Table des matières

1 Introduction aux classes

1.1 Introduction générale

Nous avons vu jusqu'ici un nombre restreint de types composés. En particulier, pour grouper plusieurs variables, nous ne connaissons encore que les tableaux. Or ceux-ci ne permettent de grouper que des variables de même type. Pour en grouper de différentes, il faut utiliser les *structures*[1], héritées du C, ou mieux encore les *classes*, qui sont la pierre de base de la *programmation orientée objet*.

Les données d'un programme sont rarement dispersées. Elles peuvent en général être pensées sous la forme de groupes plus ou moins importants, ayant une cohérence significative. Par exemple, dans les fichiers informatiques de gestion du personnel dans une entreprise ou une administration, on utilisera des fiches contenant le nom, le prénom, l'âge, l'adresse, etc., de chaque employé. Il serait peu logique de placer chacun des éléments de ces fiches dans des tableaux différents, car cela compliquerait la recherche de l'ensemble des caractéristiques d'un employé donné.

Les *classes* permettent de créer des variables (appelées *objets* ou *instances*) regroupant plusieurs types de données (les *membres*) et sur lesquelles des fonctions spécifiques sont définies, appelées *méthodes* de la classe. Une classe peut donc être vue comme un nouveau type. Le fait que les objets soient accompagnés de méthodes est à rapprocher des structures mathématiques : par exemple, un groupe est équipé d'une loi de composition et d'un inverse par exemple.

1. Nous ne présenterons pas les structures dans ce texte, leur maniement étant très similaire à celui des classes.

1.2 Premier exemple

Voici par exemple comment on construit une classe `fiche` dont chaque instance (ou objet) représente la fiche d'un employé. Les membres (i.e. les composantes) des objets de cette classe sont tous publics (on verra plus tard la différence entre membres publics et membres privés) : ce sont le nom, le prénom, le service, l'année de naissance et l'indice salarial de l'employé en question. Cette classe possède une seule méthode (i.e. fonction spécifique) : celle qui donne l'année de départ à la retraite de l'employé. Notons que la fonction `salaire`, déclarée hors de la classe, n'est pas ce que l'on appelle une méthode.

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

class fiche {
        public :
                        string nom;
                        string prenom;
                        string service;
                        int indice_salarial;
                        int annee_de_naissance;
                        int retraite() {return annee_de_naissance+65;}
};

double salaire(fiche employe){return 4*employe.indice_salarial;}

int main(){fiche employe1;
        employe1.nom = "Dupont";
        employe1.prenom = "Nicolas";
        employe1.annee_de_naissance = 1976;
        employe1.service = "Comptabilite";
        employe1.indice_salarial = 551;

        fiche employe2;
        employe2.nom = "Durand";
        employe2.prenom = "Jacques";
        employe2.annee_de_naissance = 1966;
        employe2.service = "Export";
        employe2.indice_salarial = 651;

        fiche *pointeur_fiche;
        pointeur_fiche=&employe1;
        cout<<employe2.service<<" "<<pointeur_fiche->nom
        <<" "<<salaire(employe1)<<" "<<employe1.retraite()<<endl;};
```

(nom du fichier : `Chapitre3/prog01.cpp`*)*

On remarque ici la syntaxe particulière `p->nom` qui est strictement équivalente à la syntaxe [2] `(*p).nom` lorsque `p` est du type `fiche *`.

Voici un autre exemple, dans lequel on voit que les méthodes peuvent avoir des arguments, et que via *l'opérateur global de résolution* : : , on peut définir les méthodes hors de la déclaration de la classe (d'ailleurs, le plus souvent, la définition des classes se fait dans un fichier en-tête `.h` et les méthodes sont définies dans un fichier `.cpp`).

2. Dans l'expression précédente les parenthèses sont inévitables pour éviter la confusion avec `*(p.nom)` qui est tout autre chose !

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

class point { public : double abs, ord;
                        void afficher();
                        void translater(double x, double y); };

void point::afficher() {cout<<"("<<abs<<","<<ord<<")"<<endl;};

void point::translater(double x, double y){abs+=x; ord+=y;};

int main(){point P;
        P.abs=1;
        P.ord=1;
        P.afficher();
        P.translater(3,5);
        P.afficher();};
```

(nom du fichier : `Chapitre3/prog02.cpp`*)*

Exercice 1 (Écriture d'une classe très simple)**.** *Compléter la classe* `point` *en y ajoutant une méthode de prototype*

