaire;

    public :
        Compte(int, string);
        void depot(int);
        void retrait(int);
};

Compte::Compte(int num, string prop)
{
    numero = num;
    proprietaire=prop;
    solde = 0;
}

main(void)
{
    Compte c1(123, « Falten");
}
```

77

## Notions de base

### Les classes

#### Constructeur et Destructeur

- ❑ À partir du moment où une classe possède un constructeur, il n'est plus possible de créer un objet sans fournir les arguments requis par son constructeur
  - La déclaration **Compte c1;** ne convient plus
  - L'utilisateur est obligé de donner des valeurs d'initialisation

L'utilisation de constructeur est fortement recommandée

78

## Notions de base

### Les classes

#### Constructeur et Destructeur

Fichier Point.h

```
class Point
{
    double x, y;
    int numero;

public:
    Point(int n);
    Point(double xx, double yy);
};
```

Il peut y avoir plusieurs constructeurs (surcharge de fonction).

Fichier Point.cpp

```
Point::Point (int n)
{
    x = y = 0.0;
    numero = n;
}

Point::Point (double xx, double yy)
{
    x = xx;
    y = yy;
    numero = 0;
}
```

## Notions de base

### Les classes

#### Constructeur et Destructeur

Lors de la définition de l'objet, les arguments du constructeur sont passés entre parenthèses après le nom de l'objet :

```
Point p1(10);
Point p2(1.0, 2.0);
```

On peut aussi utiliser le signe = suivi du nom du constructeur :

```
Point p1 = Point(10);
Point p2 = Point(1.0, 2.0);
```

Dans le cas où une seule valeur d'initialisation est mentionnée, il est possible d'omettre le nom du constructeur (constructeur de transtypage) :

```
Point p1 = 10;
```

Pour la création d'objets dynamiques, le constructeur est précédé du mot-clé **new** :

```
Point *p1 = new Point(10);
Point *p2 = new Point(1.0, 2.0);
```



## Notions de base

### Les classes

#### Constructeur et Destructeur

##### Paramètres par défaut et constructeur sans argument

Fichier Point.h

```
class Point
{
    double x, y;
    int numero;

public:
    Point(double xx = 0.0,
          double yy = 0.0,
          int n = 0);
};
```

Il peut y avoir des paramètres par défaut dans un constructeur.

```
#include "Point.h"

{
    Point p1(1.0, 2.0, 10); // 3 args
    Point p2(1.0, 2.0);    // 2 args
    Point p3(1.0);         // 1 arg

    Point p4;              // 0 arg

    Point p5();             // ERREUR ? C'est
                          // la déclaration
                          // d'une fonction
                          // qui retourne
                          // un 'Point'

    Point P6 = Point();    // 0 arg
}
```

## Notions de base

### Les classes

#### Constructeur et Destructeur

- ❑ Les constructeurs sont soumis aux contraintes de droits d'accès spécifiées par `private`, `protected` ou `public`.
- ❑ Pour permettre à des sections de programmes de créer des objets, on rend le constructeur correspondant *public*.
- ❑ Mais parfois, on souhaite justement qu'aucun objet de cette classe ne soit créé (classe abstraite) et on protège ou on privatise le constructeur.

```
class Point
{
    private:
        double x, y;
        int numero;
        Point(); // => Interdiction de l'utiliser

public:
    Point(double xx, double yy, int n);
};
```

## Notions de base

### Les classes

#### Constructeur et Destructeur

❑ Un constructeur par défaut est un constructeur que l'on peut utiliser sans lui passer de paramètres : il peut s'agir de constructeurs sans paramètre formel, ou de constructeurs qui ne comportent que des valeurs par défaut.

```
Point();
```

```
Point(double xx = 0.0,  
      double yy = 0.0,  
      int n = 0);
```

❑ Mais les règles sur les surcharges de noms de fonctions empêchent que plusieurs constructeurs par défaut soit présents : **une classe ne doit comporter qu'un seul constructeur par défaut.**

❑ **Si pour une classe on ne définit pas de constructeur (du tout), le compilateur en créera un, par défaut, public, pour la construction d'objets.**

❑ Ce constructeur ne réalise aucune initialisation et laisse les attributs de l'objet créé dans un état indéfini.

## Notions de base

### Les classes

#### Constructeur et Destructeur

```
#include <iostream>
using namespace std;
class Point {
public:double x, y;
int numero;};
int main() {
Point p; // ca marche
// warning local variable 'p' used without having been initialized
cout << p.numero << " ---- " << p.x << " ---- " << p.y << endl;
// Affiche : 0 - 1.29702e-312 - 2.07372e-317
return 1;
}
```

## Notions de base

### Les classes

#### Destructeur

- ❑ Méthode **appelée automatiquement** au moment de la destruction de l'objet (fin de la fonction, boucle... où l'objet a été créé)
- ❑ Porte le même nom que la classe précédé d'un tilde (~)
- ❑ Ne dispose **d'aucun argument** et ne renvoie pas de valeur
- ❑ Un destructeur devient **indispensable** lorsque **l'objet alloue de la mémoire dynamiquement**

85

## Notions de base

### Les classes

#### Destructeur

