PROGRAMMATION PAR OBJETS

LANGAGE C++

Notes de Cours

N. Castagné, M. Desvignes, F. Portet

Version 2013.02

PLAN DU COURS

1. C++

==> **C++ = C** + typage fort + programmation objet <==

➢ Naissance du langage C++ : 1979/1983

Bjarne Stroustrup, ATT Bell Labs.

Inspirés de SIMULA, ALGOL...
Initialement : extension du langage C, traducteur « C++ vers C »
Ajout de concepts objet (classification, encapsulation, héritage, abstraction...)

➢ Versions

1983 : première version (à l'origine *via* un traducteur de C++ vers C)
1998 : premier standard ISO, C++98
2011 : dernier standard en date (C++11)

➢ Objectifs et intérêt du langage C++ :

L'un des langages les plus utilisés aujourd'hui.

Communs aux approches objets :

Ecrire facilement de bons programmes de grande taille.
- Protection et robustesse
- Structuration, Réutilisabilité
- Lisibilité et évolutivité
- ...

Propres au C++ :

Haute performance : compilé, possibilité d'écrire « proche de la machine »
Du bit (bas-niveau) au concept (haut-niveau)
Possibilité de mixer paradigmes objet et impératif (« comme en C »).

2. LES APPORTS DU C++ AU C

OU LE C++ SANS LES OBJETS

➢ Le C++ comme une extension du C

Tout programme écrit en C compile avec un compilateur C++. Ajout au C :

- nombreux compléments : *dans cette partie*
  Rq : Certains ont été ajoutés ensuite au standard C récents.

- notions propres à l'approche objets : *partie 3 du cours*

➢ Compilateur et environnement de travail

A l'origine (1979-83), le C++ était implanté avec un traducteur « C++ vers C ».
Désormais : compilateur C++.
Nous utiliserons le compilateur **g++** de la GNU Compiler Collection.

Fichier header main.cpp pour « hello world »

```
#include <string>
#include <iostream>
main() {
    std ::string str = "Hello World" ; // un objet string !
    std::cout << str << std::endl ; // <=> printf(...)
}
```

Compilation dans le Terminal avec :

```
% g++ main.cpp -o main
```

Exécution avec :

```
% ./main
```

A part l'utilisation de g++, rien ne change dans l'environnement de travail, eg :

- Compilation en deux phases : compilation puis édition des liens
- Utilisation de fichier Makefile et de la commande make
- Debug avec le debugger gdb
- Possibilité d'utiliser la libc
- ...

COMMENTAIRES, ARGUMENT PAR DEFAUT, DECLARATION DE VARIABLES

➢ Commentaire de fin de ligne avec //

```
// Ceci est un commentaire
```

➢ Arguments par défaut

```
int f(int a, float x=3.14159)
// si x est omis lors de l'appel, x vaudra 3.14159
```

Rq 1 : les valeurs par défaut ne sont possibles que si les paramètres les plus à droite ont une valeur par défaut

```
int f(int a=9, float x) // Erreur
```

Rq 2 : les arguments de fonctions sont empilés (Stack) de la droite vers la gauche (C et Fortran) après évaluation et dépilés par la fonction appelée.

➢ Déclaration libre des variables et variables de boucle

Une variable peut être déclarée n'importe où dans le programme.

Portée d'une variable : une variable ne peut être utilisée qu'entre la ligne où elle est déclarée et la fin du bloc (ou du fichier) dans lequel elle est déclarée.

Variables de boucle: une variable peut être déclarée dans une boucle. Elle existe alors jusqu'à la fin de la boucle.

```
main() {
    int i;      // Création de i
    cin >> i;  // Lecture de i;
    j = 9;      // ERREUR : j n'existe pas encore
    int j = 12;    // Création de j
    for (int k=0; k<i; k++) {  // Création de k
      int localVar = k*10*j ;// Création de localVar.
      cout << localVar;
    } // fin de k et de localVar
    k = 8;          // ERREUR ! k n'existe plus
    localVar= 9;    // ERREUR ! localVar n'existe plus
} // fin de i,j
```

PORTEE DES VARIABLES

** 5 types de portées : locale, fonction, boucle, fichier, classe.

** **locale** : un nom déclaré dans un bloc (`{..}`)est local à ce bloc. Il ne peut être utilisé qu'entre la déclaration et la fin du bloc.

** **fonction** : les étiquettes peuvent être utilisées uniquement dans la fonction où elles sont déclarées.

** **variable de boucle** : du début à la fin de la boucle

** **fichier** : un nom déclaré hors de toute fonction, classe ou bloc a une portée de fichier. Il ne peut être utilisé qu'entre la déclaration et la fin du fichier. Attention à la directive `extern`.

** **classe** : un nom déclaré dans une classe est utilisable dans cette classe (fonction membres et amies).

** **Opérateur de résolution de portée : `::`**

accès à un membre de la classe nom_classe ou à la variable globale

Syntaxe : `<nom_classe>::membre`
`::nom`

Exemple:

```
int i;
void f(int j)
{
  int i;
  i=5;        // variable i locale
  ::i=10;     // variable i globale
}
```

NAMESPACE

Problème : mêmes symboles dans plusieurs bibliothèques utilisées
➔ Ambiguité sur les noms

Solution : les espaces de noms (namespace)
Nom complet des symboles:

```
nom_de_namespace::symbole
```

Exemple :

```
std::cin // la variable cin du namespace std
```

➢ Namespace global :

```
int i;
main(){ ::i = 6 ; }
```

➢ Utilisation d'un name space:

mot clé `using`

```
#include <iostream>
using namespace std;
main() {
    int i;
    cin >> i; // evite std::cin >>i;
}
```

➢ Définition d'un namespace

```
namespace nom_du_namespace{ …. };
```

Peut etre sur plusieurs fichiers. Le namespace est la reunion de tous les namespace de meme nom.

```
namespace monnamespace {
    int f() {…};
}
main() {int i;
    i=monnamespace::f();
}
```

SURCHARGE DE FONCTIONS

➢ Rappel : prototype d'une fonction

nom de la fonction, liste des types des paramètres, type de retour.
C++ : s'y ajoute pour les fonctions membres la classe et l'éventuel qualificateur const.

En C++, la déclaration des prototypes est obligatoire.

➢ Surcharge des fonctions

Un même nom de fonction peut être utilisé pour définir plusieurs fonctions si :

- nombre d'arguments différent
- ou au moins un des arguments est de types différents .

➢ Surcharge : déclaration

```
// Elévation à la puissance
int       pow(int x, int y);
double    pow(double x,  double y);
complex   pow(complex x, int y);
etc.
```

➢ Surcharge : appel

Lors de l'appel, une seule fonction est exécutée.
La surcharge est résolue à la compilation sur la base du type.
Choix en 3 étapes :

1. Recherche d'une correspondance exacte des types
2. Recherche d'une correspondance en employant les conversions prédéfinies du langage : `int` en `long`, `int` en `double`, `float` en `double` ...
3. Recherche d'une correspondance en employant les conversions définies par le développeur.

```
void affiche(double a)      { printf("%lf\n",a); }
void affiche(long a)        { printf("%ld\n",a); }
main(){  int i = 2; double x = 3.14;
    affiche(i);     // Conversion de i en long
                    // puis appel de      affiche(long)
    affiche(x);     // Appel direct de    affiche(double)
}
```

RESUME DES E/S

Notion de flot

Classes prédéfinies facilitant les E/S : gestion des flots (« stream »)

```
#include <iostream> <fstream> <ostream> <sstream> ...
```

Lecture : Clavier, fichier, stringstream

flot >> variables

Ecriture : Ecran, fichier, stringstream

flot << variables, constantes

Entrée / sortie standard
#include <iostream>
Variables `std::cin, std::cout, std::cerr`
`(l'équivalent de stdin, stdout, stderr en C)`

Exemple

```
#include <iostream>
main() { int i; float x; double z;
    std::cout << "Entrez un entier et deux réels ";
    // Pas de retour chariot automatique
    std::cin >> i >> x >> z;
    // Lecture d'un entier, puis d'un float
    // puis d'un double
    std::cout  << "i=" << i << " x=" << x
               << " z=" << z << std::endl;
    // Affiche i puis x puis z et retour chariot
}
```

Base pour la mise en forme des sorties

Retour chariot, affichage hexadécimal / decimal, notation exponentielle / scientifique et precision pour les flottants…

```
#include <iomanip>
...
std::cout << std::hex << 16 << " "
          << std::dec << 16 << std::endl;
std::cout << std::scientific << 0.5 <<  " "
          << std::fixed << 0.5 << std::endl;
std::cout << std::setprecision(2) << 3.14159
          << std::endl;
```

LA CLASSE STRING

➢ Les chaines, du C au C++

Rappel : en langage C ; pas de chaines mais des tableaux de char terminés par un caractère de fin de chaines : \0.

En C++, objets et encapsulation facilitent beaucoup les choses.
De nombreuses librairies définissent leur propre classe pour manipuler les chaines de caractères.

Nous utiliserons la classe std ::string du standard C++.

➢ La classe std::string

Premier exemple de classe dans ce cours ! page suivante.

Sortie du programme page suivante si l'utilisateur tape « Salut » :

```
taille de s1 : 13
entrez une chaine : Salut
s1: Hello World !
s2: Salut
s3 en chaine C : Hello World ! coucou Salut
s1 et s2 different
second caractere de s1: e
s4: World
Position de la chaine 'coucou' dans s3 : 14
 'plop' ne figure pas dans s3
s1 :
```

```
#include <string>
#include <iostream>
using namespace std; // évite « std:: » partout
main() {
    string s1;         // s1 : objet de type « string »
    string s2, s3, s4; // autres variables « string »

    s1 = "Hello World"; //conversion char* => string
    s1 += " !" ;        //opérateur de concaténation

    cout << "taille de s1 : " << s1.length() << endl ;

    // utilisation des string avec les flux standard c++
    cout << "entrez une chaine : ";
    cin >> s2 ;              // s2 est lue au clavier
    s3 = s1 + " coucou " + s2 ; // concaténation

    // utilisation des string avec les flux standard c++
    cout << "s1: " << s1 << endl << "s2:" << s2 << endl;

    // conversion string C++ vers chaine C
    printf("s3 en chaine C : %s\n",s3.c_str());

    // opérateur de comparaison ==.
    // Note : l'opérateur != existe aussi sur les string
    if(s1 == s2) {cout<< "s1 et s 2 sont identiques\n"; }
    else { cout << "s1 et s2 different\n" ; }

    // opérateur crochet [ ]. Ici, affiche 'e'
    cout << "second caractere de s1: " << s1[1] << endl;

    // extraction d'une sous-chaine
    s4 = s1.substr(6, 5) ;
    cout << "s4: " << s4 << endl ; // s4: World

    // recherche d'une chaine dans une chaine
    cout << "Position de la chaine 'coucou' dans s3 : "
         << s3.find("coucou") << endl ;
    if( s3.find("plop") == std::string::npos ) {
       //  std::string::npos est une constante qui vaut -1
       cout << " 'plop' ne figure pas dans s3" <<endl;
    }

    s1.clear() ; //vide s1
    cout << "s1 : " << s1 <<endl ; // s1 :
}
```

LA CLASSE STRINGSTREAM

➢ Classe stringstream : conversion string <=> valeur et variable

Comment convertir une valeur (eg : un int) en chaine de caractère ?
Comment lire une valeur (eg : un int) dans une chaine de caractère ?
Equivalent C++ des fonctions C sscanf() et sprintf() ?

On utilise une variable de type **stringstream** :
un flot (comme cin/cout) qui travaille sur / depuis la mémoire.

Ensuite, tout se passe comme avec n'importe quel flot !

➢ Exemple

```
#include <string>
#include <sstream>
using namespace std ;
main() {
    double x = 9.2;
    int i = 16;
    stringstream sStream1;
    sStream1 << "salut " << x << " " << std::hex << i;
    // conversion du stringstream en string
    string s = sStream1.str();
    cout << s <<endl ;

    string s1 = "bonjour 12 45.78";
    // nouveau stringstream
    // qui travaille sur le contenu de la chaine s1
    stringstream sStream2(s1);
    string s2 ;
    double x2;
    int i2;
    sStream2 >> s2 >> x2 >> i2;

    cout<<"s2 : "<<s2<<" x2 : "<< x2<<" i2 : "<<i2<<endl;
}
```

Ce programme affiche :

```
salut 9.2 10
s2 : bonjour x2 : 12 i2 : 45
```

REFERENCES (1)

➢ Définition

- **Synonyme pour un objet informatique** (une variable, une instance d'une classe...).
- Utilisation indifférente de la référence ou de l'objet informatique initial, quelque soit l'action réalisée.
- La référence et la chose ont même adresse, même valeur ...

➢ Syntaxe

```
type & identificateur = <variable> ;
// identificateur est une variable
// de type « reference vers type»
```

Une référence s'initialise toujours au moment de la déclaration.
On ne peut jamais changer ensuite cette initialisation.

➢ Exemple : variable

```
main() { int i, pi=NULL, j=2;
    int & ri = i;          // ri ⇔ i
    ri = 9 ;               // affecte 9 a ri... donc a i !
    ri = j ;               // affecte 2 a ri... donc a i !
    cin >> i;              // ⇔ cin >> ri;
    cout << ri;            // ⇔ cout << i;
    pi = &ri;              // ⇔ pi = &i;
    ri++;                  // Incrémente i
    &ri=j;                 // Interdit de référencer une
                           // autre variable !
}
```

➢ Exemple: Paramètre de fonction

```
// a et b sont des références à des objets
// extérieurs à la fonction, passés à la fonction !
swap( int &a, int &b){ int inter=a; a=b; b=inter; }
main() { int i,j;
    cin >> i >> j;         // Entrée de i et j
    cout << i << j;        // Affichage de i et j
    swap(i, j);
    // la fonction utilise des paramètres référence.
    // On l'exécute avec des références à i et j
    // La fonction travaille donc bien sur i et j !
    cout << i  << j;       // Affichage de i et j :
    // leurs valeurs sont inversées !
}
```

REFERENCES (2)

➢ Référence et pointeurs ? Référence, ou pointeur ?

Les notions de référence et de pointeur sont très proches.

La plupart des compilateurs C++ travaillent en fait dans votre dos avec des pointeurs lorsque votre code utilise des références.

« Quand une référence est utilisée, un pointeur pourrait l'être »

Exemple :

```
main() { int i;
int &ri = i; // ri est la même chose que i
ri =5 ;
/* Le code équivalent est :
     int *ri_assembleur = &i;
     *ri_assembleur = 5;   */
}
```

Pourquoi alors à la fois des références et des pointeurs en C++ ?

La différence entre pointeur et référence est essentiellement de nature *syntaxique*.

Dans certains cas, les références permettent un code **beaucoup** plus « naturel » et « lisible ».
Dans d'autres cas, toujours pour la clarté et la lisibilité du code, il faut au contraire éviter les références et préférer les pointeurs.

Conseils à venir plus tard dans ce cours.

REFERENCES (3)

➢ Remarques

On peut prendre l'adresse d'une référence : c'est l'adresse de la variable référencée :

```
int i = 9 ;
int &ri = i;
int *pi = &ri ;   // pi prend l'adresse... de i !
*pi = 10 ;        // i vaut 10
```

Attention : pas de références à des références, ni de références à des champs de bits, ni de tableaux de références (pb d'initialisation), ni de pointeurs sur des références (une référence n'est pas un objet « physique » : elle n'a pas d'adresse ou valeur au niveau assembleur).

➢ Référence et retour de fonction

Une référence à un objet peut être retournée par une fonction.