`double distance_a(point Q)`

qui, appliquée à un objet `P`*, donne la distance de* `P` *à* `Q`*. Tester cette fonction.*

Exercice 2 (Écriture d'une classe très simple)**.** *Écrire une classe représentant les nombres complexes, possédant deux membres (représentant les parties réelle et imaginaire) et une méthode (donnant le module), tous publics. Tester cette classe. Vous pouvez également essayer de programmer de nombreuses autres fonctions liées aux nombres complexes (conjugaison, etc).*

1.3 Masquage d'information

On peut indiquer, avec une catégorie `private` (en opposition à `public`) les membres et méthodes dont on veut interdire l'accès via . et -> Ceux-ci ne seront accessibles que dans les méthodes de la classe (et celles des classes amies ainsi que dans les fonctions amies, *cf.* le paragraphe suivant). L'intérêt de cette spécification est double : d'une part cela devient primordial pour l'héritage (cf. chapitre ultérieur) et d'autre part, cela apporte une saine discipline de programmation lors de l'utilisation finale d'une classe.

Lorsque l'on déclare des membres privés, il devient nécessaire de définir des accesseurs (qui permettent de lire les membres) et des mutateurs (qui permettent de les changer). Bien que ce soit "plus long", procéder de la sorte évite bien souvent des erreurs de programmation difficiles à déceler ! De plus, via l'utilisation de fonctions `inline`, cela ne ralentit en rien le programme (seulement sa compilation).

Exemple :

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

class point {
  public :
        double get_abs() {return abs;};
        double get_ord() {return ord;};
        void set_abs(double x) {abs=x;};
        void set_ord(double y) {ord=y;};
        void afficher() {cout<<"("<<abs<<","<<ord<<")"<<endl;};
  private :
        double abs, ord;
        void initialiser(){abs=0; ord=0;};
        };

int main(){point P;
  P.afficher(); //on va obtenir les valeurs par defaut des abs et ord
  P.set_abs(2);
  P.set_ord(5);
  cout<<P.get_abs()<<endl;}
```

(nom du fichier : `Chapitre3/prog03.cpp`*)*

Exercice 3 (Écriture d'une classe très simple)**.** *Compléter la classe* `point` *en y ajoutant un membre privé* `nom` *de type* `string`*, écrire les accesseurs et mutateurs* `get_nom`*,* `set_nom` *associés. Tester.*

1.4 Classes et fonctions amies

1.4.1 Fonctions amies

Une fonction `f` amie de la classe `C` a accès (lecture et écriture) aux attributs privés des objets de `C`. Bien entendu, la déclaration "`f` est amie de `C`" se fait dans la déclaration de `C`, via le mot-clé `friend`, placé avant la déclaration (qui n'est pas la définition) de la fonction.

Exemple :

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

class point {double abs, ord; //ce qui est place avant public est private
  public :
        void set_coord(double x, double y) {abs=x; ord=y;}
        friend double length(const point, const point); };

double length(const point P, const point Q){
        double x=P.abs-Q.abs, y=P.ord-Q.ord;
        return sqrt(x*x+y*y);};

int main(){point P,Q;
  P.set_coord(2,5);
  Q.set_coord(6,4);
  cout<<length(P,Q)<<endl;}
```

(nom du fichier : `Chapitre3/prog04.cpp`*)*

Exercice 4 (une question d'optimisation)**.** *Quel serait l'intérêt d'utiliser une fonction de prototype*

```
double length(const Point & Q,const Point & Q)
```

(quitte à changer les prototypes associés) dans un programme qui l'utiliserait de très nombreuses fois ?

1.4.2 Classes amies

Les méthodes de la classe `C'` amie de la classe `C` ont accès (lecture et écriture) aux attributs privés des objets de `C`. Bien entendu, la déclaration "`C'` est amie de `C`" se fait dans la déclaration de `C`.

