4. CLASSES ET OBJETS EN C++

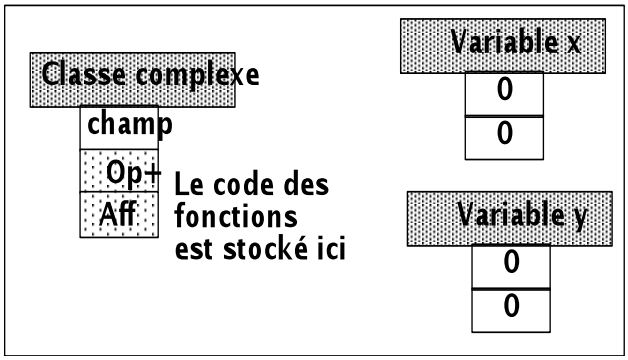
➢ Déclaration d'une classe : syntaxe

```
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();
}
```

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
```

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.
- L'encapsulation permet que l'objet garantisse son état : respect des invariants de classe.

Notion d'invariant de classe : loi que la classe s'engage à respecter sur les valeurs des attributs de ses instances.
Exemple : la classe `CompteBancaire` s'engage à ce que l'attribut solde de tous les objets `CompteBancaire` soit toujours positif

➢ Notion d'accesseur et de modifieur

On a recours à des méthodes pour accéder aux valeur des 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* qui donc ne modifie pas l'état de 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: la méthode est déclarée 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 ; }
} ;

//fonction
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 la référence sur l'objet passée directement) tout en gardant la syntaxe d'un passage par valeur.

- Comme la référence est `const`, on ne pourra appeler que des méthodes déclarées `const` (des accesseurs) qui ne modifieront pas l'état de l'objet. Cela est vérifié syntaxiquement (par le compilateur).

ACCESSEUR RETOURNANT UNE CONST &

RETOUR D'ATTRIBUT PAR PSEUDO-VALEUR

➢ Retour par pseudo valeur de la valeur d'un objet attribut

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 ! Beaucoup plus rapide !
    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 :

- Le retour par **référence évite que l'objet soit copié**
- Le qualificateur *const* de la référence retournée permet la **vérification syntaxique** (par le compilateur) de la non modification de l'objet retourné par la méthode.

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 !
}
```