=> La fonction renvoie un synonyme de la variable retournée (au lieu d'une valeur)

Cela permet par exemple d'écrire :

```
int & f(int x) {...; return(aaa); }
// aaa retourné par reference !
f(i)=a+b; //=> modifie aaa via sa référence !
```

- **Exemple 1**

L'opérateur >> de cin est défini par :

```
istream & operator>> (istream &s, int &i)
{ // code lisant un entier
   return (s);
}

ce qui permet     cin >> i;
// Transcrit en istream::>>(cin,i)
cin >> i >> j;
// Transcrit en istream::>>(istream::>>(cin,j),i);
```

REFERENCES (4)

- **Exemple 2**

```
// f(int*, int) retourne une reference sur le premier
// élément nul du tableau supposé existé.
int & f (int *t, int n)
{ for (int i=0; i<n; i++)
    if (t[i]==0) return(t[i]);
}
// Le type de retour de la fonction est une
// référence, donc ce qui est retourné n'est pas
// la valeur de t[i] mais une référence à t[i] !

main() {
    int tab[10]={1,2,3,4,0,5,6,7,8,9};
    for (int i=0; i<10; ) cout << tab[i++]<< " ";
    // Affiche 1,2,3,4,0,5,6,7,8,9

    f(tab,10) = 15580; /* change la valeur du premier
                          élément du tableau qui est nul */
    for (int i=0; i<10; ) cout << tab[i++] << " ";
    // Affiche 1,2,3,4,15580,5,6,7,8,9
}
```

Attention : ne jamais retourner de référence à une variable locale à une fonction !

Ceci serait équivalent à retourner l'adresse d'une variable locale.

```
int & f (int n)  {
    int i;  // i est locale et automatique
    •••;
    return(i);// destruction de i. ERREUR de conception !
}

main()    { int a,b;
  a=f(10);
    // a est une référence à un objet qui
    // n'existe plus => plantage !
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (1)

const : nouveau mot clé pour imposer la « constance » de la valeur d'un objet informatique.

➢ Constantes - Variable const

Deux syntaxes :

```
main(){
    const int i = 2;          // i est un entier constant
    double const x = 3.14;    // x est un double constant
    i = 3 ;                   // INTERDIT : i est const
    printf("%lf\n",x);        // OK
}
```

Intérêt : **garantit le typage fort** vérifié à la compilation.

➢ Const et type des variables

Le mot clé "const" fait partie du type d'une variable : une "variable entière const" n'est pas du meme type qu'une "variable entière".

Le caractère const ou non const est vérifié à la compilation.
Exemple de messages d'erreur du compilateur :

```
error: assignment of read-only location
error: invalid conversion from 'const int*' to 'int*'
```

➢ Pointeur const

Comme toute variable, un pointeur peut être constant :

<type> * **const** ptr = <adresse> ;

=> plus le droit de changer la valeur de ptr (l'endroit ou il pointe).

```
main(){
    int i = 2, j = 3;
    int * const p_i = &i ;  // p_i est un ptr constant
    printf("%d\n", *p_i);   // OK
    *p_i = 8 ;              // OK
    p_i = &j ;              // INTERDIT
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (2)

➢ Pointeur et référence sur une variable const

Deux syntaxes équivalentes :

pointeurs	*références*
<type> **const** * ptr;	<type> **const** & ref = <adresse>;
const <type> * ptr;	**const** <type> & ref = <adresse>;

=> On a pas le droit d'utiliser le pointeur (ou la reference) pour *modifier* la variable

```
main(){
    int i = 2;
    const int * p_i = &i;
    // equivalent à int const * p_i = &i ;
    // p_i pointe sur int constant
    i = 9 ;                  // OK
    printf("%d\n", *p_i);    // OK
    *p_i = 8 ;               // INTERDIT

    const int & r_i = i;
    // equivalent à int const & r_i = i ;
    // r_i est une référence sur int constant
    printf("%d\n", r_i);     // OK
    r_i = 8 ;                // INTERDIT
}
```

➢ Pointeur, références et vérification de la *constness*

Un pointeur sur variable "const" peut pointer une variable "non const".
Un pointeur sur variable "non const" ne peut pas pointer une variable "const".

```
main(){
    int i = 2;
    const int * p_i = &i;   // OK
    const int & r_i = i;    // OK
    *p_i = 2 ;              // INTERDIT : *pi read_only
    r_i = 9 ;               // INTERDIT : ri read_only
    const int j = 9 ;
    const int * p_j = &j;   // OK
    int * p_jnonconst = &j; // INTERDIT
    const int & r_j = j;    // OK
    int & r_jnonconst = j;  // INTERDIT
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (3)

➢ Paramètre « const pointeur » d'une fonction

La syntaxe *const ptr* pour un paramètre interdit la modification dans la fonction de la variable pointée.

Exemple : paramètre tableau "const"

```
// équivalent à void afficherTab(int const * t, int n)
void afficherTab(const int t[ ], int n){
    for (int i=0; i<10; ) cout << t[i++]<< " ";
    cout << endl ;
    t[5] = 9 ; // INTERDIT du fait du const
}
```

=> une fonction qui travaille sur un tableau ou un objet *sans le modifier* devrait toujours declarer ce parametre "const".

Exemple :

```
// signature de strcpy() :
char *strcpy(char* s1, const char* s2);
```

=> une fonction qui travaille sur un pointeur *sans modifier la variable pointée* devrait toujours declarer ce paramètre pointeur const.

➢ Paramètre « const référence » d'une fonction

La syntaxe *const reference* pour un paramètre interdit la modification dans la fonction de la variable référencée.

Exemple :

```
// équivalent à void afficherString(string const & str)
void afficherString(const string & str){
    cout << "str vaut : " << str << endl ; // OK
    str = 8 ; // INTERDIT du fait du const
}
```

Nous reviendrons sur cette notion dans la suite du cours.

MOT CLE *CONST ET* VARIABLES *CONST* (4)

➢ Exemple

Exemple :

```
 #1                    #2          #3
const int * function(const int * const p_i);
```

est équivalent à :

```
     #1                    #2       #3
int const * function(int const * const p_i);
```

#3 indique que le pointeur à gauche est *const* : **dans la fonction**, on ne pourra pas changer l'endroit ou il pointe.

```
// INTERDITS dans le code de la fonction:
p_i = <une autre adresse> ;
```

#2 indique que l'entier pointé est *const.* Permet de savoir que la fonction ne modifiera pas l'entier pointé. D**ans la fonction**, on ne pourra pas changer la valeur de cet entier

```
// INTERDITS dans le code de la fonction:
*p_i = 9 ;
p_i[6] = 2 ;
```

#1 indique que l'adresse retournée par la fonction pointe un entier const : après de l'appel de la fonction, le cone de pourra pas changer la valeur de cet entier.

```
main() {
    int k = 9 ;
    * ( function(&k) ) = 3;              // INTERDIT
      // car le ptr retourné pointe un entier const :
      // "assignment of read-only location"
    int * ptr1 = function(&k) ;          // INTERDIT
    // invalid conversion from 'const int* const' to 'int*'
    int * const ptr2 = function(&k) ;    // INTERDIT
    const int * ptr3 = function(&k) ;    // OK
    printf("%d\n", *ptr3);               // OK
    *ptr3 = 9;                           // INTERDIT
}
```

GESTION MEMOIRE – ALLOCATION

Gestion de la mémoire dynamique par les opérateurs new et delete
Plus efficace, meilleure cohérence (pas de fonctions spécifiques)
Même priorité que `++ ;-- ;+ ;- ;(type) ;* ;& ;sizeof ;`

➢ Allocation dynamique : opérateur new

Syntaxe : `new <type>`

- Alloue sizeof(type) octets.
- N'initialise pas la mémoire, retourne l'adresse allouée.
- Implémenté par une fonction `void * operator new (long)`
- Exemple :

```
int *pi= new int; // Création d'un entier. Sans init !
*pi = 4567;
```

Allocation avec initialization:

```
int *pi= new int(12); // int initialisé à la valeur 12
```

➢ Allocation dynamique de tableaux : opérateur new [ ]

Syntaxe : `new <type> [ <nb> ]`
Pas d'initialisation possible

```
// Création d'un tableau de 1000 char
char *pc = new char[1000]; // valeurs aléatoires !
int n;
cin >> n;
double *pd = new double[2*n+1]; // 2*n+1 double
```

➢ Allocation dynamique : instruction set_new_handler

Quand l'opérateur new ne peut allouer la mémoire, il peut faire appel à une fonction de l'utilisateur, précisée grâce à `set_new_handler`

```
#include <new>
void ma_fonction() {
    cout << "Erreur allocation mémoire : sortie";
    exit(1);
}
main() { set_new_handler(&ma_fonction);
     char *p=new char [1000000000];
}
```

GESTION MEMOIRE : LIBERATION

➢ Libération : opérateur delete

Syntaxe : `delete pointeur`
Libère la mémoire allouée par new. Implémenté par une fonction
`void * operator delete (void *)`

```
int *pi= new int; // Création d'un entier
*pi = 4567;
delete pi;
```

- delete(NULL) ne provoque pas d'erreur (sauf dans les premières versions)
- delete(p) est indéfini si p n'est pas correctement affecté

➢ Libération d'un tableau alloué dynamiquement : opérateur delete []

Syntaxe : `delete [ ] pointeur`
Libère la mémoire allouée par new [ ]

```
// Attention aux []
char *pc = new char[1000];
// Création d'une chaîne « C » de 1000 char
delete [] pc; // Attention aux [] !
int n;
cin >> n;
double *pd = new double[2*n+1];
…
delete [] pd; // Attention aux []
```

➢ Remarques

1. new et delete peuvent être redéfinis. `::new` et `::delete` sont les opérateurs standards
2. new invoque un constructeur d'un objet, delete le destructeur
3. Ne pas mélanger les pointeurs obtenus avec malloc et avec new
4. Ne pas confondre delete et delete[ ]

ECHANGE DE FONCTIONS

➢ Fonction C -> appel C++ :

Pour utiliser une librairie compilée avec un compilateur C dans un programme C++, il suffit de déclarer les prototypes des fonctions C avec la directive extern "C"

```
extern "C" { double mafct_C (double); }
main() {
    double rest = mafct_C(4) ; //appel depuis du code C++
}
```

➢ Handler C qui appelle du code C++

Pour qu'une fonction d'une librairie C puisse utiliser du code C++, : définir de nouvelles fonctions qui feront le lien avec les méthodes d'un objet particulier et imiteront ces méthodes.

Exemple : utiisation de l'algorithme qsort() de la librairie C.
Prototype de qsort (man qsort) :

```
void qsort(void *base, size_t nel, size_t width,
                  int (*compar)(const void *, const void *));
```

ou *compar()* est un *pointeur sur une fonction C* qui doit retourner la comparaison des deux arguments passés en paramètre.

```
#include <stdlib.h>
int monComparateur_C(const void * p1, const void * p2) {
    const string * s1 = static_cast<const string *> (p1);
    const string * s2 = static_cast<const string *> (p2);
    return s1->compare(*s2); //methode string::compare()
}

main() {
    string tab[3];
    tab[0] = "ef"; tab[1] = "ab" ; tab[2] = "cd";
    for(int i = 0 ; i < 3 ; i ++) cout << tab[i] << " " ;
    cout << endl;

    qsort( tab, 3, sizeof(string), monComparateur_C);
    for(int i = 0 ; i < 3 ; i ++) cout << tab[i] << " " ;
    cout << endl;
}
```

CONVERSION ; CAST

Nouveaux opérateurs de conversion, plus sécurisés et adaptés à la programmation objet.

➢ Conversion prédéfinie : `static_cast<T> (expr)`

- idem anciennes conversions
- conversions résolues à la compilation
- pas de vérifications à l'exécution
- doit être utilisé pour des conversions non-ambiguë

➢ Suppression de la constance : `const_cast<T> (expr)`

- résolu à la compilation
- Exemple :

```
void affStr(string & str ) {
    cout << "Chaine : " << str << endl;
}

void g(const string & s) {
    affStr(s);          // Erreur: j constant
                        // et affStr n'attend pas un const
    affStr(  const_cast< string & >(s)  );  // Ok
}

//Remarque : affStr(), en fait, devrait prendre un
//const string& en paramètre puisqu'elle ne modifie pas s
```

➢ Conversion dynamique : `dynamic_cast<T> (expr)`

- Utilisé sur des pointeurs ou des références dans une hiérarchie de classes

➢ Conversion de pointeur : reinterpret_cast<T> (expr)

- Permet de transformer n'importe quel pointeur en n'importe quel pointeur.
- Pas de vérification à l'exécution.
- Peut être utilisé sur des objets sans liens : dangeureux

FONCTIONS INLINE

Une fonction (fonction simple ou fonction membre) déclarée **inline** est expansée par le compilateur : l'appel de la fonction est remplacé par son code.

Efficacité :

- Pas de passage de paramètres sur la pile, pas d'appel de fonction, pas de restauration de la pile.
- Très utile dans les méthodes d'accès aux membres privés.

Déclaration explicite :

```
inline int uneFctInline() { .... } ;
```

Déclaration implicite : les fonctions membres définies dans la déclaration de la classe (=> en général, dans le fichier header), sont inline (dépend du compilateur).

Le corps doit être mis dans le fichier header (.h) et non pas dans le .cpp :

Fichier header MaClasse.h

```
// fonctions C
inline int fac(int n) {return i<2 ? i : i* fac(i-1) }

// fonctions membres
class A {
    //implicite dans la déclaration de la classe
    int f() { return 2 ; }
    int g() ;
} ;

// déclaration explicite de g en inline
inline int A::g() { return 3 ; }
```

Peut être ignorée par le compilateur (en particulier s'il y a des boucles à l'intérieur)
Implantation réelle dépend des compilateurs.
Exemple :