Exemple :

```
#include <iostream>

class point {double abs, ord;
  public :
        void set_coord(double x, double y) {abs=x; ord=y;}
        friend class segment; };

class segment { public : point A,B;
                void translater(double x, double y)
                                {A.abs+=x; A.ord+=y; B.abs+=x; B.ord+=y;} };


int main(){point P,Q;
  P.set_coord(2,5);
  Q.set_coord(6,4);
  segment S;
  S.A=P;
  S.B=Q;
  S.translater(1,1);}
```

(nom du fichier : `Chapitre3/prog05.cpp`*)*

1.5 Constructeurs

1.5.1 Le constructeur de base

Jusque là, nous ne nous sommes pas soucié de la construction des éléments des classes. La raison est que toutes les classes que nous avions construites possédaient des constructeurs par défauts. Par exemple, lorsque, dans le premier exemple, nous écrivions

```
fiche employe1;
```

un objet de la classe `fiche` appelé `employe1` était créé en donnant à ses membres leurs valeurs par défaut (qui sont la chaîne vide pour les membres `string` et 0 pour les membres `int`).

En fait, lors de la création d'un objet d'une classe, on fait toujours appel à un *constructeur* : une méthode de la classe qui préside à la création de l'objet. Lorsque cette méthode n'apparaît pas dans la déclaration de la classe, un *constructeur par défaut* est créé par le compilateur : il construit les objets en affectant aux membres leurs valeurs par défaut (lorsqu'il en existe, sinon, il n'y a pas de constructeur par défaut). Bien souvent (par exemple lorsque l'un des membres est un pointeur), les services rendus par ce constructeur par défaut sont insuffisants et l'on a besoin d'un ou de plusieurs constructeurs que l'on écrit nous même.

La syntaxe, pour l'écriture d'un constructeur d'une classe `ma_classe`, est la suivante :

`ma_classe` *(arguments) : instanciation des membres { instructions}*

Exemple :

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

class point { double abs, ord;
  public :
        point(double x=0, double y=0) : abs(x), ord(y) {}
        void afficher() {cout<<"("<<abs<<","<<ord<<")"<<endl;} };

int main(){point P;
  P.afficher();
  point Q(5,7), R(9);
  Q.afficher();
  R.afficher();}
```

(nom du fichier : `Chapitre3/prog06.cpp`*)*

Exercice 5 (fondamental)**.** *Reprennez l'exemple avec les classes* `point` *et* `segment` *ci-dessus et écrivez deux constructeurs pour la classe* `segment` *: l'un ayant pour arguments une paire de* `point` *et un autre ayant pour arguments 4* `double` *(ces deux constructeurs coexisteront dans la même déclaration de classe). Tester.*

Remarque 1. *La présence d'un constructeur par défaut est* fondamentale *et le compilateur impose d'en écrire un, dès que les types sont non-triviaux.*

1.5.2 Le constructeur par copie

Certaines classes possèdent de nombreux constructeurs, les "jeux d'ingrédients" nécessaires à la construction de leurs objets pouvant être assez variés (par exemple, dans la classe `segment` plus haut, il serait possible d'écrire un constructeur ayant pour arguments une paire de `point`, mais aussi un autre constructeur ayant pour arguments 4 `double`).

Un constructeur se distingue des autres : *le constructeur par copie*. Il permet de "copier" un objet pour en créer un autre. Son utilité est centrale, car il est utilisé dans chacune des situations (très courantes) suivantes :

— lorsqu'un objet est copié via une initialisation,
— lorsqu'un objet est passé par valeur dans une fonction,
— lorsqu'un objet est retourné par valeur par une fonction.

Son prototype est **toujours** le suivant :

```
ma_classe( const ma_classe& )
```

(i.e. son argument est un objet constant de la classe, passé en référence) et c'est avec cela que le compilateur l'identifie. On s'en sert ensuite comme suit

`ma_classe x(...)` *(construction d'un objet par un autre constructeur)*
`ma_classe y(x);` *(on utilise ici le constructeur par copie pour créer* `y` *en copiant* `x`*)*

Remarque 2. *Comme le constructeur par défaut, la présence d'un constructeur par copie est* obligatoire *dès que les types sont non-triviaux.*

Exemple :

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

class point { double abs, ord;
  public :
        point(double x=0, double y=0) : abs(x), ord(y) {}
        point(const point& P) : abs(P.abs), ord(P.ord) {}
        void afficher() {cout<<"("<<abs<<","<<ord<<")"<<endl;} };

int main(){point P(3,4);
  point Q(P);
  Q.afficher();}
```

(nom du fichier : `Chapitre3/prog07.cpp`*)*

Exercice 6 (Écriture d'un constructeur par copie)**.** *Reprendre les classes* `point` *et* `segment` *ci-dessus et écrire des constructeurs par copie. Tester.*

1.5.3 Utilité du constructeur par copie

Lorsque aucun constructeur par copie n'est déclaré dans la classe, le compilateur en crée un par défaut. Par exemple, on pourra remarquer que le programme ci-dessus fonctionne encore si l'on met en commentaire la ligne

```
point(const point& P) : abs(P.abs), ord(P.ord) {};
```

Cependant, dans de nombreux cas (lorsqu'un membre est un pointeur), le constructeur par défaut n'est pas satisfaisant du tout et donne lieu à de nombreux problèmes. L'exercice suivant illustre ce cas de figure.