```
class Point
{
    double x, y;
    int numero;

public:
    Point(int n);
    Point(double xx, double yy);
    ~Point();
};
```

Grâce à la surcharge des noms de fonctions, il peut y avoir plusieurs Constructeurs, mais il ne peut y avoir qu'un seul destructeur.

86

# Notions de base

## Les classes

### Méthodes « constantes »

- ❑ Parmi les méthodes d'une classe, il est possible de distinguer :
  - celles qui modifient les attributs d'un objet ;
  - celles qui ne modifient aucun attribut ("*méthodes constantes*").
- ❑ Les méthodes "constantes" ne font pas d'accès "en écriture". Le mot réservé **const** doit figurer dans leur déclaration et dans leur définition, après la liste des arguments :

Fichier Point.h

```
class Point
{
    double x, y;
    int numero;

public:
    void setNumero (int n);
    double norm() const;
};
```

Fichier Point.cpp

```
void Point::setNumero (int n)
{
    numero = n;
}

double Point::norm() const
{
    return sqrt (x*x + y*y);
}
```

# Notions de base

## Les classes

### Méthodes « constantes »

Fichier Point.h

```
class Point
{
    double x, y;
    int numero;

public:
    void setNumero (int n);
    double norm() const;
};
```

Fichier Point.cpp

```
void Point::setNumero (int n)
{
    numero = n;
}

double Point::norm() const
{
    return sqrt (x*x + y*y);
}
```

Important : « const » fait partie de la signature

## Notions de base

### Les classes

#### Méthodes « constantes »

Donner un accès protected ou private aux attributs, écrire des méthodes accesseurs, utiliser les fonctions const sont les moyens à utiliser pour protéger les données d'un objet (**encapsulation**).

Fichier Point.h

```
class Point
{
private:
    double x, y;
    int numero;
    double norm() const;
    double diff();

public:
    void setNumero (int n);
    void show() const;
};
```

S'il y a une erreur dans les données x, y ou numero, les seules méthodes à mettre en cause sont les méthodes non constantes (diff et setNumero).

## Notions de base

### Les tableaux

#### Les tableaux statique

- ❑ Pour les tableaux dont la taille n'est connue que durant l'exécution, il faut utiliser l'**allocation dynamique**
- ❑ Ce mode de déclaration de tableaux permet de déclarer des tableaux dont la taille est connue **à la compilation**:

```
int n = 10;
double tab3[n]; // error : expected constant expression
```

On peut **initialiser** les valeurs d'un tableau avec la syntaxe suivante :

```
int t1[] = {1, 2, 3}; /* dim. implicite : 3 */
double t2[5] = {4.0, 5.0}; /* dimension 5, mais
                           initialisation de
                           t2[0] et t2[1] */

int t3[2][3] = {10, 20, 30, 40, 50, 60};
int t4[2][3] = {{10, 20, 30}, {40, 50, 60}};
```

90


## Notions de base

### Les tableaux

#### Les tableaux statique

- ❑ Les chaînes de caractères sont des tableaux de caractères à une dimension.
- ❑ Une chaîne se termine au premier caractère "null" ('\0') rencontré. Il n'y a donc pas de limite théorique de longueur.
- ❑ Le "null" est automatiquement ajouté pour les constantes, dans les autres cas, il faut le gérer *à la main*.
- ❑ La taille déclarée d'une chaîne de caractères doit toujours être supérieure de 1 à ce que l'on désire y stocker.

```
char chaine[100];  
  
char texte[] = "Hello\n"; // Tableau de 7 caractères  
  
char s[5];  
s[0] = 'a';  
s[1] = 'b';  
s[2] = 'c';  
s[3] = '\0';
```



91

## Notions de base

### Les tableaux

#### Les tableaux dynamique

Désallocation mémoire

C	<pre>int* var = (int*)malloc(sizeof(int)); int* array = (int*)malloc(n*sizeof(int)); ... free( var ); free( array );</pre>
C++	<pre>int* var = new int(); int* array = new int[10]; ... delete var; delete []array;</pre>

92

## Notions de base

### Les tableaux

#### Les tableaux dynamique

Allocation mémoire

C	<pre>int value = 10; int* var = (int*)malloc(sizeof(int)); *var = value; int* array = (int*)malloc(n*sizeof(int));</pre>
C++	<pre>int value = 10; int* var = new int( value ); int* array = new int[10];</pre>

93

## Notions de base

### Les tableaux

#### Pointeur et tableaux

Dans les deux déclarations : `int a1[10];`  
`int *a2;`

a1 et a2 sont du même type, pointeur sur un entier. Mais il y a une différence fondamentale :

a1 est un pointeur **constant**. La déclaration réserve un emplacement mémoire pour 10 entiers, a1 pointe réellement sur quelque chose et sa valeur ne peut être modifiée :

a2 est un pointeur **variable**. Aucun emplacement mémoire n'est réservé pour des variables de type int, la valeur de a2 est indéfinie à l'origine, il est donc prudent de l'initialiser à NULL.

```
int a1[10];
int *a2;
a2 = NULL; /* c'est prudent */
...
a2 = a1; /* a2 pointe sur les éléments de a1 */
a1 = a2; /* ILLEGAL, a1 est constant */
```

94

## Notions de base

### Les tableaux

#### Pointeur et tableaux

A l'usage, les deux déclarations permettent de manipuler le tableau de valeurs de manières identiques :

<pre>void reset (double t[],             int n) {     int i;     for (i=0; i&lt;n; i++)         t[i]=0.0; }</pre>	<pre>void reset (double *t,             int n) {     int i;     for (i=0; i&lt;n; i++)         t[i]=0.0; }</pre>
---	--

---