```
inline fac(int n) {return i<2 ? i : i* fac(i-1) }
```

L'appel se fait par `fac(5);`
Le compilateur générera :

- soit 720
- soit 6*5*4*3*2*1
- soit 6*fac(5)

3. PROGRAMMATION ORIENTEE OBJETS ?

➢ Programmation procédurale (en C par exemple)

Paradigme : Choisissez les ***traitements*** dont vous avez besoin et utilisez les meilleurs algorithmes

Programme : ensemble séquentiel de sous-programmes

Caractéristiques :

Séparation complète entre code et données manipulées
Communication entre sous-programmes par paramètres et variables globales
Usage permanent de conditions portant sur le type des données manipulées

➢ Programmation par objets (en C++ par exemple)

Paradigme : Choisissez les ***données*** *et les* ***types*** dont vous avez besoin.

- Identifiez les entités que manipulera le programme **=> objets**
- Déduire les concepts (types) dont ces objets sont des instances **=> classes**
- Pour chaque classe :
  - Précisez les **attributs**, qui définissent l'état des objets
  - Définissez un ensemble complet **d'opérations** sur ces entités.
  - Identifiez l'interface : séparez ce qui est interne (privé) et ce qui est visible de l'extérieur (public) **=> interface publique et encapsulation**
- Rendez explicites les points communs entre ces classes et les relations entre classes **=> hiérarchie de classes.**
- Organisez les données et leur sécurité (dépendances)

Programme : ensemble d'objets caractérisés par leur état et par les opérations qu'ils connaissent.

CONCEPTS ET MOTS CLES DE LA POO (1)

NOTION DE CLASSE ET D'OBJET

➢ Qu'est-ce qu'un objet ?

En POO, on utilise des **objets** pour représenter toute entité identifiable :
- une chose tangible ex: une ville, un étudiant, un bouton sur l'écran...
- une chose conceptuelle ex: une date, une réunion, une collection...

Un objet existe en mémoire et est caractérisé par :
- Son **état**, défini par la valeur de chacun de ses **attributs**.
  Chaque objet a un état qui lui est propre.
  Exemple: l'objet « *maVoiture* » a un attribut nommé 'couleur' qui vaut vert, un attribut *position* qui indique la position de la voiture sur une carte, un attribut *moteur*...
  L'objet *laVoitureDePaul* a une autre couleur, une autre position, un autre moteur.

- Son **comportement** : ce qu'il sait faire, ce qu'on peut faire avec. Défini par des **méthodes** (ou *fonctions membres*, ou *opérations*).
  Exemple: l'objet « *maVoiture* » réagit au message *démarrer()* en allumant le moteur de l'objet.

➢ Qu'est-ce qu' une classe ? un concept / un type / un moule

Un objet est créé à partir d'un « moule », ou « type » : sa **classe**.

Instancier une classe, c'est créer un objet qui suit le « moule ».
Tout objet est une **instance** d'une et une seule classe.

Une classe est donc une abstraction qui représente un ensemble d'objets de même nature (mêmes types d'attributs et mêmes méthodes).

Exemple : les objets *maVoiture* et *laVoitureDePaul* sont deux instances de la classe *Voiture*.

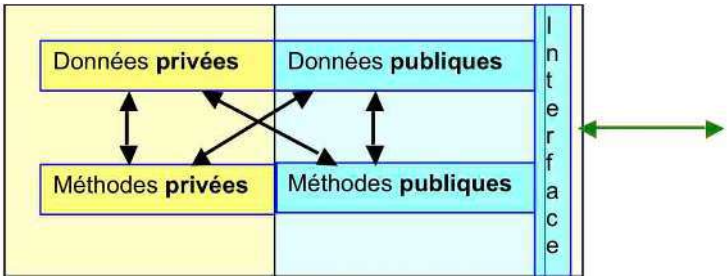
Une classe définit les membres qu'auront toutes ses instances :
- Les types des ***attributs*** (eg : toute « `Voiture` » a une « couleur » une « position » et un « `Moteur` »)
- des ***opérations*** (ou *fonctions membres* ou *méthodes*) qui manipulent ces attributs (eg : toute Voiture a une opération *démarrer()*, qui allume le moteur).

CONCEPTS ET MOTS CLES DE LA POO (2)

➢ Encapsulation, interface publique

La POO permet de séparer dans les objets (et dans les classes) :
- Une partie visible, publique, qui définit la façon dont l'objet (la classe) s'utilise.
- Une partie encapasulée, privée, invisible depuis l'extérieur.

L'interface publique d'un objet (ou d'une classe) est l'ensemble de ses attributs et méthodes publiques, accessibles de l'extérieur.
La **visibilité** d'un attribut ou d'une méthode détermine les conditions d'accès à cet attribut ou d'utilisation de cette méthode.



Exemple :
En interne, dans une classe `Montre` on peut choisir de stocker l'heure courante *soit* en nombre de millisecondes écoulées depuis 1970 *soit* avec trois attributs : heure, minute, seconde. Ceci relève d'un détail d'implémentation, qui sera privé.

Quel que soit ce choix « encapsulé » dans la classe, l'interface publique de `Montre` permettra d'accéder à l'heure courante de toutes manières utiles.

➢ Composition entre objets et classes

Possibilité de définir des objets composites, fabriqués à partir d'autres objets.

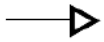
◆—— **"composé-de"** (lien *a-un* ou *a-des*)
relation contenant, champ de la classe ou de la structure

Exemple : Un chien a des pattes, une gueule, etc.
- ⇨ Un objet instance de `Chien` possède 4 objets instances de `Patte`, et un objet instance de `Gueule`, etc.
- ⇨ La classe `Chien` est composée avec les classes `Patte` et `Gueule`.

CONCEPTS ET MOTS CLES DE LA POO (3)

➢ Héritage

Une classe X peut servir de base pour définir une nouvelle classe Y.



"sorte-de" (lien *est-un*)

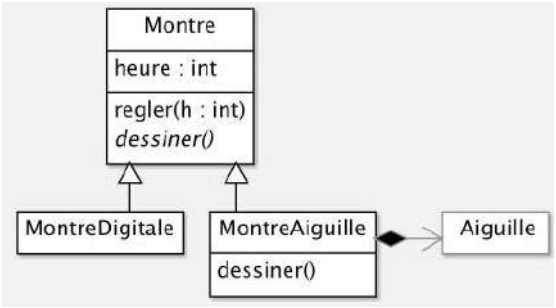
relation hiérarchique père-fils, notion d'héritage

On réutilise ce qui a été déjà fait (classe X), en ajoutant de nouveaux membres (extension) et/ou en modifiant les fonctions de la classe X (redéfinition).

Exemple :
Intuitivement, une « montre digitale » et une « montre à aiguille » sont deux types de « montres ». En programmation objet, on fait hériter les classe `MontreDigitale` et `MontreAiguille` d'une classe `Montre`.

Tout ce qui est commun à toutes les montres (e.g : le fait qu'elles ont l'heure, qu'on peut les regler(), ...) :

- est défini dans la classe `Montre` (écrit pour tous les types de montres)
- est *hérité* dans les classes `MontreDigitale` et `MontreAiguille`.



Un objet instance de `MontreAiguille` :

- **hérite** des propriétés (attributs et méthodes) de la classe `Montre`
- y ajoute ses propres spécificités (par exemple, deux instances d'une classe `Aiguille`).

On dit que :

- Les classes `MontreDigitale` et `MontreAiguille` **héritent de** ou **spécialisent** ou **étendent** ou sont des **classes filles** ou des **sous-classes** de la classe `Montre`.
- La classe `Montre` **généralise**, est une **classe mère**, est une **super-classe** des classes `MontreDigirale` et `MontreAiguille`.

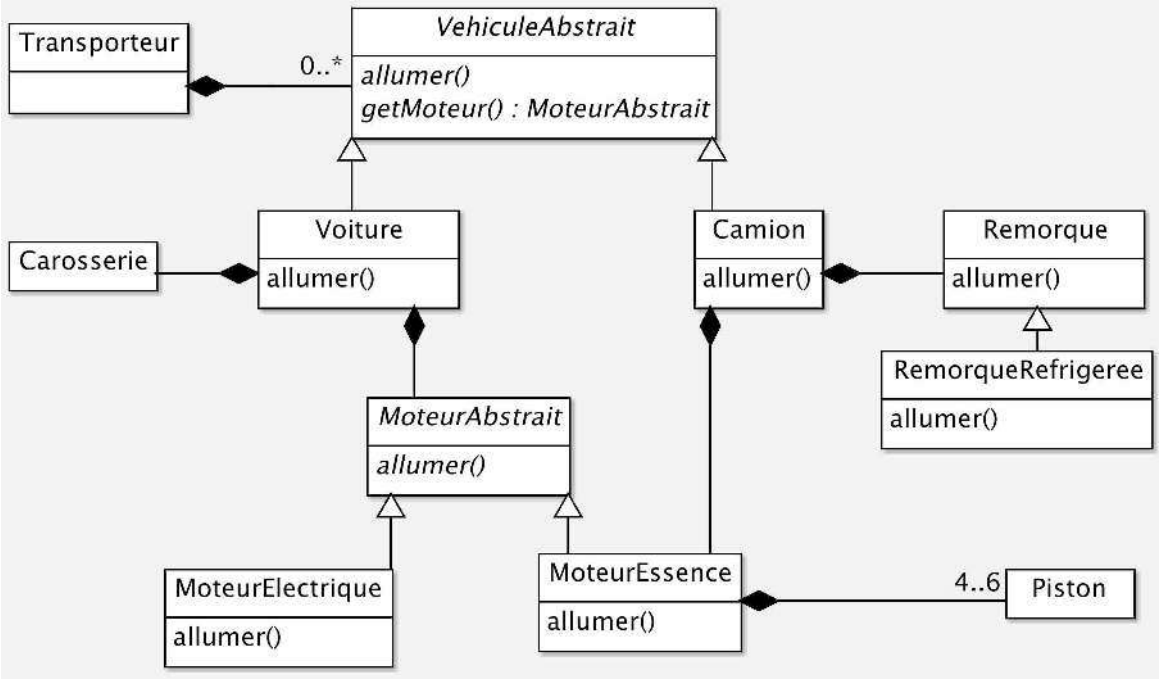
Intérêts:

- factorisation du code commun à toutes les montres dans `Montre`
- **Polymorphisme** : possibilité de manipuler à la fois des objets intances de `MontreDigitale` ou `MontreAiguille` en tant que `Montre` (sans s'intéresser à ce qu'elles sont véritablement)

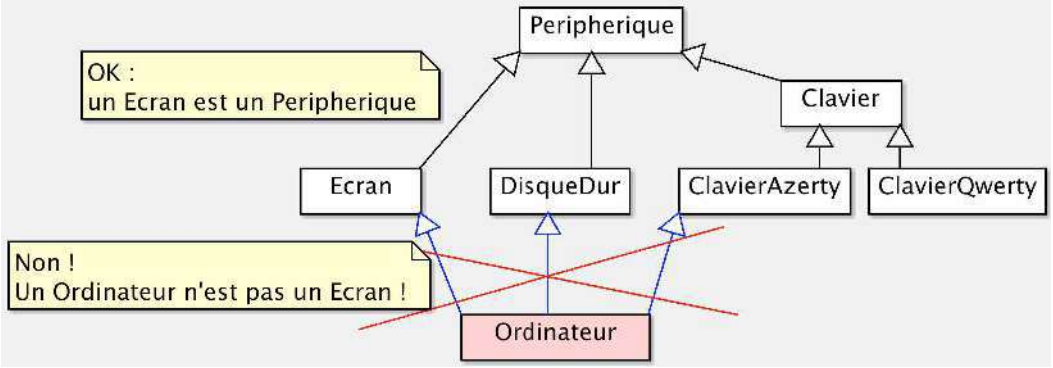
CONCEPTS ET MOTS CLES DE LA POO (4)

➢ Exemple

Une voiture est un véhicule. Un camion est aussi un véhicule.
Une voiture est composée d'une carrosserie et d'un moteur.
Le moteur peut être électrique ou à essence.
Un moteur à essence est composé de pistons. Il y a 4 ou 6 pistons.
Un camion a un moteur à essence et une remorque. Il existe des remorques réfrigérées.
Tous les véhicules peuvent être allumés.
Allumer une voiture, c'est allumer son moteur.
Allumer un camion, c'est allumer son moteur et sa remorque, si elle est réfrigérée.
Un transporteur a plusieurs véhicules. Le matin, ils sont tous allumés.



➢ Ne pas confondre héritage et composition !



Un ordinateur *n'est pas* une sorte de disque ou de clavier. Mais il est *composé* d'un clavier, d'un type particulier (AZERTY ou QWERTY) , d'un disque dur ...

AVANTAGES DE LA POO

L'approche objet :

- permet la conception et l'implantation de logiciels *plus fiables* et *plus aisés à maintenir*.

- accompagne toutes les phases de la vie d'un programme : analyse du problème, conception, implantation.
  Les concepts objets se prêtent bien à l'échange entre concepteurs et développeurs, entre développeurs, entre utilisateurs et concepteurs...

- s'appuie naturellement sur des représentations graphiques
  L'approche objet est « naturelle ».
  UML (Unified Modeling Language) : un standard graphique pour la POO.

- permet de préciser et manipuler plus aisément les relations entre les concepts du programme.
  Exemple : exprimer immédiatement des relations est-un *ou* est-composé-de *et leurs variantes.*

- permet d'exprimer et de maitriser des architectures logicielles complexes.

- permet d'organiser ce qui relève de la partie publique (*l'interface*) d'un objet, de ce qui relève de la partie *privée* (le fonctionnement interne, les choix d'implantation...).
  Exemple : une classe Complexe permettra de manipuler un complexe indifféremment en représentation cartésienne ou polaire. Peu importe le choix fait pour stocker « en interne » la valeur du nombre Complexe !

Remarques :

- On peut « penser objet » / « concevoir un programme orienté objet » quelque soit le langage utilisé.
  Mais certains langages sont bien sûr plus adaptés.
- Un programme orienté objet n'est pas (nécessairement) plus lourd qu'un programme !

4. CLASSES ET OBJETS EN C++

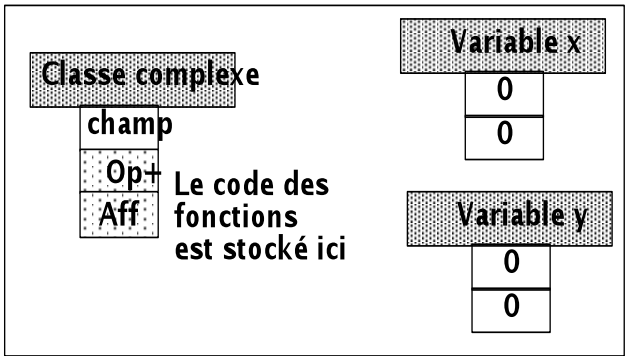
➢ Déclaration d'une classe : syntaxe

```
class id_classe {
    déclaration données_membres;
    déclaration fonction_membres;
}; // attention au « ; » !
```

Exemple:

```
class Complexe {
    float re;
    float im;  // Deux valeurs pour chaque point

public:   // Pb de protection. A voir plus tard

    // Fonction membre d'affichage
    void affiche() { cout << re << im; }

    // Redéfinition de + pour l'addition
    Complexe operator + (Complexe y) {
      Complexe r;
      r.re=re+y.re; r.im=im+y.im;
      return r;
    }

    •••
    // Fonctions amies
    friend ostream& operator << (ostream&, complexe)
};  // ATTENTION au ;
```

```
main() {
    // création de 2
objets x et y

    Complexe x, y;

}
```



ACCES AUX MEMBRES

➢ Membre : méthode ou donnée propre à une classe // à un objet

Le nom complet d'un membre est :

```
identificateur_de_classe::membre
```

En l'absence d'ambiguité, on omet `identificateur_de_classe::`

Données membre : l'accès se fait par

```
identificateur_de_variable.NomClasse::donnée
identificateur_de_pointeur->NomClasse::donnée
```

en l'absence d'ambiguité :

```
identificateur_de_variable.donnée
identificateur_de_pointeur->donnée
```

Fonction membre : toute fonction déclarée dans une classe est une fonction membre, ou « méthode ».
Son exécution se fait par :

```
identificateur_de_variable.NomClasse::fonction(params)
identificateur_de_pointeur->NomClasse::fonction(params)
```

en l'absence d'ambiguité :

```
identificateur_de_variable.fonction(paramètres)
identificateur_de_pointeur->fonction(paramètres)
```

Exemple :

```
class Complexe {
public: //  données membres » ou « attributs » :
    float re,im;
    // « fonction membres » ou « méthodes » :
    void affiche()      {....}
    void ajouteReel(float i) ;
};
void Complexe::ajouteReel(float i) { .... }

main() { // On cree 2 objets a et b
    Complexe a,b, *pa;
      // On donne des valeurs aux données de a et b
    a.re=10 ; a.im=0 ; b.re=15 ; b.im=20 ;
    // On execute la fonction affiche pour l'objet a
    a.affiche();
    // On pourrait écrire a.Complexe ::affiche()
    b.ajouteReel(5.1);
    pa = &a;
    pa->ajouteReel(3.0);
}
```

ACCES AUX MEMBRES DANS UNE METHODE

➢ Pointeur *this*

Une fonction membre s'exécute toujours sur un objet.

Dans le code d'une fonction membre, le mot clé ***this*** est toujours un **pointeur sur l'objet sur lequel la fonction est en train d'être exécutée**.

this est utilisé pour désigner les attributs et autres fonctions membres de l'objet sur lequel la fonction est executée

this n'existe que dans les fonctions membres

```
class Complexe {
public:
    float re, im;
    void affiche ( ) {
      cout     << "Complexe = (" << this->re
      << " + " << this->im  << " * i)" << endl ;
      // this est de type "Complexe *"
      // et pointe l'objet sur lequel la méthode
      // aff() est en train d'être exécutée.
      // this->re est l'attribut "re" de cet objet.
    }
} ;


main() {
    Complexe x, y, *py=&y ;
    x.re = x.im = y.re = 0; y.im=1;

    // Exécution de la méthode complexe::affiche() sur x
    // this->re et this->im du code de la fonction
    // complexe::aff() sont ceux de x.
    x.affiche();   // affiche "Complexe = (0 + 0 * i)"
    py->affiche(); // affiche "Complexe = (0 + 1 * i)"
}
```

ACCES AUX MEMBRES DANS UNE METHODE

➢ Que se passe-t-il en fait ?

Dans le code assembleur généré par le compilateur, en fait, la méthode affiche() aura un premier paramètre qui sera l'adresse de l'objet sur lequel elle s'applique.

```
main() {
    complexe x ; x.re = 0 ; x.im = 1 ;
    x.aff();  // En réalité : complexe::aff(&x) : this=&x
}
```

➢ Le pointeur *this* est optionnel

Dans une fonction membre, le pointeur **this** peut être omis pour désigner les données ou fonctions membres s'il n'y a pas d'ambiguité.