Exercice 7 (Copie pour une classe ayant un membre pointeur)**.** *Que va rendre le programme suivant ? Comment rendre la construction par copie plus satisfaisante ici ?*

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

class vecteur{ public : int n; int *coord;
  vecteur(int n) : n(n), coord(new int[n]) {}
  void affiche() {for(int i=0;i<n;i++) cout<<coord[i]<<" "; cout<<endl;} };

int main(){vecteur u(5);
  u.coord[0]=7;
  u.affiche();
  vecteur v(u);
  v.coord[0]=9;
  u.affiche();};
```

(nom du fichier : `Chapitre3/prog08.cpp`*)*

1.5.4 Pointeurs vers objets et constructeurs

On peut définir des pointeurs sur objets de deux manières :
- en utilisant l'opérateur adresse `&` à partir d'un objet existant,
- en utilisant l'opérateur d'allocation de mémoire `new` avec un constructeur (sans oublier `delete`) sans objet existant préalable.

On utilise alors la flèche `->` pour accéder aux membres : si `p` est un pointeur vers un objet `x`, alors les notations `x.f()`, `(*p).f()` et `p->f()` sont équivalentes.

Exercice 8 (Pointeurs vers objets)**.** *Que rend le programme suivant ?*

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

class vecteur{public : int n; int *coord;
        vecteur(int n) : n(n), coord(new int[n]) {};
        vecteur(const vecteur& u) : n(u.n), coord(new int[n])
                {for(int i=0;i<n;i++) coord[i]=u.coord[i];}
        void affiche() {for(int i=0;i<n;i++) cout<<coord[i]<<" "; } };

int main(){vecteur u(4);
        vecteur *p = &u;
        vecteur *q = new vecteur(2), *r = new vecteur(u);
        p->affiche();
        q->affiche();
        r->affiche();
        delete q,r;};
```

(nom du fichier : `Chapitre3/prog09.cpp`*)*

Le fonctionnement précis de `new` est assez délicat, les exercices de la suite du texte permettent de mieux le comprendre.

Exercice 9. *Que va afficher l'exécution du programme ci-dessous ?*

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

class Complex { double Re, Im;
public : Complex(double x=3, double y=0) : Re(x), Im(y) {}
          double partie_reelle() {return Re;} };

int main() {Complex *p=new Complex;
        cout<<p->partie_reelle()<<endl;
        Complex *q=new Complex[5];
        cout<<q[2].partie_reelle()<<endl;}
```

(nom du fichier : `Chapitre3/prog10.cpp`*)*

1.6 Le destructeur

Pour chaque classe, un opérateur est utilisé à chaque fois que l'on sort d'un bloc d'instructions où un objet (non déclaré `static`) a été créé pour détruire l'objet en question (et libérer ainsi la mémoire). Cet opérateur, qui existe par défaut si on ne le réécrit pas, s'appelle le *destructeur*. La syntaxe de sa réécriture est la suivante :

`~ma_classe(){` *instructions* `};`

Il est utile dès qu'un membre de la classe est un pointeur car il faut décider soi-même s'il faut effacer ou non l'emplacement derrière un pointeur (selon qu'il a été créé par un `new` ou selon qu'il est lié à une structure plus globable qu'on ne souhaite pas effacer).

Exemple :

```
class vecteur{ public : int n; int *coord;
vecteur(int n) : n(n), coord(new int[n]) {}
vecteur(const vecteur& u) : n(u.n), coord(new int[n])
     {for(int i=0;i<n;i++) coord[i]=u.coord[i];}
~vecteur() {delete [] coord;};
void affiche() {for(int i=0;i<n;i++) cout<<coord[i]<<" ";cout<<endl;} };
```

(nom du fichier : `Chapitre3/prog11.cpp`*)*

Exercice 10 (Utilisation des constructeurs et destructeurs)**.** *Jouer avec la classe modifiée suivante de façon à voir clairement quand sont utilisés les constructeurs et le destructeur.*