```
int main(void)
{
    double tab[100];
    reset(tab, 100);
    ...
    reset(tab, 50);
    ...
    return 0; /* TOUT A ETE OK */
}
```

95

## Notions de base

### Les tableaux

#### Objets et tableaux

Comme tous les types, il est possible de définir des tableaux d'objets

Si des constructeurs ont été définis

- C'est le constructeur sans arguments qui est appelé
  - il faut donc qu'il existe !
- Il est possible de faire appel à un constructeur particulier, mais dans ce cas, il faut le préciser pour tout les éléments !

96



## Notions de base

### Les classes

#### Objets et tableaux : exemple

```
class maClasse
{
    string nom;
public :
    maClasse();
    maClasse(string);
    void affiche();
};
maClasse::maClasse()
{
    nom="inconnue";
}
maClasse::maClasse(string s)
{
    nom = s;
}
```

```
void maClasse::affiche()
```

```
{
    cout << nom;
}
main()
{
    maClasse tab1[5];
    maClasse tab2[2]={
        maClasse("c0"),
        maClasse("c1") };
    for (int i=0; i<5; i++)
        tab1[i].affiche();
    for (int i=0; i<2; i++)
        tab2[i].affiche();
}
```

```
inconnue
inconnue
inconnue
inconnue
inconnue
c0
c1
```

## Notions de base

### Les classes

#### Tableaux d'objet

❑ Quand on crée un tableau d'objets, un appel de constructeur est effectué pour la construction et l'initialisation de chaque élément individuel.

❑ Rien n'oblige à ce que le même constructeur soit appelé pour tous les éléments.

Fichier Point.h

```
class Point
{
private:
    double x, y;
    int numero;
public:
    Point();
    Point(int n);
    Point(double xx,
           double yy);
    ~Point();
};
```

Fichier Main.cpp

```
#include "Point.h"

// Appels de constructeurs par défaut :
Point tab1[3];
Point tab2[3] = {Point(), Point(), Point()};
Point *tab3 = new Point[3];

// Appels hétérogènes :
Point tab4[3] = {10, Point(1.0, 2.0), Point()};
```

## Notions de base

### Les tableaux

#### pointeurs et chaînes de caractères

On peut déclarer une chaîne de caractères de deux façons différentes :

```
/* Tableaux de  
caractères */
```

```
char chaine1[100];  
char chaine2[] = "Hello\n";
```

chaine1 et chaine2 sont des pointeurs **constants**, on peut modifier le contenu du tableau mais pas son adresse.

```
/* ILLEGAL */  
chaine2 = chaine1;
```

```
/* Pointeur vers UN  
caractère */
```

```
char *chaine3;
```

chaine3 est un pointeur **variable**, on peut le diriger vers le début d'une éventuelle chaîne de caractères.

```
/* LEGAL */  
chaine3 = chaine1;
```

99

## Notions de base

### Les tableaux

#### pointeurs et chaînes de caractères

Mais pour accéder aux valeurs, tout se passe comme s'il s'agissait de tableaux de caractères :

```
char chaine1[100];  
char chaine2[] = "Hello\n";  
char *chaine3;
```

```
chaine3 = chaine2;
```

```
chaine1[0] = ...;
```

```
... = chaine2[4]; /* Le caractère 'o' */  
... = chaine2[5]; /* Le caractère '\n' */  
... = chaine2[6]; /* Le caractère '\0' */
```

```
... = chaine3[4]; /* Le caractère 'o' */  
... = chaine3[5]; /* Le caractère '\n' */  
... = chaine3[6]; /* Le caractère '\0' */
```

100

## Notions de base

### Définition des structures

En C, les structures (mots-clés `struct`) permettent de créer de nouveaux types, complexes, en assemblant d'autres types :

```
struct point
{
    double x, y;
    int numero;
};
```

Définit un nouveau type  
« struct point »  
avec trois champs

On accède aux différents champs en utilisant une notation « pointée » :

```
struct point p1 = {0.0, 0.0, 1}, P[100];
point p2;
p2.numero = ... ;
p1.x      = ... ;
...       = P[50].x ;
```

En C++, le « struct »  
est facultatif

101

## Notions de base

### Définition des structures

```
struct point
{
    double x, y;
    int numero;
};

typedef struct point Point;

Point p3, tabp[100], *pp;

p3.numero = ... ;
...       = tabp[50].x ;
pp        = &tabp[30];
pp->x     = 10.0;
...       = pp->numero;
(&p3)->x  = 20.0;
...       = (&p3)->x;
```

Lorsque la structure est  
référéncée par un pointeur, on  
utilise une notation « fléchée »  
pour accéder aux champs à  
partir du pointeur

102

## Exercice 1

Écrire un programme C++ qui permet le remplissage d'un tableau **Tab** de **N** entiers ( $N \leq 10$ ). Calculer puis afficher la somme de ses éléments.

## Solution

```
#include <iostream.h>
void main()
{
    int Tab[10], n, i, s;

    do{
        cout<<"DONNER N=";
        cin>>n;
    } while((n<=0)||(n>10));
    for(i=0; i<n;i++)
    {
        cout<<"Tab["<<i<<"]= ";
        cin>> Tab[i];
    }
    for(i=0, s=0; i<n;i++)
        s+=Tab[i];
    cout<<"la somme des éléments du tableau Tab est = "<<s<< endl;
}
```

## Exercice 2

Créer une classe point ne contenant qu'un constructeur sans arguments, un destructeur et un membre donnée privé représentant un numéro de point (le premier créé portera le numéro 1, le suivant le numéro 2, et ainsi de suite ...).

Le constructeur affichera le numéro du point créé et le destructeur affichera le numéro du point détruit.

Ecrire un petit programme d'utilisation créant dynamiquement un tableau de 4 points et le détruisant.

## Solution 2