```
class Complexe {
public:
    float re, im;
    void affiche ( ) {
      cout     << "Complexe = (" << re
      << " + " << im  << " * i)" << endl ;
      // re <=> this->re    et   im <=> this->im
    }
} ;
```

En général, on n'utilise **this** que lorsqu'il permet de lever une ambiguité :

```
class Complexe {
public:
    float re, im;
    void ajouterReel(float re) {
      this->re = this->re + re ;
      // utiliser "this" leve l'ambiguité
      // entre le paramètre re de la fonction
      // et l'attribut re de l'objet
    }
} ;
```

CLASSE : DECLARATION ET CODE SOURCE

Déclaration d'une classe : liste de ses attributs et prototypes de ses méthodes, entre les acolades suivant le mot clé ***class***.
Implantation d'une classe : code de ses méthodes.

Une classe=>2 fichiers : header .h (déclaration + méthodes *inline*) source .cpp

Fichier header Employe.h : *définition* de la classe

```
#ifndef EMPLOYE_H_ // identificateur unique, comme en C !
#define EMPLOYE_H_
#include <string>
//déclaration de la classe Employe et méthodes inline
class Employe {
   string nom;
public: // on verra plus tard...
   //inlining implicite
   void setNom(const string & s) { nom = s; }
   const string & getNom() const ;
   void aff() const ;
};

// inlining explicite
inline const string & Employe::getNom() const {return nom;}
#endif
```

Fichier source Employe.cpp *(ou.cc)* : *code des fonctions membres* de la classe

```
#include <iostream>
#include "Employe.h"
//implantation des méthodes non inline
void Employe::aff() const {
   cout << "employe de nom : " << getNom();
}
```

Fichier main.cpp : programme principal

```
#include "Employe.h"
main() {
   Employe e, *pe;
   pe = &e;
   e.setNom("Paul");
   e.aff();
   cout << "coucou " << pe->getNom();
}
```

CLASSE : VISIVIBILITE DES MEMBRES

Protection : en C++, les données membres sont en général **protégées** contre les intrusions de fonctions non autorisées.

C'est la classe en cours de définition qui déclare quelles sont les fonctions qui peuvent accéder à ses membres (données ou fonctions) au moyen de **qualificateurs de visibilité**.

➢ mots clés : public, private, protected

** **private :** les membres déclarés privés ne sont accessibles que par les fonctions membres ou les fonctions amies de la classe

** **public :** membres accessibles par n'importe quelle fonction

** **protected :** accessibles que par les fonctions membres, les fonctions amies et les membres des classes dérivées (-> voir « *héritage* »)

** **Par défaut,** tous les membres des classes sont de type **privé** en C++.

Exemple:

```
class Complexe {
    float re, im; // Privés par défaut
public:
    int essai; // accessible de l'extérieur
private: // Fonctions inaccessibles de l'exterieur
    void affiche() { cout << re << " " << im; };
public: // Fonctions accessibles de l'extérieur
    void aff() { affiche(); };
    // on peut utiliser un membre privé
    // depuis une méthode public bien sur !
};

main() {
    Complexe x;
    x.re=0;       // INTERDIT car re est privé.
    x.essai=0;    // Autorisé car essai est public
    x.affiche(); // INTERDIT car affiche est privé.
    x.aff();      // Autorisé car aff est public
}
```

ACCESSEUR ET MODIFIEUR/MUTATEUR.

➢ Encapsulation : *private* par défaut !

Principe *d'encapsulation* :

- **Tout** ce qu'il n'est pas **nécessaire** de connaître de l'extérieur est caché (en particulier : les « détails de son implantation ») : **private**.
- **Seul** ce qui relève de « l'interface » de la classe est visible : **public**.

Règle générale : **déclarer *private* tous les membres qui n'ont pas de raison d'être public**.

Intérêts :

- Il devient possible, plus tard, de modifier « l'intérieur » sans avoir à modifier le reste du programme.

➢ Notion d'accesseur et de modifieur

On a recours à des méthodes pour accéder aux attributs ***private***.

Accesseur : méthode qui pour rôle de renvoyer/consulter la valeur d'un attribut.

```
<type> getNomDeAttribut()
```

Modifieur : méthode qui modifie la valeur d'un attribut(s) ou de plusieurs.

```
void setNomDeAttribut( <paramètre(s) > 
```

Intérêts :

1. *accéder* à l'état de l'objet nécessite parfois de faire un calcul (voir le Td sur les complexes). Les accesseurs unifient l'accès à l'état des objets.

2. *modifier* la valeur d'un attribut nécessite parfois de faire des traitements : des vérifications d'erreur (valeur voulue paramètre invalide...), des corrections/modifications de la valeur d'autres attributs pour garantir un « *invariant de classe* », etc. Les modifieurs systématisent cela.

Inconvénients

Un peu plus lourd à ecrire.

METHODES CONST ET NON CONST

➢ Méthode const : mot clé *const* dans la signature d'une méthode

En C++, le mot cle *const* à la fin du prototype d'une méthode indique que la méthode est un *accesseur et ne peut en aucun cas modifier l'objet.*

Vérification à la compilation :

- une méthode *const* ne peut pas modifier les valeurs des attributs de *this*
- une méthode non *const* ne peut être appelée que sur un objet non *const*

```
class X {
private:
    int t, p;
public:
    // accesseur: const
    int getT() const {  return t ; }

    // Accesseur déclaré non const : erreur de conception
    // mais pas signalée par le compilateur.
    int getP() {  return t ; }

    void afficher() const { // accesseur : const
      cout << p << t;       // OK
      t=5;                  // INTERDIT à la compilation
    }

    // modifieur: non const
    void setP(int p) { this->p=p; }
    // Modifieur déclaré const ! Conception erronée
    void setT(int t) const {
      this->t=t;            // INTERDIT
    }
};
main() {
    const X x_const;
    x_const.afficher();     // OK
    x_const.setP(3);   //INTERDIT, x_const est une
constante
}
```

« const correctness » *:* fixer le mot caractère *const* des méthodes dès le début !

- gain de temps plus tard
- code plus robuste, plus lisible

PARAMETRES CONST & DES FONCTIONS

PASSAGE PAR PSEUDO-VALEUR (1)

➢ Rappel : passage par valeur lors de l'appel de fonctions

En C++, *les passages des paramètres aux fonctions se font « par valeur »*.

Rappel : si on utilise un paramètre pointeur, par exemple :

```
void swap(int* a, int* b) {...}
```

il s'agit *toujours* d'un passage par valeur... la *valeur* passée à la fonction est alors une *adresse* !

➢ Passage par valeur d'un objet

Comme toute variable, un objet est passé par valeur :

```
class X {
    int a ;
    public :
    int getA() const { return a ;}
    void setA(int valeur) { a = valeur ; }
} ;

// travaille sur une COPIE de l'objet passé en paramètre
void fonctionTravaillantSurX( X x ) {
    x.setA( 4 ) ;
    cout << "x.a vaut " << x.getA() ;  // affiche 4
}

main() {
    X x ;
    x.setA(3) ;
    fonctionTravaillantSurX(x) ;
    cout << "x.a vaut " << x.getA() ; // affiche 3
}
```

Problème d'optimalité lorsqu'on passe un objet par valeur, copier un gros objet peut être très couteux !

PARAMETRES CONST & DES FONCTIONS

PASSAGE PAR PSEUDO-VALEUR (2)

➢ Passage par pseudo valeur

Lorsqu'une fonction travaille sur un **objet** *sans le modifier*, on utilise un paramètre ***const référence*** pour simuler un passage par valeur.

```
class X {
    int a;
public :
    int getA() const { return a ; }
    void setA(int a)  { this->a = a ; }
} ;

void afficherX( const X & param ) {
    // x est une reference const sur un objet X
    // => on ne peut appeler que des accesseurs (const)
    cout << "param.a() vaut " << param.getA() ;
}

main() {
    X x ;
    x.setA(3) ;
    afficherX(x) ; // Syntaxe d'un passage par valeur
    // La fonction afficherX() prend une référence sur X
    // Elle travaille sur directement avec x.
}
```

Avec une const rérérence tout se passe comme si on avait fait un passage par valeur, mais on évite la copie de l'objet.

- La référence évite que l'objet soit copié (la fonction travaille avec l'adresse de l'objet passé directement) tout en gardant la syntaxe d'un passage par valeur.

- Le mot clé *const* permet la vérification syntaxique (par le compilateur) de la non modification de l'objet par la fonction.

ACCESSEUR RETOURNANT UNE CONST &

RETOUR D'ATTRIBUT PAR PSEUDO-VALEUR

➢ Retour par pseudo valeur

Accesseur d'une classe retournant un attribut qui est un objet : on utilise une **const reference** pour **simuler un retour par valeur.**

```
class A {  string s; // attribut objet string
public :
    // retour par copie : eviter !
    string getS1() const { return s ; }

    // retour par pseudo valeur : OK !
    const string & getS2() const { return s; }
} ;

main() {
    A a ;
    // c'est une COPIE de a.s qui est affichée !
    cout << a.getS1() <<endl;
    // pas de copie ! Bien !
    cout << a.getS2() <<endl;
    // la const reference interdit de modier a.s :
    a.getS2().erase() ; // INTERDIT par le compilateur :
                        // car a.getS2() est const !

    // tmp1 est une COPIE de COPIE de a.s !
    string tmp1 = a.getS1();
    // tmp2 est une copie de a.s
    string tmp2 = a.getS2();
}
```

Retour d'un attribut objet par const rérérence : tout se passe comme si on avait fait un passage par valeur, mais on évite la copie de l'objet :

- La **référence évite que l'objet soit copié** (la fonction travaille sur l'objet passé directement)
- Le mot clé *const* permet la **vérification syntaxique** (par le compilateur) de la non modification de l'objet retourné par la fonction.

DU BON USAGE DES REFERENCES A PHELMA (1)

➢ Référence et pointeurs sont deux concepts proches.

Presque tout ce qu'on fait avec l'un peut être fait l'autre.
Tout cela est affaire de **convention**.
Les us et coutumes des équipes de programmations et les « conventions de codage » jouent donc ici un rôle important.

A phelma...

Les **références** facilitent dans certains cas l'écriture et la lisibilité du code (**« cookie » syntaxique**) : quand on manipule une référence, on écrit le code **comme si on manipulait la variable d'origine**.

A l'inverse, l'utilisation d'un **pointeur** permet de marquer au niveau de la syntaxe une **indirection entre l'objet et le code dans lequel il est pointé**.

Choisir entre référence et pointeur : essentiellement suivant un critère de *lisibilité et facilité de compréhension* du code que l'on écrit.

➢ Opérateurs et références

Remarque: la notion d'opérateur sera vue ultérieurement.

Les références sont très utiles pour les opérateurs les plus usuels (surcharge d'opérateurs) et pour les collections.

C'est grâce à elles qu'on peut par exemple redéfinir l'opérateur [ ]. Sans référence, pas possible d'écrire des chose comme :

```
T[i] = 9 ; // affecte 9 à la 9eme case. T[i] retourne une référence
```

Mis à part ces cas, en général, on manipule surtout des const reference pour pour faire du passage par pseudo-valeur.

DU BON USAGE DES REFERENCES A PHELMA (2)

➢ Paramètres des fonctions

- **Fonctions travaillant sur des objets en paramètres sans les modifier :**
  => passage par « pseudo valeur » avec des paramètres const référence
- **Fonctions modifiant des objets passés en paramètres :**
  => passage par *pointeurs non const* comme en C

```
// correct : const & pour passage par pseudo valeur
 string addString(const string & a, const string & b) {
      string tmp = a + b;      return tmp;
 }

// A éviter : copie des objets string passés en paramètre !
 string addString(string a, string b) {
      string tmp = a + b;      return tmp;
}
// correct : ptr non const pour paramètre modifié
void modifierChaine(string * p_chaine) {
    *p_chaine = "nouvelle valeur" ; modifie * p_chaine
}
// A éviter :
// 1/ on ne voit pas que chaine n'est pas une variable locale
// mais une référence sur une variable hors de la fonction.
// 2/ lors de l'appel de la fonction on ne voit pas que
// la fonction va modifier la variable passée en paramètre
void modifierChaine(string & chaine) {
    chaine = "nouvelle valeur" ; //modifie la chaine !
}
// correct : pas de référence pour les types simples
void afficherInt( double v ) {
    cout << "la valeur est : " << v <<endl ;
}
// Absurde ! Inutile et plus couteux que sans référence !
void afficherInt( const double & v) {
    cout << "la valeur est : " << v <<endl ;
}
```

Exception : par habitude et convention, exception pour les **swap** de variables :

```
swap( int& a, int& b) { int inter=a; a=b; b=inter; }
```

Remarque : pas de référence sur des paramètres de types de base.

DU BON USAGE DES REFERENCES A PHELMA (3)

➢ Retour d'un attribut par une fonction membre

- **Méthode d'une classe qui retourne la valeur d'un objet attribut :**
  ⇨ retour par pseudo-valeur, avec une const reference en type de retour

- **Méthode d'une classe qui *donne accès* à un attribut :**
  ⇨ A éviter : rompt le principe d'encapsulation !
  ⇨ Si jamais... retour par adresse, avec un pointeur non const en type de retour

Remarque : pas de retrour par référence sur les attributs de type de base (int, float...)

Rappel : ne jamais retourner un pointeur ou une référence sur variable locale !

```
class BiString { // une classe composée de 2 string
private :
    string s1, s2 ;        int test ;
public :
    // retour par pseudo-valeur d'un attribut
    const string & getS1() const { return s1 ; }

    // Retour de l'adresse d'un attribut
    // pour permettre sa modification hors de la classe
    string * getS1ptr() { return & s1 ; }

    // retour par copie pour les types de base
    int getTest() const { return test ; }

    // Attention aux variables locales
    const string & getS1_concat_S2() const {
      string tmp = s1 + s2 ;
      return tmp ; // Erreur grave !
    }
} ;
main() {  BiString bs; // …
    cout << bs.getS1() << endl;
    bs.getS1ptr()->erase(); // efface bs.s1 !
}
```

MEMBRES AMIS : FRIEND

Les **fonctions** ou les **classes** déclarées **amies** dans la **déclaration** d'une classe peuvent accéder aux membres privés de cette classe

```
class Complexe {
    // Déclaration des fonctions et classes amies
    friend ostream & operator<<(ostream &,
                                const Complexe &);
    //operator<< avec un complexe comme argument
    // peut accéder à im et re;

    friend class X;
    // Toutes fonctions membres de X peuvent accéder à
    // im et re

private:
    float re, im; // PRIVES
public:
    void aff() ;
};

void Complexe::aff() { cout << re <<im; }

// Définition de la fonction amie operator<<
ostream & operator<<(ostream &s, const Complexe & z) {
    s << z.re << z.im;      // Autorisé car
    // ostream & operator<<(ostream &s,const Complexe& z)
    // est une fonction amie de Complexe.
    return s;
}

// Définition de la classe X
class X {
    ...
public:
    void essai(Complexe * a) { a->re=0; a->im=15; }
    // Autorisé car X est une classe amie de complexe.
};
```

CONSTRUCTEUR (1)

Constructeur d'une classe : fonction membre particulière qui initialise l'objet au moment où celui ci est instancié.

➢ Rôle des constructeurs

Contrôle automatique et systématique de l'etat initial de l'objet (attributs)

- Initialiser les champs,
- Réaliser systématiquement certaines actions,
- Déclencher les constructeurs des objets qui composent l'objet créé
- Allouer la mémoire si nécessaire

A la création d'un objet, **UN** constructeur est **toujours automatiquement** exécuté.

Remarques :

- un constructeur peut être appelé directement pour créer un objet temporaire non nommé (valeur de retour de fonction par exemple)

➢ Syntaxe

Fonction membre de même nom que la classe, sans valeur de retour

```
class Fred { ... .... ...
public:
    // Un constructeur :
    Fred(int toto, double titi) { .... }
};
```

➢ Surcharge

- Plusieurs constructeurs avec des paramètres et des types différents i.e. plusieurs manières d'initialiser l'objet.
- Mêmes règles d'appel que pour les autres appels de méthode (correspondance exacte, conversions du langage, conversion utilisateur)
- Lors de la création d'un objet en mémoire, il faut qu'il existe un constructeur correspondant dans la classe.
- On peut utiliser le mécanisme de valeurs par défaut pour les arguments.

CONSTRUCTEUR (2)

➢ Constructeur par défaut

Un **constructeur par défaut** est un constructeur sans paramètre.

Un **constructeur par défaut** est créé automatiquement par le compilateur si aucun autre constructeur n'est déclaré. Ce constructeur par défaut ne fait *rien* ; il n'initialise même pas les attributs à des valeurs par défaut !

Dès qu'un autre constructeur est déclaré, ce constructeur par défaut n'existe plus.
Conseil : toujours définir au moins un constructeur dans chaque classe **!**

Il est possible de définir soi même un contructeur par défaut qui remplace le constructeur par défaut fourni par le compilateur :