```
class vecteur{ public : int n; int *coord;
vecteur(int n) : n(n), coord(new int[n])
    {cout<<"Appel au constructeur de base"<<endl;}
vecteur(const vecteur& u) : n(u.n), coord(new int[n])
     {cout<<"Appel au constructeur par copie"<<endl;
     for(int i=0;i<n;i++) coord[i]=u.coord[i];}
~vecteur()
     {cout<<"Appel au destructeur"<<endl; delete [] coord;}
void affiche() {for(int i=0;i<n;i++) cout<<coord[i]<<" ";cout<<endl;} };
```

(nom du fichier : `Chapitre3/prog12.cpp`*)*

1.7 Membres et méthodes `static`

Ce sont des membres ou des méthodes (privées comme publiques) qui sont la propriété d'une classe et non des objets de cette classe (c'est à dire que ce sont des membres et des méthodes qui ont la même valeur pour *tous* les membres de la classe). De tels membres ou méthodes sont signalés par le mot clé `static` et initialisés via l'opérateur global de résolution : : .

L'exercice suivant permet de bien comprendre leur fonctionnement.

Exercice 11 (Membres et méthodes static)**.** *Que rend ce le programme suivant ?*

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

class point { double abs, ord;
        public : static int nombre_de_points;
        point(double x=0, double y=0) : abs(x), ord(y) {nombre_de_points++;}
        point(const point& p) : abs(p.abs), ord(p.ord) {nombre_de_points++;}
        ~point() {nombre_de_points--;}
        static void aff_nbre() {cout<<nombre_de_points<<endl;} };

int point::nombre_de_points = 0;

int main(){point P;
  point::aff_nbre();//remarquer que cet appel n'est lie a aucun objet
  point Q(5,7), R(9), S(P);
  point::aff_nbre();
  point* pointeur1=new point(3,5);
  point::aff_nbre();
  point* pointeur2=new point(S);
  point::aff_nbre();
  delete pointeur1;
  delete pointeur2;
  point::aff_nbre();}
```

(nom du fichier : `Chapitre3/prog13.cpp`*)*

Comme l'exemple l'indique, cela permet de stocker à bas prix une information globale, par exemple le nombre total de variables déclarées (ou leurs adresses, etc). On se rend compte sur cet exemple que le destructeur et les constructeurs ont souvent des actions non triviales sur ces variables globales.

1.8 Gestion des fichiers multiples dans l'écriture et le maniement de classes

On créera et manipulera les classes un peu lourdes selon le schéma suivant. Imaginons que l'on veuille créer une classe `Point` ayant deux membres privés `abs` et `ord` et les méthodes publiques `Point`, `~Point`, `afficher` et `distance` (cette classe est assez légère pour être écrite en un seul fichier, mais elle permet de comprendre la démarche).

On écrit alors le fichier `Point.hpp` suivant (appelé *fichier d'en-tête*).

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

#ifndef POINT_HPP
#define POINT_HPP

class Point {
        public:
                Point(double x = 0, double y = 0);
                Point(const Point &p);
                ~Point();
                void afficher ();
                double distance (Point p);
        private:
                double abs, ord;};
#endif
```

(nom du fichier : `Chapitre3/Point.hpp`*)*

Remarque 3. *Un mot pour expliquer la présence de* `#ifndef POINT_HPP`*,* `#define POINT_HPP` *et* `#endif` *: on peut mettre plusieurs fichiers d'en-tête (des headers) en préambule d'un fichier* `.cpp`*. Il est alors possible, par le biais de référencements mutuels, que sur la somme, on voie la classe* `Point` *définie plusieurs fois. Cela pourrait poser problème à la compilation, et donc pour éviter, on utilise la commande*

```
#ifndef nom_nouveau
#define nom_nouveau
...instructions...
#endif
```

qui commande, au cas où `nom_nouveau` *(qui est, ici,* `POINT_HPP`*) serait déjà passé, de sauter jusqu'à* `#endif`*.*

On définit alors les méthodes dans le fichier `Point.cpp` suivant.