```
#include <iostream.h>
class point{
    int static nombre;
public:
    point();
    ~point();
};
int point::nombre=0;
point::point(){
    nombre++;
    cout<<"point n ° "<<nombre<<" construit"<<endl;
}
point::~~point(){
    nombre--;
    cout<<"point n ° "<<nombre<<" détruit"<<endl;
}
```

```
int main(){
    point *tableau=new point[4];
    delete [] tableau;
    return 0;
}
```

## Exercice 3

On se propose de définir la classe Etudiant. Cette classe est caractérisée par : Numéro d'inscription, nom, prénom, moyenne du premier semestre, moyenne du second semestre, et la moyenne générale. On vous demande de :

- 1) Définir la classe Etudiant.
- 2) Développer les fonctions suivantes :
  - Saisie\_etudiant
  - Afficher\_etudiant
  - Calcul\_moy\_gen\_etudiant
- 3) Utiliser cette classe dans une fonction main en :
  - Créant un tableau de 10 étudiants représentant les étudiants d'un groupe
  - Calculant la moyenne générale du groupe.

## Solution 3

```
1) #include<iostream.h>
   class Etudiant {
       private:
           int Nins;
           char Nom[10];
           char Prenom[10];
           float moys1;
           float moys2;
           float moy_gen;
       public:
           void saisie_etudiant();
           void affichage_etudiant();
           float clacul_moy_gen_etudiant() ;
       };

```

## Solution 3

```
2) void etudiant::saisie_etudiant()
{ cout<< "entrer le numéro d'inscription d'etudiant" ;
  cin>>Nins ;
  cout<<"entrer le nom d'etudiant" ;
  cin>>Nom ;
  cout<<"entrer le prénom d'etudiant" ;
  cin>>Prénom ;
  cout<<"entrez la moyenne du 1er semestre d'etudiant" ;
  cin>>moys1 ;
  cout<<"entrez la moyenne du 2ème semestre d'etudiant" ;
  cin>>moys2 ;
}
```

## Solution 3

```
2) //suite de la correction
void etudiant::affiche_etudiant()
{
  cout<<"Le numéro d'inscription d'etudiant"<<Nins ;
  cout<<"Le nom d'étudiant"<<Nom ;
  cout<<"La moyenne du 1er semestre"<<moys1;
  cout<<"La moyenne du 2ème semestre"<<moys2;
  cout<<"La moyenne générale d'étudiant"<<moy_gen ;
}
float Entier::clacul_moy_gen_etudiant()
{
  Moy_gen=(moys1+moys2)/2 ;
  return moy_gen ;
}
```

## Solution 3

```
3) void main()
{
    Etudiant A[10];
    int i;
    float Mg, S=0;
    for(i=0; i<10; i++)
        A[i].saisie_etudiant();
    for(i=0; i<10; i++)
    {
        S=S+A[i].calcul_moy_gen_etudiant();
        A[i].affiche_etudiant();
    }
}
```

## Notions de base

### Classe

#### L'héritage simple

- ❑ L'héritage est un concept de base de la programmation par objets dont un but est la réutilisabilité du logiciel. En particulier, l'héritage permet de dériver une nouvelle classe d'objets à partir d'une classe déjà existante.
- ❑ Une classe peut être définie comme une classe dérivée d'une autre. La classe dérivée hérite alors automatiquement des définitions de la classe de base. L'intérêt de l'héritage est :
  - 1) obtenir la même interface pour des classes similaires.
  - 2) ne pas redéfinir des membres qui peuvent être héritées.



## Notions de base

### Classe

#### L'héritage simple

La déclaration d'héritage se fait dans l'entête de la définition de la classe dérivée.

Exemple :

```
class Base
{
    //déclarations propres à Base
    ...
};
class Dérivée : public Base
{
    // déclarations propres à la classe dérivée
    ...
};
```

## Notions de base

### Classe : notion d'héritage

- ☐ Plusieurs classes peuvent être dérivées d'une même classe de base
- ☐ Une classe dérivée peut devenir à son tour classe de base d'une autre classe
- ☐ L'héritage multiple est autorisé par le C++, mais reste controversé (et même déconseillé)
- ☐ Déclaration :

```
class classeDerivee : public classeDeBase
{
    ...
};
```

## Notions de base

### Classe : notion d'héritage

#### Déclaration d'une classe de base

```
class Point
{
    int x;
    int y;

public:
    void init(int, int);
    void deplace(int, int);
    void affiche();
};
```

## Notions de base

### Classe : notion d'héritage

#### Déclaration d'une classe dérivée

```
class PointColore : public Point
{
    string couleur;

public:
    void colore(string c){couleur =
    c;}
};
```

## Notions de base

### Classe : notion d'héritage

#### Conséquences

❑ Le mot clé **public** signifie que les attributs et méthodes publics de la classe de base resteront publics (notion classique de l'héritage).

- Précisé plus tard

Chaque objet de type **PointCouleur** peut alors avoir accès :

- Aux attributs et méthodes publics de **PointCouleur**
- Aux attributs et méthodes publics de **Point**

## Notions de base

### Classe : notion d'héritage

#### Exemple de programme