```
class Fred {
    int i ;
public:
    Fred ( )  {
      // permet d'initialiser un objet Fred par défaut
      i = 9 ; // par exemple
    }
} ;
```

➢ Constructeur de copie

Role : Copie la valeur d'un objet dans un nouvel objet

Appelé en particulier lors du passage par valeur de paramètres, pour le retour de valeur des fonctions.

C++ crée un **constructeur de copie par défaut**, qui copie les valeurs des attributs de l'objet copié dans le nouvel objet.
Il est possible de redéfinir ce contructeur de copie avec un constructeur de copie de votre choix :

```
class Fred {
public:
    Fred ( const Fred & other)  {
      // cosntructeur de copie de thread
    }
} ;
```

CONSTRUCTEUR (3)

Exemple 1

```
class Complexe {      float re, im;
public: // trois constructeurs
    Complexe(double a, double b=0.){ re=a ; im=b ; }//(1)
    Complexe(int a, int b) { re=a; im=b;}            //(2)
    Complexe(const Complexe& a){re=a.re; im=a.im;}   //(3)
};
main() {
    Complexe w(2.0, 3.0);                            //(1)
    // eq. à  Complexe  w = Complexe (2.0,3.0);
    Complexe x(1. );                                 //(1)
    // eq. à Complexe x(1. , 0. );
    Complexe y(4, 5);                                //(2)
    Complexe z(y);                                   //(3)
    // idem Complexe z = y; Ce n'est pas l'opérateur =
    Complexe t ;        // INTERDIT : pas de constructeur
    Complexe *p ;  // Pas de constructeur appelé
    p=new Complexe (5.0, 6.0); // alloc dynamique    //(1)
}
```

Exemple 2

```
class Chaine { char *p; int t;
public:
    Chaine (char *s) {
          strcpy(p=new char[t=strlen(s)+1],s);
    }
    /* Identique à
     chaine (char *s) { t=strlen(s)+1; // taille
     p=new char[t];          // Allocation mémoire
     strcpy(p,s); }          // copie de s dans l'objet
     */
    Chaine(const chaine& s){
          strcpy(p=new char[t=s.t],s.p);
    }
};
main() {
    Chaine c1("Voici une chaîne");
    Chaine c2 (c1); // ou chaine c2=c1;
    // que se passe-t-il en mémoire ?
}
```

CONSTRUCTEUR (4)

➢ Liste d'initialisation dans les constructeurs

La liste d'initialisation d'un constructeur est une partie du code du constructeur
- placée après son prototype et ":"
- et avant le bloc de code entre accolades { }

La liste d'initialisation permet d'initialiser les attributs de l'objet construit.

Syntaxe:

```
class X {
private:
    string toto;
    int nb ;
    double *tab;
public:
    X(const string &valeur):toto(valeur),nb(0),tab(NULL){
          cout << "appel de X(const string &)";
    }


    X(int _nb):toto(""),nb(_nb),    tab(new double(_nb)){
          cout << "appel de X(int _nb)";
    }
};
```

Les attributs doivent être initialisés dans l'ordre de leur déclaration dans la classe.

Conseil :
- De préférence, toujours recourir aux listes d'initialisation : optimalité, systématisme
- prendre garde à ce que chaque constructeur initialise systématiquement tous les attributs explicitement.

DESTRUCTEUR

Destructeur : fonction membre qui est exécutée quand l'objet sur lequel elle travaille est détruit :

- variables locales, paramètres de fonction : fin de bloc (bloc = { ... })
- variables globales : fin de programme
- Si allocation dynamique avec ***new***, lors de la destruction avec ***delete***

➢ Rôle du destructeur

Réaliser systématiquement certaines actions, comme :

- Libérer la mémoire si nécessaire.
- Libérer les objets qui composent l'objet détruit.

➢ Syntaxe

```
class Fred { ... .... ...
    public:    ~Fred() { .... } // Le destructeur
};
```

- Le destructeur n'est pas obligatoire. On ne l'écrit que quand on en a besoin.
- Un destructeur n'a pas de paramètre ni de valeur de retour.
- Un seul destructeur par classe.

➢ Exemples

```
class Complexe {      float re, im;
public: // le destructeur :
    ~Complexe() { cout << "il n'y a rien a faire !";}
};
main() {
    Complexe z();
}     // Le destructeur est appelé à ce moment pour z
      // Affiche "il n'y a rien a faire"

class Chaine { char *p;int t;
public:
    Chaine (char *s) {strcpy(p=new char[strlen(s)+1],s);}
    ~Chaine() {delete [] p; p=NULL;} // destructeur
};
main() {
    Chaine str("Voici une chaîne");
}   // str est détruit à ce moment
```

MEMBRES DE CLASSE ; STATIC (1)

➢ Attribut de classe

- Attribut relatif à la *classe* elle même et non à chaque *objet*
- valeur partagée par toutes les instances/objets de la classe.

Exemples :

- valeur constante
- compteur du nombre d'instances d'une classe donnée.
- etc.

➢ Syntaxe. Instanciation des attributs de classe

```
static <déclaration d'attribut>

class Complexe {
public:
    static int nbInstances; // DECLARATION
    // nbInstances est un attribut de classe.
    // Il n'est alloué qu'une fois en mémoire,
    // au lancement du programme.
    // Toutes les intances de la classe partagent
    // cet attribut.
    // Il est attaché à la classe, pas à ses instances
    // Il accessible par la classe elle même
    // ou par chacun des objets de la classe
...
}
```

➢ Initialisation des attributs de classe

- Création au lancement du programme.
- Initialisation explicite, comme les variables globales en C.

En général dans le .cpp de la classe :

```
// Dans le fichier source .cpp
 int Complexe::nbInstance = 0; // INITIALISATION
```

MEMBRES DE CLASSE ; STATIC (2)

➢ Notion de méthode de classe

Méthode qui existe indépendamment des instances de la classe.
Elle n'a accès qu'aux attributs de classe (static), mais pas aux autres attributs.
Déclarée également avec ***static***.

➢ Accès aux méthodes et attributs de classe

Deux syntaxes :

- syntaxe habituelle si on dispose d'une instance de la classe :

```
Complexe c ;
    cout << c.nbInstances;
```

- comme attributs et méthodes de classe sont relatifs à la classe, et pas aux objets, on peut aussi y accéder sans objet avec **<NomDeClasse>::**

```
    cout << Complexe::nbInstances;
```

➢ Exemples d'usages

- Compteur d'instances, comme dans l'exemple précédent.
- Constante de classe (=>valeur est partagé par toutes les instances).

```
class Voiture {
    static const int NB_PORTES=4 ;
    // remarque : un membre static const
    // peut être initialisé dans la déclation
} ;
```

➢ Notion de classe utilitaire

Classe dont tous les membres sont statiques.
Permet de regrouper des algorithmes d'usage courant sous un nom unique.

```
class TimeConvertTool {
    static const int MIN_PER_HOUR = 60 ;
    static double minToHour(int nbMin) {
        return ((double) nbMin) / MIN_PER_HOUR ;
    }
    static int hourToMin(int nbHour) {
        return nbHour * MIN_PER_HOUR ;
    }
} ;
main() {  cout << TimeConvertTool::secToHour(3600) ; }
```

MEMBRES DE CLASSE ; STATIC (3)

➢ Exemple : compter le nombre d'instances d'une classe Etudiant

```
class Etudiant {
private:
     static int s_nbInstances; // attribut de classe
     int age;   string nom ;
public:
     static double s_test ; // un attribut de classe public

     Etudiant() { s_nbInstances++ ; }
     Etudiant(const string & nom, int age) {
          this->nom = nom;          this->age = age;
          s_nbInstances++ ;
     }
     // constructeur de copie
     Etudiant(const Etudiant & other) {
          this->nom = other.nom;    this->age = other.age;
          s_nbInstances++ ;
     }
     ~Etudiant() { s_nbInstances-- ; }  //Destructeur

     void setNom(const string & nom) {this->nom = nom ;}
     const string & getNom() const { return nom ; }
     static int getNbInstances() { return s_nbInstances;}
} ;


// instanciation et initialisation des attributs de classe
// Dans le fichier source .cpp
int Etudiant::s_test = 0;
double Etudiant::s_nbInstances = 0;

main() {
     Etudiant a("Paul", 21), *pb = new ("Sylvie", 20) ;
     cout << a.getNbInstances() <<endl;          // affiche 2
     cout << Etudiant::getNbInstances() <<endl;  // Pareil !

     a.s_nbInstances = 9 ;              //INTERDIT: membre privé!
     Etudiant::s_nbInstances = 2 ;      //INTERDIT , idem

     Etudiant::s_test = 8.3 ;           // OK : s_test est public
     cout << a.s_test <<endl;                    // affiche 8.3

     delete pb ;
     cout << Etudiant::getNbInstances() <<endl;  // affiche 1
}
```

SURCHARGE D'OPERATEURS (1)

La plupart des opérateurs peuvent être surchargés et avoir ainsi une signification particulière pour une classe donnée.

L'action de la fonction n'est pas obligatoirement identique à celui de l'opérateur initial (+ peut ne pas être une addition)

Un opérateur conserve son arité (ie le nombre d'opérandes)

Un opérateur binaire (unaire) est implanté soit par une fonction membre avec un (zéro) paramètre, soit par une fonction amie avec 2 (1) paramètres, mais pas les deux à la fois.

➢ Syntaxe

```
operator xxx      où xxx est un symbole parmi
+ - * / % ^ & | ~ ! << >> = != == || && ++ -- [] ()
new delete toutes_les_affectations_composées (+= ..)
tous_les_opérateurs_relationnels (< <= ...)
```

➢ Exemple

```
class Complexe {
friend Complexe operator+ (   const Complexe&,
                              const Complexe&);
private: float re,im;
public:
   Complexe operator- (const Complexe&) const;  // *this-y
   Complexe operator- () const;                 // -y
};
Complexe Complexe::operator- (const Complexe & b) const {
   Complexe r; r.re=re-b.re; r.im=im-b.im; return r;}
Complexe Complexe::operator- () const {
   Complexe r; r.re=-re; r.im=-im; return r;}
Complexe operator+ (const Complexe &a, const Complexe & b) {
   Complexe r; r.re=a.re+b.re; r.im=a.im+b.im; return r;
}

main() { Complexe x,y,z;  z = x+y; x = -y; y = x-z; }
```

SURCHARGE D'OPERATEURS (2)

Rq: opérateurs non surchargeables :

```
::.     .*      sizeof   ?:
```

```
.* = pointeur sur membre
```

Rq: les opérateurs `= [] () ->` doivent être des fonctions membres non statiques et non des fonctions amies

Rq: les fonctions opérateurs peuvent être appelées explicitement, bien que cela ne soit pas l'usage courant

Exemple: `z = a.operator+(b); // idem z=a+b;`

Rq: les opérateurs `++ et --` sont l'incrémentation et la décrémentation. Distrinction entre entre pré (++i) et post (i++) incrémentation : ce sont deux opérateurs distincts

```
class X {
public:
  X& operator++();
        // Pré-incrémentation : fonction appelée pour
++a
  X& operator++(int); // paramètre int Obligatoire
        // Post-incrémentation : fonction appelée pour
a++
};
main() {
  X b;
  ++b; // b.operator++()      :   1ère  fonction
  b++; // b.operator++(0)     :   2ième fonction
}
```

Conseils avec les opérateurs

- Pour tous les opérateurs usuels (affectation, crochet...) pensez à utiliser les signatures conseillées...
- Visez la complétude, eg opérateur =, deux versions de l'opérateur [] ...

SURCHARGE DE ->

** Classe qui se comporte comme un pointeur.

** Contrôle d'accès à un membre à travers les pointeurs (problème de typage statique & dynamique) : debogage, protection, réflexe

** Opérateur **unaire** de nom : `operator->()`

** Syntaxe d'appel : `expression->nom_de_membre`

** Evalué comme : `(expression.operator->())->nom_de_membre`

** Doit retourner un **pointeur** ou une **référence sur un objet comportant un mebre**

Exemple:

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

class X {  int data;
  public:
  X(int a) : data(a) {};
  void f() { cout << "Appel de f :"<<++data<<endl;  }
};

class Ptr{   X* pointeur;
  public:
  Ptr(X* p=NULL) : pointeur(p) {}
  X* operator->() { cout << "Point. intelligent"<< endl;
    if (pointeur) return pointeur;
    else { cout << "Erreur: non valide:"<<endl;
      exit(1);
    }
  }
};

main() { X x1(2);
  Ptr p1(&x1), p2;;
  p1->f();   p2->f();  p1->f();
}
```

SURCHARGE DE [ ]

Opérateur **binaire** de nom : `<type> operator[](int)`
Utilisé pour trouver un élément dans une collection, e.g. : `t[i]`

Pour écrire : `t[i]=0;` [ ] retourne une **Lvalue** (un objet) et non une valeur

En général en deux versions :

- version *const* qui retourne une copie ou une référence const *pour appeler [ ] sur un objet const*
- version *non const* qui retourne une référence non const *pour écrire* `t[i]=0;`

➢ Exemple

Le programme suivant affiche « v1 D v2 v2 Z »

```
class Chaine {    char *p;
public:
    Chaine(const char *s) {  p=strdup(s);  }

    // V1 -  Operateur[ ]const. Retourne copie ou const &
    // const char &  operator[](int i) const { ...}
    char operator[](int i) const {
      cout << "v1" << endl;     return p[i] ;
    }

    // v2 -  Operateur[ ]non const. Return une const & !
    char & operator[](int i) {
      cout << "v2" << endl;     return p[i];
    }
};

main() {
    const Chaine s1 ("ABSDEF");
    cout << s1[3] << endl; // Pas de problème : v1

    Chaine s2 ("ABSDEF");
    s2[3] = 'Z';           // Pas de problème : v2
    cout << s2[3] << endl; // Pas de problème : v2
}
```

CONVERSION UTILISATEUR

➢ Conversion via un constructeur

Exemple: conversion réel-> complexe

```
Complexe::Complexe (double reel) {re=reel; im=0; }
main() {
     complexe x;
     x = 2.0; // idem x = complexe(2.0);
}
```

Usage très limité :

- Conversion vers un type de base impossible
- Pas de différence construction-conversion

➢ Opérateur de conversion

Syntaxe : fonction membre operator de même nom que le type de base
Pas de type de retour précisé.

Exemple: conversion complexe -> réel

```
class Complexe {
    float re, im;
public:
    Complexe(double a, double b = 0. ) { re=a; im=b;}
    operator double () const    ;
};

Complexe::operator double () const { return(re); }

main() {
    Complexe x(2.0,3.0);
    double a;
    a = x; // a=2.0 :valeur retournée par la conversion
}
```

Remarque : il est possible de définir un tel opérateur de conversion vers non pas un type de classe mais un objet.

FONCTION AMIE *VS* FONCTION MEMBRE

Une **fonction** qui modifie l'état de l'objet sur lequel elle travaille doit de préférence être une fonction membre.

L'appel se fait par `id_variable.id_fonction(paramètres)`

Exemple : `x.setQuelqueChose(4);`

```
class complexe { float re,im;
  public :
  init(float a, float b) { re=a; im=b; }
  complexe operator+ (const complexe &); // Pour x+y
  complexe operator+ (double);   // Pour x+1.0
};
main() { complexe x; x.init(10,20); }
```

Rq: impossible de faire **`1.0+x`**

Une **fonction** commutative qui nécessite des conversions éventuelles de ses paramètres doit de préférence être une fonction amie.

L'appel se fait par `id_fonction(id_variable,paramètres)`

Exemple : `affiche2(x);`

```
class complexe { float re,im;
  public :
  friend complexe operator+ (const complexe &,
                             const complexe &);
  // Pour x+y
  friend complexe operator+ (const complexe&,double);
  // Pour x+1.0
  friend complexe operator+ (double,const complexe&);
  // Pour 1.0+x
};

complexe operator+ (complexe a, complexe b) {
  complexe r; r.re=a.re+b.re; r.im=a.im+b.im; return r;}
main() { complexe x,y,z;
     z=x+y; x=y+1; y=2+z; }
```

CYCLE DE VIE DES OBJETS (1)

CONSTRUCTION, INITIALISATION, AFFECTATION

➢ Cycle de vie d'un objet

Ensemble des états par lequel il passe au cours de l'exécution d'un programme :
- depuis sa **création**
- jusqu'à sa **destruction**,
- en passant par son **initialisation**, sa modification, sa copie, etc.