```
#include "Point.hpp"

Point::Point(double x, double y) : abs(x), ord(y) {};

Point::Point(const Point &p) : abs(p.abs), ord(p.ord) {};

Point::~Point() {};

void Point::afficher() {cout<<"("<<abs<<","<<ord<<")"<<endl;};

double Point::distance(Point p)
        {return sqrt((abs-p.abs)*(abs-p.abs)+(ord-p.ord)*(ord-p.ord));};
```

(nom du fichier : `Chapitre3/Point.cpp`*)*

On compile ensuite le fichier `Point.cpp` avec la commande

```
g++ -c Point.cpp
```

Si tout se passe bien, un fichier `Point.o` est alors créé. Pour utiliser la classe `Point`, on a besoin du fichier `Point.o` et du fichier `Point.hpp`.

Lorsque par exemple, que l'on veut utiliser cette classe dans le programme `test-Point.cpp` suivant,

```
#include "Point.hpp"

int main() {Point p=Point(1,1), q;
        Point r=Point(q);
        q.afficher();
        cout<<p.distance(r)<<endl;}
```

(nom du fichier : `Chapitre3/test-Point.cpp`*)*

on compile comme suit :

```
g++ Point.o test-Point.cpp -o test-Point
```

Exercice 12. *Que va afficher l'exécution du programme* `test-Point` *ci-dessus ?*

Exercice 13 (Création d'une classe, gestion de l'allocation et de la restitution de mémoire)**.** *Construire une classe* `Pointnomme` *possédant les mêmes méthodes et membres que la classe* `Point` *ci-dessus avec, en plus, le membre privé* `nom`*, de type* `char *`*, ainsi que les méthodes* `lire_abs`*,* `lire_ord`*,* `change_abs`*,* `change_ord`*,* `change_nom`*, qui permettent respectivement de lire et changer les membres* `abs`*,* `ord` *et* `nom`*. Les méthodes* `lire...` *seront* `inline`*. La longueur du nom sera limitée à 100 caractères, sans espace. La méthode* `afficher` *sera aussi modifiée de façon à afficher aussi* `nom`*.*

Rappel : les principes de la création et la copie de chaînes de caractères sont résumés dans le petit programme ci-dessous. Dans la pratique, le recours à la bibliothèque `string` est plus simple.

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

int main(){char *c = new char;
        char d[]="abcdef";
        cout<< d <<endl;
        cout<< strlen(d) <<endl; //longueur de la chaine
        strcpy(c,d); //copie de la chaine c dans la chaine d
        cout<< c <<endl;
        cout<<c[0]<<" "<<c[3]<<endl;};
```

(nom du fichier : `Chapitre3/prog14.cpp`*)*

1.9 Exercices supplémentaires

Les exercices de ce paragraphe, relativement théoriques, permettent de bien comprendre les mécanismes subtils de la création, la copie, et la destructions d'objets. Ils ne sont pas indispensables lors d'une première approche du cours.

Exercice 14 (Classes sans constructeur par défaut)**.** *Quelles lignes peut-on décommenter sans empêcher ce programme de compiler ? Que modifier pour qu'il compile même en décommentant les trois lignes ?*

```
#include <iostream>

class Cl{ public : Cl(double x) {};};

int main() {// Cl *q;
// Cl *p = new Cl;
// Cl c;
}
```

(nom du fichier : `Chapitre3/prog15.cpp`*)*

Exercice 15. *On désire que le programme suivant* `prog16.cpp` *compile. Laquelle ou lesquelles des quatres possibilités peut-on "décommenter" ? Que modifier dans la définition de la classe pour que les quatres soient "décommentables" ?*

```
#include <iostream>

class Complex { double Re, Im;
        public : Complex(double x, double y) : Re(x), Im(y) {}; };

int main() {
        // possibilite 1 :
        // Complex z;

        // possibilite 2 :
        // Complex *p;

        // possibilite 3 :
        // Complex *q=new Complex;

        // possibilite 4 :
        // Complex *q=new Complex[5];
}
```

(nom du fichier : `Chapitre3/prog16.cpp`*)*

Exercice 16 (Classes ayant un membre de type pointeur, utilisation des références)**.** *Que doit-on ajouter à la classe* `Complexe`*, dans le programme suivant, pour que la ligne* `a=c` *soit décommentable ? À quoi sert la fonction* `Complexe (const Complexe &z)` *? À quoi sert la fonction* ∼`Complexe ()` *?*