```
main ()
{
    PointCouleur p;
    p.init(10, 10);
    p.couleur("bleu");
    p.affiche();
    p.deplace(15, 30);
    p.affiche();
}
```

## Notions de base

### Classe : notion d'héritage

#### Utilisation des attributs et méthodes de la classe de base

- ❑ Une classe dérivée n'a pas accès aux attributs et méthodes privés de la classe de base
  - Ex : impossible de définir une méthode dans la classe **pointCouleur** qui affiche directement les valeurs des attributs x et y
- ❑ Une classe dérivée (public) a accès aux attributs et méthodes publiques
  - Ex : ajouter une méthode d'affichage à la classe **pointCouleur**

## Notions de base

### Classe : notion d'héritage

#### Exemple :

```
class PointCouleur : public Point
{
    string couleur;
public:
    void colore(string c){couleur = c;}
    void afficheC();
};
void pointCouleur::afficheC()
{
    affiche();
    cout << "ma couleur est : " << couleur;
}
```

## Notions de base

### Classe : notion d'héritage

#### Redéfinition de méthode

- ❑ Une méthode d'une classe de base peut être redéfinie dans une classe dérivée

```
class PointColore : public Point
{
    string couleur;
public:
    void colore(string c){couleur = c;}
    void affiche();
};
void pointColore::affiche ()
{
    Point::affiche();
    cout << "ma couleur est : " << couleur;
}
```

## Notions de base

### Classe : notion d'héritage

#### Constructeurs et destructeurs

- ❑ Les constructeurs et destructeurs ne sont pas hérités
- ❑ Pour créer un objet de type dérivé :
  - ❑ Appel au constructeur de la classe de base
  - ❑ Appel au constructeur de la classe dérivée
- ❑ Lors de la destruction d'un objet de type dérivé :
  - ❑ Appel du destructeur de la classe dérivée
  - ❑ Appel du destructeur de la classe de base

## Notions de base

### Classe : notion d'héritage

#### Transmission d'information

- ❑ Si le constructeur de la classe de base nécessite des arguments, il faut lui transmettre à partir du constructeur de la classe dérivée
- ❑ Il faut alors utiliser une syntaxe précise :
  - On appelle le constructeur de la classe mère

## Notions de base

### Classe : notion d'héritage

#### Exemple classe de base

```
class Point
{
    int x;
    int y;

public:
    Point(int, int);
    void deplace(int, int);
    void affiche();
};
```

## Notions de base

### Classe : notion d'héritage

#### Exemple classe dérivée

```
class PointColore : public Point
{
    string couleur;
public:
    PointColore(int, int, string);
    void colore(string c){couleur = c;}
    void afficheC();
};

PointColore::PointColore(int a, int o, string c):Point(a, o)
{
    couleur = c;
}
```

## Notions de base

### Classe : notion d'héritage

#### L'héritage simple

##### Protection et héritage

class Base

{ **private:** /\* membres privés : les membres privés peuvent être référencés uniquement par les membres ou les amies de la classe\*/

**protected:** /\* les membres protégés peuvent être référencés par les membres (et les amies) de la classe et également par les membres d'une classe dérivée. \*/

**public:** /\* les membres publiques peuvent être référencés par n'importe quelle fonction \*/

} ;

## Notions de base

### Classe : notion d'héritage

Exemple:

```
class A
{
    string s1;
    protected string s2;
    ...
};
class B : public A
{
    void affiche();
    ...
};
B::affiche()
{
    cout << s1;
    cout << s2;
}
```

## Notions de base

### Classe : notion d'héritage

#### Dérivation privée

- ❑ C'est la dérivation par défaut. Les membres protégés et publics de la classe de base sont privés dans la classe dérivée.
- ❑ Il est possible d'interdire à un utilisateur d'une classe dérivée l'accès aux membres publics de sa classe de base

```
class classeDerivee : private classeDeBase
{
    ...
}
```



## Notions de base

### Classe : notion d'héritage

#### Exemple:

```
class Point
{...
public:
    Point(int, int);
    void deplace(int, int);
    void affiche();
};
class PointColore : private Point
{...
public:
    PointColore(int,int,string)
    void colore(string);
};
main()
{ pointColore p (10, 10, "bleu");
  p.deplace(15, 30);
  p.affiche();
}
```

## Notions de base

### Classe : notion d'héritage

#### Dérivation protégée

- ☐ Cette dérivation consiste à protéger les membres hérités.
- ☐ Les membres protégés et publics de la classe de base sont protégés dans la classe dérivée

## Notions de base

### Classe : notion d'héritage

#### L'héritage simple

##### Dérivation et interface de classe

- L'interface obtenue pour la classe dérivée en fonction du type de la dérivation peut se résumer par le schéma suivant :

```
class Base
{ private:
    void f ();
    protected:
    void g ();
    public:
    void h ();
};
```

- Le membre *Base::f* étant privé dans la classe de base, il ne pourra pas être référencé au niveau de la classe dérivée, quelque soit le type de dérivation.

## Notions de base

### Classe : notion d'héritage

#### Dérivation et contrôle d'accès

##### Dérivation publique:

- Conservation du statut des membres publics et protégés de la classe de base dans la classe dérivée
- Forme la plus courante d'héritage modélisant : « une classe dérivée est une spécialisation de la classe de base »

```
class Base
{
    public:
        void méthodePublic1();
    protected:
        void méthodeProtégée();
    private:
        void MéthodePrivée();
};

class Derivee : public Base
{
    public:
        int MéthodePublic2()
        {
            méthodePublic1(); // OK
            méthodeProtégée(); // OK
            MéthodePrivée(); // KO
        }
};
```

## Notions de base

### Héritage multiple

- ❑ Une classe peut hériter de plusieurs classes de base. L'héritage multiple permet de dériver une nouvelle classe en lui faisant bénéficier du comportement de plusieurs classes de base à la fois.

#### Syntaxe