Bien maîtriser ce qui se passe durant la vie d'un objet :
- Creation et initialization d'un objet
- Copie d'un objet dans un autre objet de même type
- Affectation d'un objet dans un autre objet de même type (avec =)
- Destruction d'un objet, etc.

➢ Rappel : constructeur et destructeur

Constructeur : fonction membre appelée automatiquement lors de la création d'un objet :
- par une définition : `chaine a("azert");`
- par une allocation dynamique :

```
chaine *pa= new chaine ("azert");
```

Destructeur : fonction membre appelée automatiquement lorsqu'un objet est détruit :
- fin d'un bloc { X x ; } pour une variable locale
- fin du programme pour les variables globales
- appel explicite de **delete** pour un objet alloué dynamiquement avec **new**

➢ Rappel : constructeur par défaut du C++

Le C++ fournit un constructeur par défaut, sans paramètre.

- initialise tous les attributs à leurs valeurs par défaut
  - (entier et flottant à 0, pointeurs à NULL, objets avec leur constructeur par défaut...)
- n'existe plus dès qu'on définit un constructeur pour la classe
- peut être redéfini.

CYCLE DE VIE DES OBJETS (2)

CONSTRUCTION, INITIALISATION, AFFECTATION

➢ Rappel : constructeur par copie : `X(const X&)`

Constructeur qui réalise une copie d'un objet de classe X dans un autre objet de classe X.

Appelé :

- définition :

```
chaine a("azert"), b(a);
```

- passage de paramètres par valeur :

```
void f (X a) { /* ••• */ };
main() { ••• f(x); •••} // Copie de x dans a
```

- valeur de retour d'une fonction :

```
X f () { X a; /* ••• */ return a; };
main() { X x; •••x=f(); •••}
                    // Copie de a dans x
```

Constructeur de copie par défaut:

appelle constructeur de copie sur tous les attributs de l'objet copié.

Pour les attributs scalaires, copie bit à bit

Pour les attributs objets, execute le constructeur par copie

➢ Opérateur d'affectation : `X& operator=(const X&) :`

Opérateur membre qui affecte l'objet passé en paramètre à ***this**.

Fonction membre exécutée lors de l'utilisation de **=**, dans un code:

```
main() { X a=2,b=3   ;  a=b;}
```

Opérateur d'affectation par defaut

appelle l'opérateur d'affectation pour chacun des attributs de l'objet affecté.

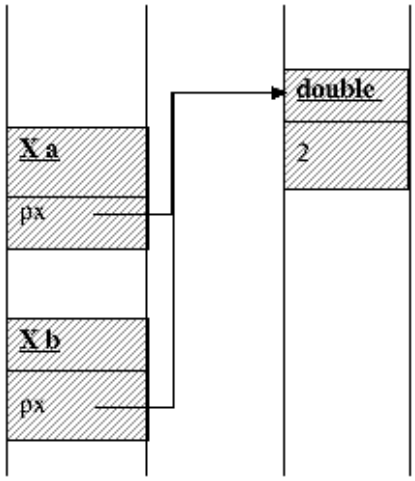
CYCLE DE VIE DES OBJETS ET ALLOCATION DYNAMIQUE (1)

Quelles sont les erreurs commises dans cette classe X ?

```
class X {
    friend X operator+(const X & a, const X & b);
    double * px;
    X()             { px = NULL } // privé
public :
    X(double x)     { px = new double (x); }
    ~X()            { delete px; }
    void setDbl(double r)   { *px = r ; }
    void aff() const        { cout << *px << endl ; }
};
X operator+(const X & a, const X & b) {
    X r;
    // Concaténation de 2 X
    r.px = new double( *a.px + *b.px ) ;
    return r;
}
void affX(X x) { cout << * x.px<<endl; }
```

➢ Exemple 1

```
main() {
    X a(2);
    X b(a); // a.px == b.px !
    b.setDbl(4);
    a.aff() ; // affiche 4 !
}   // destruction de a (libère a.px)
    // et de b (libère b.px : ERREUR)
```



➢ Exemple 2

```
main() {
    X a(2);
    X b = a;   // b.px = a.px : partage de pointeurs !
 // Toute modification de *a.px ou *b.px modifie l'autre

}   // destruction     de b (libère b.px)
    //                 de a (libère a.px : ERREUR)
```

CYCLE DE VIE DES OBJETS ET ALLOCATION DYNAMIQUE (2)

➢ Exemple 3

```
main() {
    X a(2);
    affX(a);   // Copie de a dans x : x.p=a.p
    // Après exécution : libère x.p !
    •••
    // Impossible d'utiliser a.p car il est déjà libéré !
}   // destruction     de a (libère a.p) :  ERREUR
```

➢ Exemple 4

```
main() {
    X a(1);    X b(2);
    X c = b+a; // Copie de r dans un objet temporaire tmp
    // Copie de tmp dans c
    // Libère r.px car locale à operator+()
    // Impossible d'utiliser c.px
    // car il est déjà libéré à travers r !
}   // destruction de c (libère c.p : ERREUR)
```

➢ Conclusion

Encapsulation des allocation/libération mémoire : toute classe qui alloue de la mémoire doit se préoccuper :

* de la destruction de la mémoire allouée
* de ce qui se passe quand des instances sont copiées ou affectées

En particulier une classe avec allocation dynamique doit comporter les fonctions suivantes :

- **constructeur par défaut**
- **constructeur par copie (appel lors du passage de valeur et retour de fonction)**
- **destructeur**
- **opérateur d'affectation**

CYCLE DE VIE DES OBJETS ET ALLOCATION DYNAMIQUE (3)

Reprenons pour conclure notre classe X :

```
class X {

    friend X operator+(const X & a, const X & b);

    double * px;
    X()             { px = NULL } // privé

public :

    X(double x)     { px = new double (x); }

    // Constructeur de copie
    X( const X & other)     {
      px = NULL ;
      if(other.px()
          px = new double ( * other.px );
    }

    // Destructeur
    ~X()            { delete px; }

    // Opérateur d'affectation. Affecte y à *this
    X& X::operator=(const X& y){
      if (this == &y) return *this;
      if (px) delete px;
      px = NULL ;
      if( y.px )
          px = new double ( * y.px );
      return(*this);
    }

    void setDbl(double r)   { *px = r ; }
    void aff() const        { cout << *px << endl ; }
};
```

SURCHARGE DE L'OPERATEUR = ET *SELF ASSIGNEMENT*

➢ Le problème du self-assignement

Reprrenons la classe X et considérons l'opérateur d'affectation:

```
X& X::operator=(const X & y){
     if (px) delete px;
     px = NULL ;
     if( y.px )
          px = new double ( * y.px );
     return(*this);
   }
main() {
     X x;
     x = x;
}
```

Que se passe-t-il ?

➢ Deux canevas pour l'opérateur d'affectation =

Canevas 1 : verifier que y n'est pas *this:

```
X& X::operator=(const X& y){
   if (this == &y) return *this;
   if (px) delete px;
   px = NULL ;
   if( y.px )
        px = new double ( * y.px );
   return(*this);
  }
```

Canevas 2 : copier d'abord les attributs

```
X& X::operator=(const X &y) {
   double *tmp = NULL; // temporaire
   if( y.px) tmp = new double ( * y.px ) ;
   if (px) delete px; // delete des attributs
   px=tmp;
   return *this;
  }
```

EXEMPLE : UNE CLASSE CHAINE DE CARACTERES (1)

```
class chaine {
    int t;          // Le nb de char alloues, y compris '\0'
    char *p;        // La chaîne stockée
public:
// constructeurs, destructeur, opérateur =
    chaine();                       // chaine x; (vide)
    chaine(const char *s);          //chaine x="ab" & chaine("ab")
    chaine(const chaine & y);  //chaine x(y), return(x)...
    chaine (char c, int n = 1);// n caractères c
    ~chaine ();                     // Destructeur
    chaine& operator=(const chaine&);// x=y;
    chaine& operator=(const char *); // x="abc" :

// accesseurs et modifieurs
    bool isNull() const { return t == 0; }
    // nb de caractères sans '\0' ; -1 si isNull()
    int length() const { return t-1 ;}
    // => isNull() devient true
    void clear() { delete [ ] p; p = NULL; t = 0; }
    //opérateurs []
    char  operator[](int i) const;// cout << x[i] ;
    char& operator[](int i);   // x[i]='a'; cin >> x[i]
    // conversion chaîne -> en const char *. A eviter
    operator const char* () const { return p; }

    // sous chaine débutant a pos et de taille len
    chaine substr (int pos = 0, int len = -1) const;
    // recherche de chaine
    int find (const chaine& s, int pos = 0) const;
    // concaténation "en place"
    chaine& operator+= (const chaine& str);

// Fonctions amies
    // égalité de 2 chaînes
    friend int operator==(const chaine& ,const chaine& );
    // concaténation
    friend chaine operator+(const chaine& ,const chaine& );
    // E/S
    friend ostream& operator<<(ostream&, const chaine &);
    friend istream& operator>>(istream&, chaine &);
};
```

EXEMPLE : UNE CLASSE CHAINE DE CARACTERES (2)

```
//******* Constructeurs, destructeur, opérateur =
chaine::chaine (): t(0), p(NULL) { }

chaine::chaine (const char *s): t(0), p(NULL) {
    if(s) { strcpy(p=new char[ t=strlen(s)+1 ], s); }
}

chaine::chaine (const chaine & y): t(0), p(NULL) {
    if(y.p) { strcpy(p=new char[ t=y.t ], y.p); }
}

chaine::chaine (char c, int n): t(n), p(NULL) {
    if ( n < 0 )  { cerr<<"Erreur A"; exit(1); }
    p = new char[ n + 1 ];
    memset(p, c, n);        p[n] = '\0' ;
}

chaine::~chaine () { delete [ ] p; }

chaine& chaine::operator=(const chaine& y) {
    if(this == &y) { *this; }
    if (t) delete [] p;
    if(y.t>0) { strcpy(p=new char[ y.t ], y.p); }
    else { t = 0; p = NULL ; }
    return *this ;
}

chaine& chaine::operator=(const char  *s) {
    if (t) delete [] p;
    if(s) { strcpy(p=new char[ t=strlen(s)+1 ], s); }
    else { t = 0; p = NULL ; }
    return *this ;
}

//******* accesseurs et modifieurs
char & chaine::operator[](int i) {
    if (i<0 || i>t) { cerr<<"Erreur B"; exit(1); }
    return p[i];
}
char chaine::operator[](int i) const {
    if (i<0 || i>t) { cerr<<"Erreur C"; exit(1); }
    return p[i];
}
```

EXEMPLE : UNE CLASSE CHAINE DE CARACTERES (3)

```
// retourne sous chaine commencant a pos et de taille len
chaine chaine::substr (int pos, int len) const {
    if ( pos < 0 || pos >= t) {
        cerr<<"Erreur D"; exit(1);
    }
    // correction de len
    if(len < 0) len = t;
    if(pos+len > t) { len = t-pos; }
    if(len <= 0) return chaine();
    chaine s;
    s.t=len;
    strncpy( s.p = new char[ len +1 ], p + pos, len) ;
    s.p[len] = '\0'; // null-terminated
    return s;
}

// recherche de chaine. Retourne -1 si pas trouvée
int chaine::find (const chaine& s, int pos) const {
    if(pos < 0)   { cerr<<"Erreur E"; exit(1); }
    if(pos >= t) { return -1; }
    char * found =  strstr( p+pos, s.p) ;
    if(found == NULL) return -1;
    return found - p ;
}


chaine& chaine::operator+= (const chaine& y) {
    if(y.p == NULL || y.p[0] == '\0') { return *this ; }
    int newT = t + strlen(y.p) ;
    char * tmp = new char[ newT ] ;
    if(t) {
        strcpy( tmp , p) ;
        strcpy( tmp + t-1, y.p);
    } else {
        strcpy( tmp , y.p) ;
    }
    t =  newT;
    delete [ ] p;
    p = tmp;
    return *this;
}
```

EXEMPLE : UNE CLASSE CHAINE DE CARACTERES (4)

```
//******* Fonctions et opérateurs amis
ostream& operator<<(ostream& s, const chaine & x) {
    if (x.p) { s << x.p; }
    return s;
}

istream& operator>>(istream& s, chaine & x) {
     // Attention : pas de test de la longueur
    char buf[256];
    s >> buf;
    x=buf;
    return s;
}

int operator==(const chaine& x,const chaine& y) {
    if( x.p == NULL && y.p == NULL) return 1;
    if( x.p == NULL || y.p == NULL) return 0;
    return strcmp(x.p, y.p)==0;
}

chaine operator+(const chaine& a,const chaine& b) {
    chaine r;
    r.t = a.t + b.t -1;         // -1 : un seul '\0'
    if(r.t<0) r.t = 0;// cas a et b null
    if(r.t) {
     r.p = new char[ r.t ] ;
     if(a.p) {
          strcpy( r.p , a.p) ;
     }
     if(b.p) {
          if(a.t>0) {
               strcpy( r.p + a.t-1, b.p);
          } else {
               strcpy( r.p, b.p);
          }
     }
    }
    return r;
}
```

EXEMPLE : UNE CLASSE CHAINE DE CARACTERES (5)

```
//******* Test
chaine f(chaine x, chaine y) { r = x+y; return r; }

main() {
   chaine a("chaine 1");                  // Constructeur 2
   cout << "a = "<< a << endl;      //a = chaine 1
   printf("a = %s\n", (const char*) a );    //a = chaine 1

   chaine x[100];           // Tab 100 chaînes "null"
   cin >> x[0];         // operateur >>
   x[1] = x[0];         // operateur affectation
   x[1] = x[1];         // self assignement
   x[2] = chaine('a', 12);// Constructeur 3
   x[2][3] = 'B';        // operateur [] non const
   x[3]=f(x[0],x[1]); // Concaténation, passage par valeur
   x[4]="salut"   ;     // operateur =(const char *)
   x[4]+= " veut dire bonjour" ;   // op += et conversion
   x[5] = x[4].substr(6, 4);        // substr()

   for(int i = 0 ; i < 5 ; i ++ ) {
     cout << "x["<<i<<"] = " << x[i] << endl;
   }

   cout << "x[3].length() = " << x[3].length() << endl;
   cout << " isNull() ? " << chaine("").isNull() << endl;
   cout << "find " << x[4].find("dire", 5) << endl;
   cout << "a == a "<< (a == a) <<endl;
   cout << "vide==null ? : "<<(chaine("")==chaine())<<endl;
} // Appel des destructeurs ici
```

➢ Sortie

On suppose que l'utilisateur tape 'A'

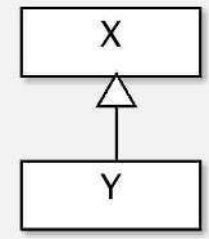
```
a = chaine 1                    x[4] = salut veut dire bonjour
a = chaine 1                    x[5] = veut
A                               x[3].length() = 2
x[0] = A                        isNull() ? 0
x[1] = A                        find 11
x[2] = aaaBaaaaaaaa             a == a 1
x[3] = AA                       vide==null ? : 0
```

5. HERITAGE, POLYMORPHISME, ABSTRACTION ET VIRTUALITE EN C++

➢ Héritage

Une classe peut hériter des propriétés (attributs et méthodes) d'une autre classe.

On dit que :
- Y hérite de, dérive, étend ou spécialise X
- Y est une clase fille ou sous classe de X
- X est une classe mère ou super classe ou classe de base de Y



On dit également :
- toute instance de Y *est aussi* ou *peut être manipulé comme* une instance de X

➢ Syntaxe

```
class <classe-dérivée> : <public,private,protected>
      <classe-de-base> {
      champs;
      fonctions membres;
};
```

Exemple :

```
// Y dérive de X
class Y : public X {
    ...
} ;
```

➢ Redéfinition d'une méthode

Une méthode définie dans une classe mère peut être *redéfinie* dans une classe fille (même signature).