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

class Complexe { double x,y;  char *nom;
 public : Complexe (double x=0, double y=0, char *s="") : x(x), y(y),
                        nom(new char[strlen(s)+1]) {strcpy(nom,s);};
  Complexe (const  Complexe &z) : x(z.x), y(z.y),
                        nom(new char[strlen(z.nom)+1]) {strcpy(nom,z.nom);};
  ~Complexe() {delete [] nom;}
  void afficher() {cout<<nom<<": "<<x<<" "<<y<<endl;};};

int main() {Complexe a(0,0,"Origine"), c(2,3,"Complexe1");
  Complexe d(c);
  d.afficher();
  //a=c;
  a.afficher();}
```

(nom du fichier : `Chapitre3/prog17.cpp`*)*

Exercice 17 (Copie par duplication et destruction des objets arguments d'une fonction)**.** *Que va rendre la fonction suivante ? Et si l'on décommente* `*p=*(x.p)` *? Et si l'on change le prototype de la fonction* `affichage` *en*

`void affichage(pointeur_vers_int &x)` *?*

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

class pointeur_vers_int {int* p;
public :          pointeur_vers_int(int n=0) : p(new int){*p=n;}
                  pointeur_vers_int(const pointeur_vers_int& x): p(new int)
                                {cout<<"utilisation du cloneur"<<endl;
                                //*p=*(x.p);
                                }
                  ~pointeur_vers_int(){delete p;
                         cout<<"utilisation du destructeur"<<endl;}
                  int contenu() {return *p;} };

void affichage(pointeur_vers_int x){cout<<x.contenu()<<endl;};

int main() {pointeur_vers_int x(5);
          cout<<x.contenu()<<endl;
          affichage(x);}
```

(nom du fichier : `Chapitre3/prog18.cpp`*)*

Exercice 18 (Copie par duplication et destruction des objets arguments d'une fonction)**.** *Que va rendre la fonction suivante ? Et si l'on change le prototype de la fonction* `affichage` *en* `void affichage(deux_int &x)` *?*

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

class deux_int { int *p;
public :         deux_int(int n=0, int m=0) :  p(new int[2]){p[0]=n; p[1]=m;}
                 deux_int(const deux_int& x) : p(new int[2]){
                                 p[0]=x.p[0]; p[1]=x.p[1];
                                 cout<<"utilisation du cloneur"<<endl;}
                 ~deux_int(){delete [] p;
                                 cout<<"utilisation du destructeur"<<endl;}
                 int contenu(int i) {return p[i%2];} };

void affichage(deux_int x){cout<<x.contenu(0)<<" "<<x.contenu(1)<<endl;};

int main() {deux_int x(5,7);
          affichage(x);}
```

(nom du fichier : `Chapitre3/prog19.cpp`*)*

Exercice 19 (Valeurs par défaut, utilisation des références)**.** *Dans le programme suivant, aurait-on pu remplacer* `Complex z(2,3);` *par* `Complex z;` *? Quelles lignes peut-on décommenter sans empêcher ce programme de compiler ? Qu'affichera-t-il alors ? Que modifier pour qu'il compile même en décommentant les deux lignes ?*

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

class Complex {double x,y;
  public:
    Complex(double x=0, double y=0) : x(x), y(y) {};
    double & Re() {return x;};
    double Im() {return y;};};

int main() {Complex z(2,3);
          cout<<z.Re()<<" "<<z.Im()<<endl;
          //z.Re()=5;
          //z.Im()=5;
          cout<<z.Re()<<" "<<z.Im()<<endl;}
```

(nom du fichier : `Chapitre3/prog20.cpp`*)*

Exercice 20 (Constructeurs et destructeurs)**.** *Donner la sortie du programme suivant.*

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

 class Truc { public :
        Truc() { cout << "constructeur" << endl ; }
        ~Truc() { cout << "destructeur" << endl ; } };

int main() {Truc x;}
```

(nom du fichier : `Chapitre3/prog21.cpp`*)*

Exercice 21 (Constructeurs et destructeurs)**.** *Donner la sortie du programme suivant.*

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

 class Truc { public :
        Truc() { cout << "constructeur" << endl ; }
        ~Truc() { cout << "destructeur" << endl ; }};

int main() {Truc * x = new Truc();}
```

(nom du fichier : `Chapitre3/prog22.cpp`*)*

Exercice 22 (Constructeurs et destructeurs)**.** *Donner la sortie du programme suivant.*

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

 class Truc { public :
        Truc() { cout << "constructeur" << endl ; }
        ~Truc() { cout << "destructeur" << endl ; } };

int main() {Truc * x = new Truc();
        delete x;}
```

(nom du fichier : `Chapitre3/prog23.cpp`*)*

Exercice 23 (Constructeurs et destructeurs)**.** *Donner la sortie du programme suivant.*

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

 class Truc { public :
        Truc() { cout << "constructeur" << endl ; }
        ~Truc() { cout << "destructeur" << endl ; } };

int main() {Truc * x = new Truc[5];}
```