```
class A { ... } ;  
class B { ... } ;  
class C { ... } ;  
class D : public A, public B, public C { ... } ;
```

## Notions de base

### Héritage multiple

- ❑ Possibilité de créer des classes dérivées à partir de plusieurs classes de base
- ❑ Pour chaque classe de base: possibilité de définir le mode d'héritage
- ❑ Appel des constructeurs dans l'ordre de déclaration de l'héritage
- ❑ Appel des destructeurs dans l'ordre inverse de celui des constructeurs

## Notions de base

### Héritage multiple

```
class Point
{
    int x;
    int y;
public:
    Point(...) {...}
    ~Point() {...}
    void affiche() {...}
};

// classe dérivée de deux autres classes
class PointCouleur : public Point, public Couleur
{
    ...
    // Constructeur
    PointCouleur (...) : Point(...), Couleur(...)
    void affiche() {Point::affiche(); Couleur::affiche(); }
};


class Couleur
{
    int coul;
public:
    Couleur(...) {...}
    ~Couleur() {...}
    void affiche() {...}
};
```

## Notions de base

### Héritage multiple

```
int main()
{
    PointCouleur p(1,2,3);
    cout << endl;
    p.affiche(); // Appel de affiche() de PointCouleur
    cout << endl;
    // Appel "forcé" de affiche() de Point
    p.Point::affiche();
    cout << endl;
    // Appel "forcé" de affiche() de Couleur
    p.Couleur::affiche();
}
```

```
** Point::Point(int,int)
** Couleur::Couleur(int)
** PointCouleur::PointCouleur(int,int,int)
Coordonnées : 1 2
Couleur : 3
Coordonnées : 1 2
Couleur : 3
** PointCouleur::~PointCouleur()
** Couleur::~Couleur()
** Point::~Point()
```

 Si `affiche()` n'a pas été redéfinie dans `PointCouleur` :

error: request for member 'affiche' is ambiguous

error: candidates are:  
void Couleur::affiche()  
void Point::affiche()

## Notions de base

### Héritage multiple

```
class A
{ public:
  A(int n=0) { /* ... */ }
  // ...
};

class B
{ public:
  B(int n=0) { /* ... */ }
  // ...
};

class C: public B, public A
{ //      ^^^^^^^^^^^^^^^^^^^^^
  //      ordre d'appel des constructeurs des classes de base
  public:
    C(int i, int j) : A(i) , B(j) // Attention l'ordre ici ne
    { /* ... */                  // correspond pas à l'ordre
                                // des appels
    // ...
};

int main()
{
  C objet_c;
  // appel des constructeurs B(), A() et C()
  // ...
}
```

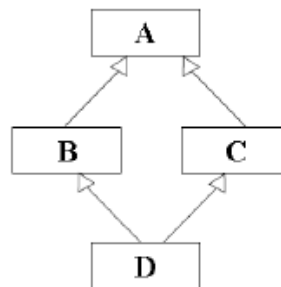
## Notions de base

### Héritage multiple

```
class A
{ int x, y;
  ....
};

class B : public A {....};
class C : public A {....};

class D : public B, public C
{ ....
};
```



- Duplication des membres données de A dans tous les objets de la classe D
- Possibilité de les distinguer les copies par `A::B::x` et `A::C::x` ou `B::x` et `C::x` (si pas de membre `x` pour B et C)
- Pour éviter la duplication : **héritage virtuel**

## Notions de base

### Héritage multiple

```
class Point{
    int x, y ;
public:
    Point (int ax, int ay) {x = ax ; y = ay ;}
    affiche ( ) {cout <<" coordonnées : " << x <<"
    "<<y <<"n"; }
};

class coul{
    int couleur ;
public:
    coul (int cl) { couleur = cl ; }
    void affiche ( )
    { cout <<"couleur : " << couleur <<"n"; }
};

class Pointcoul : public Point, public coul{
public:
    Pointcoul (int, int, int);
}

void affiche ( )
{
    Point::affiche();
    coul::affiche();
};

Pointcoul::Pointcoul (int ax, int ay, int cl) :
    Point (ax, ay), coul (cl) { }

void main ( )
{
    Pointcoul p (3, 9, 2);
    p.affiche ( ) ; // appel de affiche de Pointcoul
    p.Point::affiche ( ) ;
    p.coul::affiche ( ) ;
}
```

## Exercice 8

1) Écrivez une classe Cercle comportant :

- comme données privées trois composantes de type float : rayon, abscisse et ordonnée de l'origine du cercle
- un constructeur public défini
- une fonction amie périmètre qui calcule le périmètre d'un cercle ( $2 \cdot \text{PI} \cdot \text{R}$ ),
- une fonction amie air qui calcule la surface d'un cercle ( $\text{PI} \cdot \text{R}^2$ ),
- une fonction sortie qui affiche le rayon, l'abscisse et l'ordonnée, le périmètre et la surface d'un cercle.

2) Écrivez une fonction main utilisant des objets qui teste ces fonctions.

NB. PI doit être défini comme constante.

## Solution exercice 8

```
#include <iostream>
using namespace std;
#define PI 3.14//ou bien const float PI=3.14;
class Cercle{
float rayon, abs, ord;
public:Cercle(float r, float x, float y)
{ rayon= r;
abs=x;
ord=y; }
friend float perimetre(Cercle );
friend float surface (Cercle);
void sortie();
~Cercle();};
Cercle::~Cercle(){ }
```

## Solution exercice 8