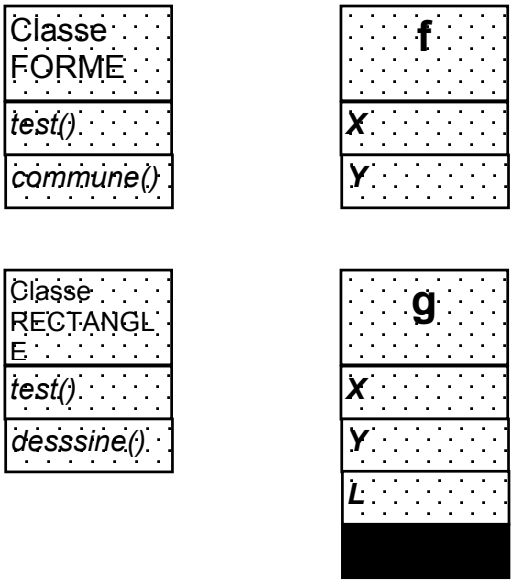
➢ Résolution de portée

Le code d'une méthode de la classe fille peut explicitement appeler une méthode de la classe mère avec la syntaxe :

```
<classe-base>::<nomDeMethode>()
```

EXEMPLE DE CLASSE HERITEE

```
class FORME {
    int X;
    int Y;
public:
    void aff()          { cout << "FORME" << X << Y ; }
    void commune()      {•••}
};

class RECTANGLE : public FORME {// RECTANGLE hérite de forme
    int L;
    int l;
public:
    void aff()     { FORME::aff() ; cout << "RECT" << L ;••• }
    void dessine()     {•••}
};

main() {
    FORME f;
    f. aff();       // FORME::aff()
    f.commune();        // FORME::commune()
    f.dessine() ;       // ERREUR :
                        //pas de méthode dessine() dans FORME
    RECTANGLE g;
    g.aff();        // RECTANGLE::aff ()
    g.commune();    // FORME::commune()
    g.dessinne();   // RECTANGLE::dessine()
}
```



POLYMORPHISME

➢ Polymorphisme

Capacité à voir un objet sous des formes différentes
En pratique, permet de manipuler un objet par un pointeur ou une référence d'une classe mere.

Le CERCLE c est manipulé via pf « comme » de FORME.

➢ Exemple

```
main() {
    FORME f, *pf, f2;
    CERCLE c, *pc, c2;
    f.aff();      // FORME::toString()
    f2 = c;                // OK ! Que se passe-t-il ?
        //1/ cast RECTANGLE -> FORME
        //2/ opérateur=(const FORME&)
        //   (opérateur d'affectation par défaut) de FORME !
        //Attention : f2 est un autre objet en mémoire.
    c2 = f;                // INTERDIT

    pc = &c;
    pc->aff();             // RECTANGLE::aff()
    pf = (FORME *) &c;     // OK un RECTANGLE est une FORME
                           // Le cast n'est pas obligatoire
    pf->aff();             // FORME::aff()

    pc = (CERCLE *) f;     // INTERDIT !
}
```

➢ Type statique, type dynamique

Type statique: type de la variable, de la réference ou du pointeur utilisé pour manipuler l'objet
Type dynamique: type effectif de l'objet manipulé en mémoire
:
Le type statique de pf est "pointeur sur FORME"..
Le type dynamique de l'objet pointé par pf est CERCLE.

CONVERSION CLASSE MERE <-> CLASSE FILLE

➢ Conversion entre classe fille et classe mère

Conversion **implicite** : lorsque la classe de base est dérivée publiquement, conversion implicite automatique entre un élément, un pointeur ou une référence à un élément de la classe dérivée vers la classe de base.

Une instance de la sous classe *« est »* instance de la super-classe.

La réciproque est fausse : un objet de base « *n'est pas* » un objet dérivé (un CERCLE est une FORME, mais une FORME n'est pas toujours un CERCLE).

➢ L'opérateur dynamic_cast< > : cast dans les hiérarchie de classes

Converti un pointeur sur un objet d'une classe de base vers un objet d'une classe de dérivée ou inversement.
Prend en compte le type dynamique (effectif) des objets manipulés.

- Si la conversion est possible, renvoie un pointeur valide
- Si la conversion est impossible, renvoie un pointeur nul sinon.

A éviter : souvent signe d'une mauvaise conception objet !!!

➢ Exemple

```
int main() {
   FORME f; RECTANGLE r; CERCLE c;
   FORME * tab_pf[4];
   tab_pf[0] = &f;
   tab_pf[1] = &r;
   tab_pf[2] = &c;
   tab_pf[3] = NULL;
   for (int i =0; i<4; i++) {
       CERCLE * pc  = dynamic_cast<CERCLE*> ( tab_pf[i] );
       if (pc)
          cout<<"CERCLE ; rayon=" << pc->rayon() << endl;
       else
          cout<<"L'objet pointé n'est pas un CERCLE"<<endl;
   }
}
```

Seul le troisième cast dynamique, sur tab_pf[2] renvoit un pointeur non NULL.
Tous les autres échouent (pointeur NULL).

HERITAGE, CONSTRUCTION, DESTRUCTION

➢ Construction et destruction d'un objet d'une classe dérivée

1. La mémoire est réservée pour stocker un objet de la classe dérivée.

2. Les objets sont construits et leurs constructeurs exécutés **de haut en bas** dans l'ordre de l'arbre d'héritage: l'objet de base, puis les membres de cet objet puis les aspects propres à la classe dérivée.

3. Les objets sont détruits dans l'ordre inverse : destructeur de la classe fille (destruction des membres de la classe fille) *puis* destructeur de la classe mère. L'appel du destructeur de la classe mère est automatique.

Remarque : le destructeur de la classe mère est en général déclaré virtual, cf ce qui suit.

➢ Exemple

```
class POINT { int X;   int Y;
public:
    POINT(int a, int b) {X=a; Y=b;}
    •••
};

class FORME { POINT origine;
public:
    FORME( int c, int d) : origine(c,d) {
      /* Rien a faire ici */
    }
};

// RECTANGLE derive de FORME
class RECTANGLE : public FORME { int L;  int l;
public:
    RECTANGLE(int a, int b, int c, int d) :
    FORME(a,b), // choix constructeur de la super classe
    L(c), l(d) {      }
};
```

DERIVATION ET PROTECTION

➢ Rappel :

	VIsivilité des membres de A dans...		
Membres de A	...Membres de A	...Amis	...Autres
private	visible	visible	Non visible
protected	visible	visible	Non visible
public	visible	visible	visible

➢ Dérivation et protection

La dérivation peut être privée ou publique.
L'évolution est toujours dans le sens de la restriction.

```
class A {...}

// héritage public. Le plus courant
class B  : public A { ....} ;

// héritage privé
class C : A { ....} ;
```

Héritage privé:

- la classe dérivée n'a pas de droits privilégié sur la classe de base.
- les membres privés de la base sont inaccessibles

➢ Evolution des protections en fonction de l'héritage

	Statut des membres de A dans la classe fille	
Membres de A	Héritage public	Héritage privé
private	*Non visible*	*Non visible*
protected	Membre protected	Membre private
public	Membre public	Membre private

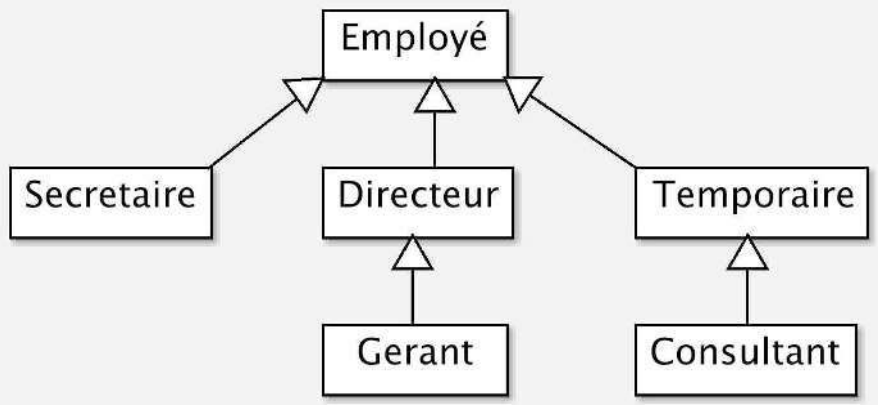
Notez donc qu'un attribut *protected* dans la classe mère *reste accessible* dans la classe fille.

DERIVATION ET PROTECTION : EXEMPLE

```
class A {         int a;
      protected : int b;
      public :    int c;
};

class B : public A {
public:
      void f() {
          a=5;      // ERREUR : a est privé
          b=5;      // OK : b protected et héritage publique
          c=5;      // OK : c public
      }
friend void g(B x);
};

class C : private A {
public :
      void f() {
          a=5;      // ERREUR : a est privé
          b=5;      // OK : f est membre de C
          c=5;      // OK : f est membre de C
      }
friend void h (C x);
};

void g(B x) {
      x.a=5;        // ERREUR : a est privé
      x.b=5;        // OK : b protected et héritage publique
      x.c=5;        // OK : c public
}
void h(C x) {
      x.a=5;        // ERREUR : a est privé
      x.b=5;        // OK : b protected et et h amie de C
      x.c=5;        // OK : c public
}
void r(B x) {
      x.a=5;        // ERREUR : a est privé
      x.b=5;        // ERREUR : b protected
      x.c=5;        // OK : c public
}
void r(C x) {
      x.a=5;        // ERREUR : a est privé
      x.b=5;        // ERREUR : b protected
      x.c=5;        // ERREUR : héritage privé
}
```

HIERARCHIE DE CLASSES DERIVEES

Plusieurs classes peuvent dériver d'une même classe de base.
L'héritage est récursif : une classe dérivée peut servir de classe de base à une autre classe dérivée.




```
class Employe  {
protected:
    int salaire;
    char nom[50], prenom[50];
public:
    void paye()const { cout << salaire; }
    void aff()  const {
      cout << "Je suis "<<nom << " "<<prenom;
    }
};

class Secretaire : public Employe  {
    char nomdirecteur[50];
public:
    void aff() const {
      Employe::aff();
      cout<<" dans le service de "<< nomdirecteur;
    }
};
class Directeur : public Employe  {
    int primes;
public:
    void paye()const  {
      cout << " salaire : " << salaire+primes;
    }
};
```

HIERARCHIE DE CLASSES DERIVEES

```
class Temporaire : public Employe  {
    int date_debut, duree;
public:
    void aff() const {
      Employe::aff();
      cout << " embauché de "<< " date_debut "
           << " à "<< date_debut+duree;
    }
};

class Gerant : public Directeur  {
    char **societes;
public:
    void aff() const {
      Directeur::aff();
      cout << " des societes : " ;
      for (char *p=societes; p!=NULL; )
          cout << *p++;
    }
};



main() {
    Secretaire a1,a2;
    Directeur b1,b2,b3;
    Gerant c1;
    Temporaire x1,x2,x3,x4;
    Employe * tab[4];

    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&b2;


    // Valeurs des a1,a2...
    a1.aff(); b1.aff();
    // Ce qui suit n'exécute que Employe::aff()
    // car le type statique de tab[i] est Employe !
     for (int i=0; i<4; i++) tab[i]->aff();
}
```

LIAISON DYNAMIQUE (1)

Problème : avec un pointeur de base (Employe *), on aurait besoin que la « bonne » méthode soit exécutée, en fonction du type dynamique de l'objet pointé.

Comment savoir quel est le type dynamique (« réel ») de l'objet pointé (Secretaire, Directeur, ..) ?

Solution 1 : RTTI et dynamic_cast<>

```
class Employe  {
protected:
    int salaire; int prime;
    char nom[50], prenom[50];
public:
    void paye() const {
      if ( dynamic_cast<Secretaire *>(this) ) {
           cout << salaire;
      } else if ( dynamic_cast<Directeur *>(this) ) {
           cout << salaire + prime ;
      }
      •••
    }
};

class Secretaire : public Employe  {
    char directeur[50];
}

main() {
    Secretaire a1,a2;
    Directeur b1,b2;
    employe * tab[4];
    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&b2;

    // Valeurs des a1,a2...
    a1.paye(); b1.paye();

    // OK: Employe::paye() affiche
    // ce qu'il faut grace au dynamic_cast<>
    for (int i=0; i<4; i++) tab[i]->paye();
}
```

LIAISON DYNAMIQUE (2)

Solution 2 : champ de type (à la C)

```
typedef enum { SECRETAIRE, DIRECTEUR, •••} TYPE_EMPLOYE ;

class Employe  {
protected:
    TYPE_EMPLOYE quijesuis;
    int salaire; int prime;
    char nom[50], prenom[50];
public:
    Employe(TYPE_EMPLOYE _quijesuis, •••) {•••}
    void paye() const {
     switch (quijesuis) {
          case SECRETAIRE : cout << salaire; break;
          case DIRECTEUR : cout << salaire+primes; break;
     }
    }
};

class Secretaire : public Employe  {
    char directeur[50];
public:  // ATTENTION au constructeur
    Secretaire(): Employe( SECRETAIRE, •••) { ••• ; }
};

main() {
    Secretaire a1,a2;
    Directeur b1,b2;
    employe * tab[4];
    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&b2;

    // Valeurs des a1,a2...
    a1.paye(); b1.paye();

    // OK: Employe::paye() sait toujours ce qui est manipulé
    for (int i=0; i<4; i++) tab[i]->paye();
}
```

Aucune de ces solutions n'est satisfaisante !

LIAISON DYNAMIQUE. METHODE VIRTUELLE. (1)

➢ Liaison dynamique : mot clé *virtual*.

1. Déclarer une méthode d'une super classe *virtuelle*
2. Redéfinir cette méthode dans les sous classes.

➔ la méthode exécutée sera celle de la classe du type **dynamique** de l'objet et sera choisie automatiquement à l'exécution.

Syntaxe :

```
class >ClasseMere> {
    virtual  <prototype de méthode>
} ;
```

➢ Mécanisme sous-jacent

Les fonctions virtuelles et le mécanisme de liaison dynamique sont implémentées par un tableau de pointeurs sur des fonctions et un champ entier indiquant le type de l'objet.

LIAISON DYNAMIQUE. METHODE VIRTUELLE. (2)

➢ Exemple

```
class Employe {
protected:
    int salaire;
    char nom[50], prenom[50];
public:
    virtual void paye() const { cout << salaire; }
    virtual void aff()  const {
      cout << "Je suis "<<nom << " "<<prenom;
    }
};
```

Les autres classes ne changent pas ! Eg, la méthode aff() de Secretaire reste :

```
void Secretaire::aff() const {
      Employe::aff();
      cout<<" dans le service de "<< nomdirecteur;
    }
```

Le main ne change pas :

```
main() {
    Secretaire a1, a2;
    Directeur b1;
    Gerant c1;
    Employe * tab[4];
    tab[0]=&a1;  tab[1]=&a2;  tab[2]=&b1;  tab[3]=&c1;

    // Employe::aff() est virtuelle
    // => la méthode aff() exécutée est celle
    // du type dynamique (« effectif ») de l'objet
    // quel que soit le type statique utilisé pour
    // manipuler l'objet
    for (int i=0; i<100; i++) tab[i]->aff();}
    // les méthodes aff() exécutées sont celles de
    // Secretaire, Secretaire, Directeur puis Gerant.
}
```

CLASSE ABSTRAITE, METHODE VIRTUELLE PURE (1)

➢ Méthode virtuelle pure

Méthodes qui ne peuvent avoir de sens ou dont le sens est incomplet

Syntaxe :

```
virtual type nom_methode (paramètres) = 0;
```

➢ Notion de classe abstraite

Classe dont l'implantation n'est pas complète:

- Elle représente un concept abstrait
- Elle possede au moins une méthode virtuelle pure
- Il est impossible de créer un objet de cette classe.
- On peut utiliser un pointeur ou une référence sur une classe abstraite (polymorphisme).

➢ Exemples de classes abstraites

Dans les exemples précédents du cours, typiquement :

- **FORME** serait une classe abstraite.
  FORME::dessiner() serait virtuelle pure.
  - On ne sait pas dessiner() une forme à ce niveau de la hiérarchie.
  - On ne peut écrire le code de dessiner que dans les sous-classes.

