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, qui s'échangent des messages.

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*.

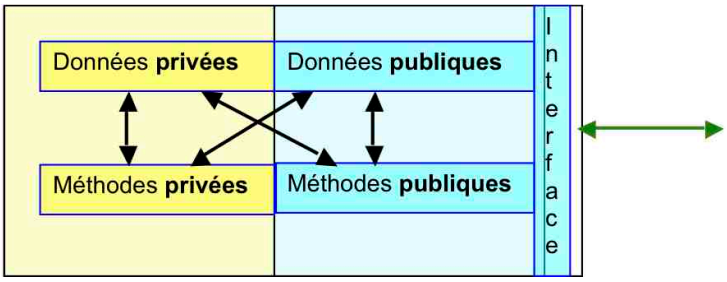
Une classe définit les membres qu'auront toutes ses instances :
- Les types des ***attributs*** (eg : toute « `Voiture` » a une « couleur » une « position » et un « `Moteur` »)
- des ***opérations*** (ou *fonctions membres* ou *méthodes*) qui manipulent ces attributs (eg : toute Voiture a une opération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

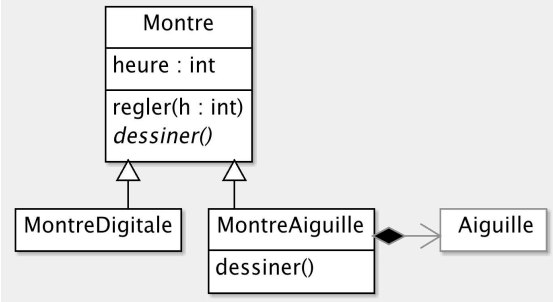
⟶▷ **"sorte-de"** (lien *est-un*)
relation hiérarchique père-fils, notion d'héritage

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`.

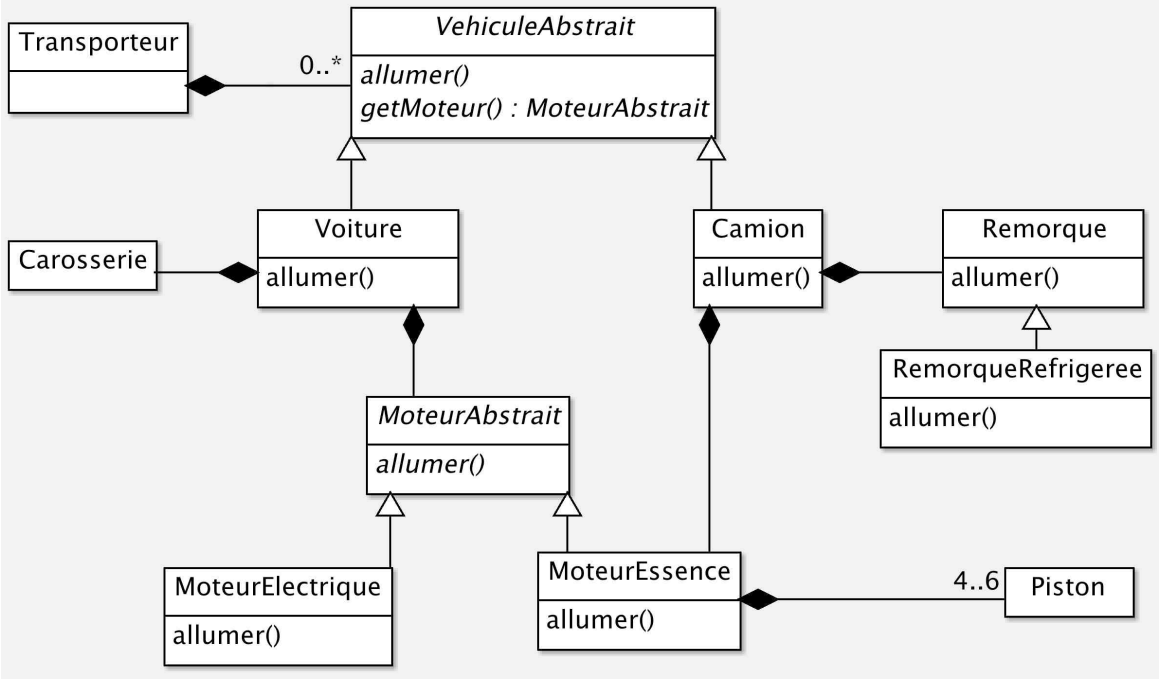
Intérêts:

- 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)

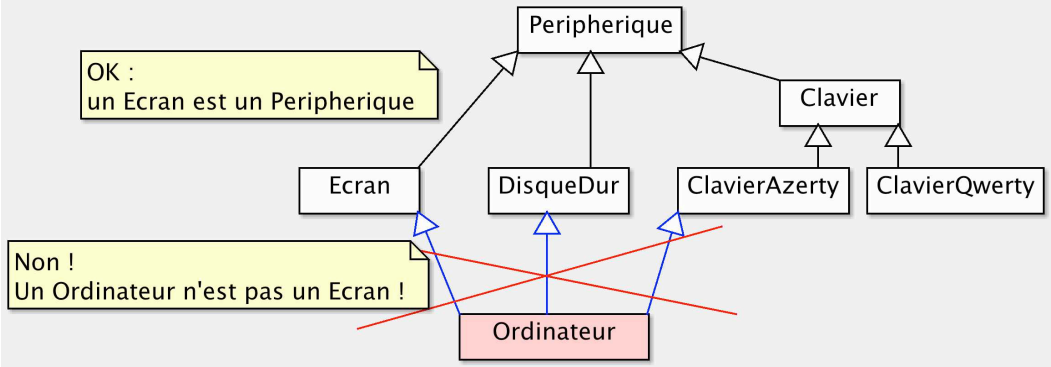
CONCEPTS ET MOTS CLES DE LA POO (4)

➢ 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++

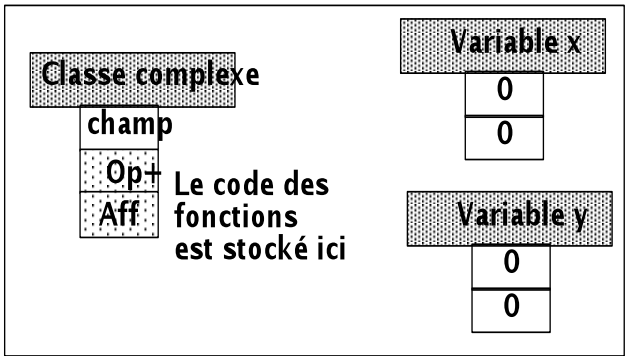
➢ Déclaration d'une classe : syntaxe

```
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
    }
} ;
```

EN PRATIQUE: .H ET .CPP

1 classe s'implémente dans 2 fichiers :

1. Le header : **`maclasse.h`** contient
   - la déclaration de la classe
   - la déclaration des fonctions et classes amies
   - les méthodes ou fonctions *inline*
2. Le source : **`maclasse.cpp`** contient
   - le code C++ des méthodes de la classe : maclasse ::

Fichier header Employe.h : *déclaration* 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 DEFINIE PAR L'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; }
```