```
Solution 8 (suite)
void Cercle::sortie(){
cout<<"le cercle est de centre ("<<abs<<","<<ord<<");
cout<<"de rayon "<<rayon;
cout<<" de surface"<<surface(*this);
cout<<"de perimetre"<<perimetre(*this);
}
float perimetre(Cercle c){
return 2*PI*c.rayon;
}
float surface(Cercle c){
return PI*c.rayon*c.rayon;
}
```

## Solution 8 (suite)

### Résultat:

le cercle 1 est de centre (1,3)  
de rayon 1  
de surface3.14  
de perimetre6.28

le cercle 2 est de centre (1,2)  
de rayon 2  
de surface12.56  
de perimetre12.56

le perimetre du cercle c1 est6.28  
le perimetre du cercle c2 est12.56

```
int main(){
Cercle C1(1,1,3);
Cercle C2= Cercle(2,1,2);

C1.sortie();
C2.sortie();

cout<<"le perimetre du cercle c1 est"<<perimetre(C1)<<endl;
cout<<"le perimetre du cercle c2 est"<<perimetre(C2)<<endl;

return 1;}
```

## Exercice 9

Soit la classe voiture ayant l'interface suivante :

```
class voiture
{ char *nom; // nom du vehicule
int annee; // annee du vehicule
public:
voiture(char*,int = 0); //constructeur
~voiture(); // destructeur
void affiche();
char* GetNom( ) ; //fonction d'accès
int GetAnnee( ) ; //fonction d'accès
};
```

- 1) Définir les fonctions membres (constructeur, destructeur, affiche, GetNom qui donne le nom de la voiture, GetAnnee qui donne l'année de la voiture) de cette classe.
- 2) Ecrire un programme principal (*main*) utilisant la classe **voiture** et testant ses différentes fonctions.



## Exercice 9

```
#include <iostream.h>
#include <string.h>
class voiture
{
    char *nom;      // nom du véhicule
    int annee;      // année du véhicule
public:
    voiture(char*,int=0);    // constructeur
    ~voiture();              // destructeur
    void affiche();
    char* getNom(); // fonction d'accès
    int getAnnee(); // fonction d'accès
};
voiture::voiture(char *n,int x)
{
    nom=new char[strlen(n)+1];
    strcpy(nom,n);
    annee=x;
    cout<<"Objet cree"<<endl;
}
```

## Solution exercice 9

```
voiture::~~voiture()
{
    delete [] nom;
    cout<<"Objet supprimee"<<endl;
}

void voiture::affiche()
{
    cout<<"nom:"<<nom<<"annee:"<<annee<<endl;
}

char* voiture ::getNom()
{
    return nom ;
}

int voiture ::getAnnee()
{
    return annee ;
}
```

## Solution exercice 9

```
void main()
{
    voiture volkswagen("beatle",1985);
    voiture usagee("jetta", 1980);
    usagee.affiche();
    cout<< volkswagen.getNom()<<endl;
    cout<< volkswagen.getAnnee()<<endl;
}
```

## Suite exercice 9

3) Soit la classe dérivée publique **voiture\_adm** de la classe **voiture** :

```
class voiture_adm : public voiture
{
    int code_m; // code du ministère
public:
    voiture_adm (char*,int, int); //constructeur
    void affiche();
    int GetCode( ) ;
};
```

Définir les fonctions membres (constructeur, affiche, GetCode qui donne le code du ministère) de cette classe.

## Solution exercice 9

```
3) class voiture_adm : public voiture
{
    int code_m; // code du ministère
public:
    voiture_adm (char*,int, int); //constructeur
    void affiche();
    int getCode( ) ;
};
voiture_adm :: voiture_adm(char* n, int x, int c) :voiture(n,x)
{
    code_m=c ;
}
void voiture_adm ::affiche()
{
    voiture ::affiche() ;
cout<< "code : " << code_m<<endl ; }
int voiture_adm ::getCode()
{
    return code_m ; }
```

## Exercice 10

On désire écrire un programme permettant de gérer les villes d'un pays.

a) Une ville est décrite par son nom et son nombre d'habitants. L'interface de classe ville est la suivante :

```
class ville
{ char* nom ; // nom de la ville
  long nbHabitants ; // nombre d'habitants
public :
  ville(char*, long=0) ; //constructeur
  ~ville() ; //destructeur
  void affiche() ;
  void modifierNbHabitants(long) ; // modifie le nombre d'habitants
  char* getNom() ; // fonction d'accès
  long getNbHabitants() ; // fonction d'accès
} ;
```

Définir les fonctions membres (constructeur, constructeur par copie, destructeur, affiche, modifierNbHabitants qui modifie le nombre d'habitants d'une ville, getNom qui donne le nom de la ville, getNbHabitants qui donne le nombre d'habitants d'une ville) de cette classe.

## Solution exercice 10

```
#include <stdio.h>
#include<iostream>
#include <string.h>
#include<assert.h>
#include<conio.h>
using namespace std;
class ville
{
    char *nom; // nom du vehicule
    int nbhabit; // annee du vehicule
public:
    ville(char*p,int a= 0) //constructeur les valeurs a initialiser
                        //sont a droite au maximum ( int a = 0)
    {
        this->nom = p ;
        nbhabit=a ;
    }
    ~ville() {} // Destructeur
```

## Solution exercice 10