- **Employe** serait une classe abstraite
  Il n'est pas possible d'instancier un « Employe » sans plus de précision ; seules les sous classes peuvent être instanciées.
  Par exemple, Employe::afficher() serait virtuelle pure.

CLASSE ABSTRAITE, METHODE VIRTUELLE PURE (2)

➢ Exemple

```
class FORME { protected: int X; int Y;
public:
   virtual ~FORME() {} ;
   // Impossible de dessiner à ce niveau => virtuelle pure
   virtual void dessine() = 0;
   virtual void aff()=0 { cout << "forme " << X << x ; }
}; // FORME est abstraite car a une méthode virtuelle pure

class RECTANGLE : public FORME { int l; int L;
public:
   void dessine(){ goto(X,Y); draw_box(X,Y,X+L,Y+l); }
   void aff(){
        FORME::aff() ;
        cout << "rectangle " << l << L ; }
   }
};

class ELLIPSE : public FORME { int ga; int pa
public:
   void dessine(){ goto(X,Y); draw_circle(X,Y,ga,pa); }
   void aff(){
        FORME::aff() ;
        cout << "ellipse " << l << L ;
   }
};

main(){
   FORME a;  // IMPOSSIBLE : FORME est une classe abstraite
   ELLIPSE b;    // OK
   b.dessine();  // ELLIPSE::dessine()
   FORME *p = &b;
   p->dessine();     // OK, ELLIPSE::dessine()
}
```

CLASSE ABSTRAITE PURE

➢ Classe abstraite pure

- Classe qui n'a aucune méthode concrete et ne declare aucun attribut.
- Définit un contrat : oblige à avoir une API commune
  - toutes les sous classes doivent implanter toutes les méthodes déclarées dans la super classe.
- *Equivalente à la notion d'Interface en UML ou Java...*

➢ Exemple

La classe abstraite pure Motorisé declare une méthode allumerMoteur().
On peut avoir un conteneur d'objets motorisés et "démarrer" tous ces objets.

```
class Motorise {
public:
   virtual ~Motorise() {} // nécessaire...
   virtual void demarrer() = 0 ;
};
class Voiture : public Motorise {
public:
   Voiture() { cout<<"Constructeur Voiture "<<endl;}
   ~Voiture() { cout<<"Destructeur Voiture "<<endl;}
   virtual void demarrer(){cout<<"Demarrer Voiture"<<endl;}
};

class Perceuse : public Motorise {
public:
 virtual void demarrer(){cout<<"Demarrer Perceuse"<<endl;}
};
void demarrerTabMotorises(Motorise * tab[], int n) {
   for (int i=0; i<n ; i++) { tab[i]-> demarrer(); }
}
int main() {
   Motorise * tabMotorise[2];
   tabMotorise[0] = new Voiture();
   tabMotorise[1] = new Perceuse ();
   demarrerTabMotorises(tabMotorise, 2);
   for (int i=0; i<2 ; i++) { delete tabMotorise[i]; }
}
```

HERITAGE ET CYCLE DE VIE : DESTRUCTEUR VIRTUEL

➢ Exemple problématique

```
class Base {
public:
    Base() { cout<<"Constructeur Base "<<endl; }
    ~Base( ){ cout<<"Destructeur Base"<<endl; }

};

class Derivee : public Base {
public:
    Derivee() { cout<<"Constructeur Derivee "<<endl;}
    ~Derivee() { cout<<"Destructeur Derivee "<<endl;}
};

int main() {
    Base * b = new Derivee; // OK !
    delete b;      // delete d'un objet Derivee
                   // au moyen d'un pointeur de type Base!
}
```

Que se passe-t-il ? Ou est le problème ?

➢ Destructeur virtuel

Conseil : le destructeur d'une classe de base (dont derivent d'autre classes) doit avoir un destructeur *virtuel*.

```
class Base {
public:
    Base() { cout<<"Constructeur Base "<<endl; }
    virtual ~Base( ){ cout<<"Destructeur Base"<<endl; }

};
```

Le programme precedent affiche alors :

```
Constructeur Base
Constructeur Derivee
Destructeur Derivee
Destructeur Base
```

HERITAGE ET CYCLE DE VIE : COPIE ET AFFECTATION

➢ Constructeur par copie et opérateur d'affectation

Imposer et respecter l'ordre hierarchique de construction :

- le constructeur par copie de la classe dérivée **doit** appeler le contructeur par copie de la classe mère.
- l'opérateur d'affectation de la classe dérivée **doit** appeler l'opérateur d'affectation de la classe mère.

➢ Exemple

```
class A {
public:
   A() { cout <<"A::A ; " ;}
   A(const A & other) { cout <<"A::A(copy) ; " ;}
   A& operator=(const A& other) {
        cout <<"A::operator= ; ";
        return *this ;
}
   virtual ~A() { cout <<"A::~A " <<endl ;}
};
class B : public A{
public:
   B() : A() { cout <<"B::B" <<endl;}
   B(const B & other): A(other) {cout <<"B::B(copy)"<<endl;}
   B& operator=(const B& other) {
        A::operator=(other);
        cout <<"B::operator=" <<endl;
        return *this
   }
   ~B() { cout << "B::~B ; "  ;}
};
```

```
main() {
   B b;
   B b2(b) ;
   B b3;
   b3 = b;
 }
```

```
A::A ; B::B
A::A(copy) ; B::B(copy)
A::A ; B::B
A::operator= ;
B::operator=
B::~B ; A::~A
B::~B ; A::~A
B::~B ; A::~A
```

HERITAGE ET CYCLE DE VIE : POLYMORHISME ET METHODE CLONE()

➢ Le problème

Avec les classes A et B precedents, polymorphisme **et** copies d'objets…

```
main() {
    B b;
    A * pa = &b;//Polymorphisme: on manipule B comme un A
    B b5(*pa);  //ERREUR ! Pas de B::B(const A &) !
}
```

➢ Méthode vritual clone()

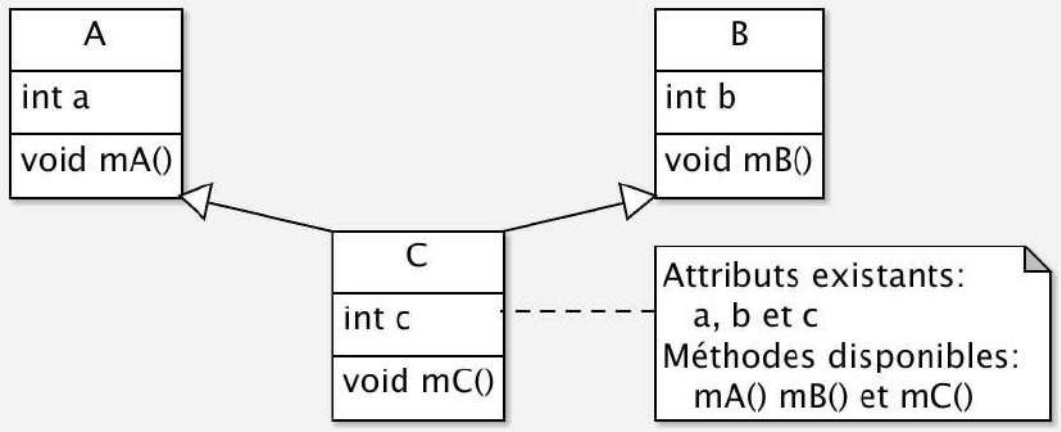
Dans les hiérarchie de classe, pouvoir faire des copies d'objets polymorphes, on définit souvent une méthode clone() comme suit :

```
class FIGURE {
public:
    virtual FIGURE * clone() const = 0 ;
};

class CERCLE : public FIGURE {
public:
    virtual FIGURE * clone() const {
          return new CERCLE(*this);
    }

};

main() {
    FIGURE * pf = new CERCLE() ;
    FIGURE * pf2 = pf ->clone();
    // le type dynamique de *pf2 est bien CERCLE  !
    // combien y a t il d'objets CERCLE en mémoire ?
}
```

HERITAGE MULTIPLE

Un concept peut avoir des points communs avec plusieurs autres concepts : il doit alors hériter de ces divers concepts : **héritage multiple**.

L'héritage multiple concerne aussi bien les champs que les méthodes.



➢ Syntaxe :

```
class derivee :
     <public,private> base1, <public,private> base2
     { •••
};
```

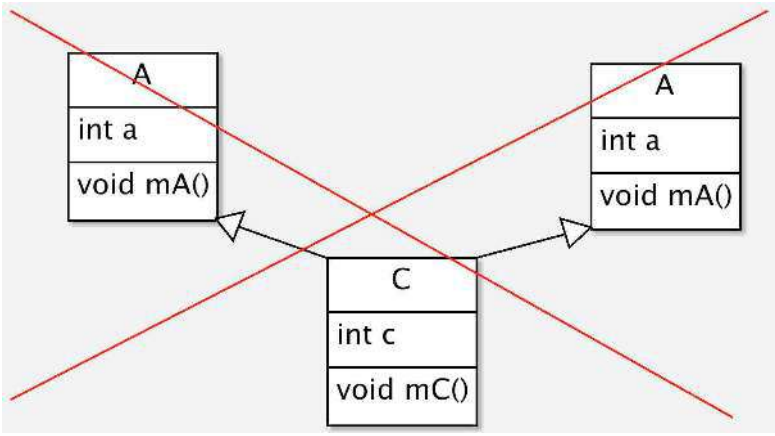
Attention : l'héritage est statique : il ne doit pas y avoir d'ambiguïté sur les méthodes. Les classes A et B ne doivent pas avoir une méthode de nom identique; Dans le cas contraire, il faut spécifier complètement le nom de fonction lors de l'appel.

➢ Exemple

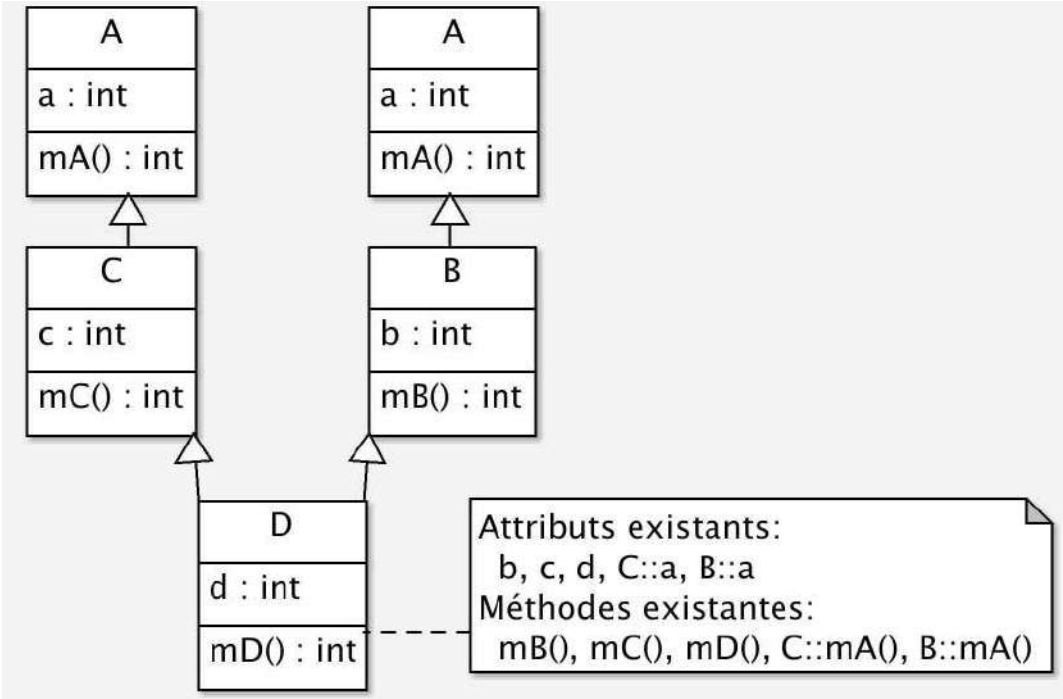
```
class A { public : void m1() { •••} ; };
class B { public : void m1() { •••} ; };

class C : public A, public B { •••};

main() {  C x;
    x.m1();          // ERREUR : ambiguïté
    x.A::m1();       // OK
    x.B::m1();       //OK
}
```

HERITAGE MULTIPLE D'UNE MEME CLASSE

L'héritage multiple d'une même classe est interdit.



L'héritage **indirect** multiple d'une même classe est autorisé. La classe de base est alors dupliquée 2 fois.




```
main() {
     C x;
     x.c = x.d= x.b = 0;   // OK
     x.a=0;                //ERREUR : ambiguïté
     x.D::a = x.B::a = 0;  // OK
}
```

HERITAGE MULTIPLE ET PARTAGE. HERITAGE VIRTUEL

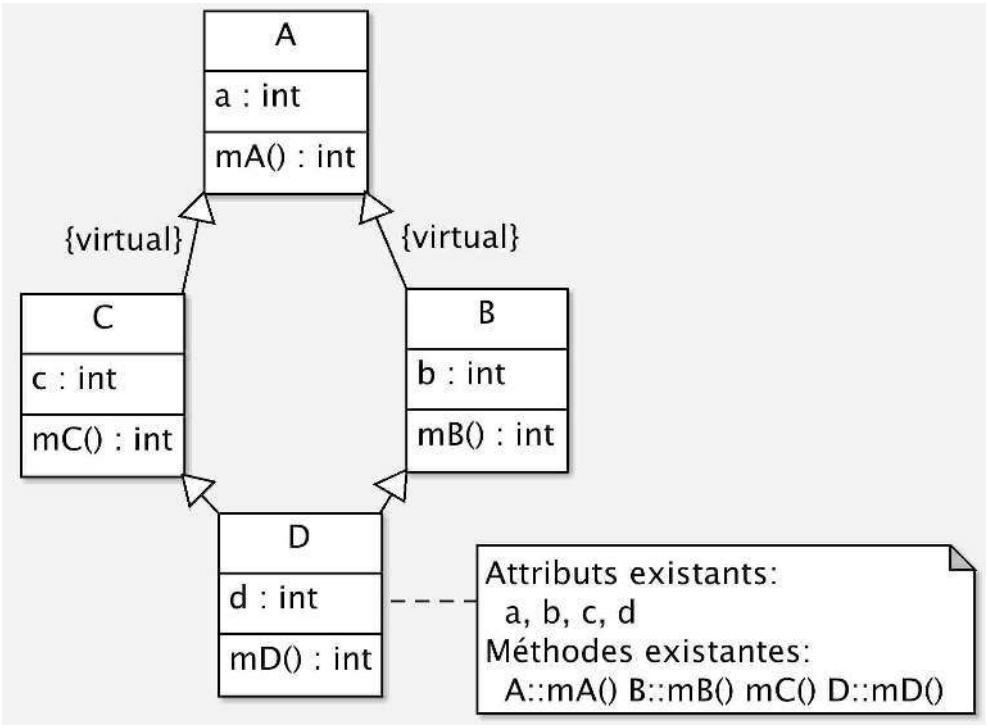
L'héritage multiple peut être vu comme une manière de rassembler des classes pour en créer de nouvelles plus complexes, pour intégrer des outils développés séparément.

Autre situation : développer des classes fortement liées entre elles, partageant des propriétés => héritage indirect d'une même classe de base, sans duplication de l'objet de base (classe iostream qui dérive de istream et ostream)

Pour ce cas : **héritage *virtuel***

➢ Syntaxe

```
class classe_dérivée : public virtual classe_base {•••};
                     : private
```



➢ Remarques

Constructeur : le constructeur de C doit passer les arguments aux constructeurs de D et B, mais aussi aux constructeurs de la classe de base virtuelle A.

Le constructeur de la classe de base virtuelle est toujours appelé avant les autres

HERITAGE MULTIPLE ET PARTAGE. HERITAGE VIRTUEL

➢ Exemple

```
class FENETRE { public : int X; int Y; •••};

class MENU : public virtual FENETRE  {
public:
    int nb_lignes;
    MENU (..) { •••}    // Constructeur
    •••
};

class F_COUL : public virtual FENETRE  {
public:
    int couleur;
    F_COUL (..) {•••} // Constructeur
    •••
};

class F_MENU_COUL :  public MENU, public F_COUL  {
public :
    F_MENU_COUL (...):
      FENETRE (...) {
      F_COUL (...),
      MENU (...),
           •••
    }
    •••
} ;

};
```