```
void modifierNbHabitants(int a)
{nbhabit= a ;}
void affiche()
{
    cout<<nom<<nbhabit<<endl;
}

char* GetNom( )
{
    return nom ;
}
int GetNbHabit( )
{
    return nbhabit ;
}

};
}
```

## Solution exercice 10

```
int main()
{
    string s="Sousse";
    char *p = &s[0] ;
    ville v1 = ville(p,200) ;
    v1.affiche() ;
    v1.modifierNbHabitants(100) ;
    v1.affiche() ;
    getch() ;
    return 1 ;
}
```

## Exercice 11

Soit la classe liste suivante:

```
class liste {int taille;
    float *adr;
    public: liste(int);
    void saisie();
    void affiche();
    ~liste(); };
```

Ecrire le constructeur **liste**, le constructeur, la fonction membre **void saisie()** permettant de saisir les composantes d'une liste (utiliser le pointeur **adr**), la fonction **void affiche()** permettant de les afficher sur l'écran et le destructeur. Les mettre en oeuvre dans **void main()**.

## Solution exercise 11

```
#include <iostream.h>
class liste
{
    int taille;
    float *adr;
public:
    liste(int taille)
    {
        this->taille=taille;
        adr=new float[taille];
    }
    void saisie()
    {
        int i;
        for(i=0;i<taille;i++)
        {
            cout<<"Element n°"<<i+1<<" : ";
            cin>>adr[i];
        }
    }
}
```

## Solution exercise 11

```
void affiche()
{
    int i;
    cout<<"Votre liste est:{";
    for(i=0;i<taille;i++)
        cout<<adr[i]<<" ";
    cout<<"}"<<endl;
}
~liste()
{
    delete [] adr;
    cout<<"Tableau elimine..."<<endl;
}
}; void main()
{
    liste l(4);
    l.saisie();
    l.affiche();
}
```

## Exercice 12

Ecrivez une classe nommée *pile\_entier* permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un tableau d'entiers alloués dynamiquement. La classe comportera 3 données membres :

- int dim : nombre maximal d'entiers de la pile,
- int \*adr : un pointeur,
- int nelem : nombre d'entiers actuellement empilés.

La classe comportera les fonctions membres suivantes :

- o *pile\_entier(int n)* : constructeur allouant dynamiquement un emplacement de n entiers,
- o *pile\_entier()* : constructeur sans argument allouant par défaut un emplacement de 20 entiers,
- o *pile\_entier(pile\_entier&)* : constructeur par recopie,
- o *~pile\_entier()* : destructeur,
- o *void empile(int p)* : ajoute l'entier sur la pile,
- o *int depile()* : fournit la valeur de l'entier situé en haut de la pile, en le supprimant de la pile,
- o *int pleine()* : fournit 1 si la pile est pleine, 0 sinon,
- o *int vide()* : fournit 1 si la pile est vide, 0 sinon.

## Solution exercice 12

```
#include <iostream.h>
#include <assert.h>
class pile_entier
{
    int dim; // nbre maximal d'entiers de la pile
    int *adr; // pointeur
    int nelem; // nbre d'entiers actuellement empiler
public :
    pile_entier(int n)
    {
        adr=new int[dim=n] ;
        nelem=0;
    }
    pile_entier()
    {
        adr=new int[dim=20];
        nelem=0;
    }
    pile_entier(pile_entier & p) // ou (const pile_entier & p)
    {
        adr=new int[dim=p.dim] ;
        nelem=p.nelem;
        for(int i=0;i<nelem;i++)
            adr[i]=p.adr[i];
    }
    ~pile_entier()
    {
        delete adr;
    }
}
```

## Solution exercice 12

```
void empile(int p)
{
    assert(!pleine());
    adr[nelem++]=p;
}
int depile()
{
    assert(!vide());
    return adr[--nelem];
}
int pleine()
{
    return (nelem == dim);
}
int vide()
{
    return (nelem == 0);
}
};
```

## Solution exercice 12

b) Ecrivez une fonction main utilisant des objets automatiques et dynamiques du type *pile\_entier*.



## Solution exercice 12

```
void main()
{
    // Création d'objets automatiques
    pile_entier a(3);          // une pile de 3 entiers
    -- ou pile_entier a=3 ; appel du constructeur avec un seul argument
    pile_entier b ;            // une pile de 20 entiers par défaut
    cout<< "a pleine ? "<< a.pleine() <<endl ;
    cout<<"a vide ? "<< a.vide() <<endl ;
    a.empile(5) ; a.empile(15) ; a.empile(31) ;
    cout<< "contenu de a : " ;
    for(int i=0 ; i<3 ; i++)
        cout<<a.depile() << " " ;
    cout<<endl ;
    b(a) ;
}
```

## Solution exercice 12

```
cout<< "b pleine ? "<< b.pleine() <<endl ;
cout<<"b vide ? "<< b.vide() <<endl ;
// Création d'un objet dynamique
pile_entier * adp=new pile_entier(5) ; // pointeur sur une pile de 5 entiers
cout<<"pile dynamique vide ? "<< adp->vide() <<endl ; //ou (*adp).vide() ;
for(int j=0 ; j<5 ; j++)
    adp->empile(10*j) ;
cout<<" contenu de la pile dynamique : " ;
for(int k=0 ; k<5 ; k++)
    cout<<adp->depile()<<" " ;

// initialisation d'objet :
pile_entier a1(10) ; // création d'un objet automatique
pile_entier b1=a1 ; // il doit exister un constructeur par recopie.
// on initialise l'objet b avec l'objet a de même type.
// on obtient la même résultat avec pile_entier
```