(nom du fichier : `Chapitre3/prog24.cpp`*)*

Exercice 24 (Constructeurs et destructeurs)**.** *Donner la sortie du programme suivant.*

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

 class Truc { public :
        Truc() { cout << "constructeur" << endl ; }
        ~Truc() { cout << "destructeur" << endl ; } };

int main() {Truc * x = new Truc[5];
        delete [] x;}
```

(nom du fichier : `Chapitre3/prog25.cpp`*)*

Exercice 25 (Utilisation de if et de endif)**.** *Donner la sortie du programme suivant.*

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

#if !defined(pi) && !defined(base_exp)
        #define pi 3.14
        #define base_exp 2.72
#endif

int main() {cout<<pi<<" "<<base_exp<<endl;}
```

(nom du fichier : `Chapitre3/prog26.cpp`*)*

Exercice 26 (Utilisation de ifndef, define et endif)**.** *Donner la sortie du programme suivant.*

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

#ifndef ma_fonction_f
#define ma_fonction_f
   double f(double x=1) {return 2*x;}
#endif

int main (){cout<<f()<<endl;}
```

(nom du fichier : `Chapitre3/prog27.cpp`*)*

Exercice 27 (Classes, manipulation de namespace)**.** *Le fichier* `exemple_include_et_namespace.h`*, ainsi que la notion de bloc d'instructions et le programme qui suit, permettent de comprendre le fonctionnement de* `namespace`*. Que va afficher le programme ?*

Fichier `exemple_include_et_namespace.h` :

```
namespace A {// declaration de l'espace de nom
int n ; // variable dans l'espace de nom
class point { // classe dans l'espace de nom
        double x, y ;
        public :
        point( double x , double y ) : x(x ), y(y ) {}
        void afficher() {std::cout<<x<<" "<<y<<std::endl;}}
}
```

(nom du fichier : `Chapitre3/exemple_include_et_namespace.h`*)*

```
#include <iostream>
#include "exemple_include_et_namespace.h"
using namespace std;

int main() {
        { A::n = 1 ;
        A::point t(5,8) ;
        t.afficher(); }
        { using namespace A ;
        n = 1 ;
        point t(3,4);
        t.afficher(); } }
```

(nom du fichier : `Chapitre3/prog28.cpp`*)*

2 Surcharge d'opérateurs

2.1 Introduction : surcharge de fonctions classiques

Les fonctions, en C++, peuvent être *surchargées* si les types des arguments lèvent toutes les ambiguïtés lors des appels de fonctions. Cela signifie que dans un même programme peuvent coexister deux fonctions différentes

ayant le même nom à condition qu'au moins l'un des types des arguments ou du résultat renvoyé diffère de l'une à l'autre. C'est ici qu'apparaît l'intérêt d'un langage fortement typé comme `C` ou `C++`.

D'ailleurs, de nombreuses fonctions (ou opérateurs[3]) couramment utilisées sont déjà surchargées : `+`, `*`, `+`, `int()`...

Exercice 28 (Surcharge de fonctions classiques)**.** *Lesquels de ces 3 programmes vont compiler ? Dans ce cas, que rendront-ils ?*

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

double f(double x) {return 2*x;};

double f(int x) {return 4*x;};

int main(){double x=1; cout<<f(x)<<endl;}
```

(nom du fichier : `Chapitre3/prog29.cpp`*)*

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

double f(double x) {return 2*x;};

double f(double x, int y=0) {return 4*x+y;};

int main(){double x=1; int y=6; cout<<f(x,y)<<endl;}
```

(nom du fichier : `Chapitre3/prog30.cpp`*)*

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

double f(double x) {return 2*x;};

double f(double x, int y=0) {return 4*x+y;};

int main(){double x=1; cout<<f(x)<<endl;}
```

(nom du fichier : `Chapitre3/prog31.cpp`*)*

Tous les opérateurs classiques peuvent s'étendre (se surcharger) aux classes, à condition que l'on écrive la surcharge. Certains d'entre eux (les opérateurs non binaires) se surchargent comme des méthodes, d'autres (les opérateurs binaires) comme des fonctions amies.

2.2 Surcharge de l'opérateur d'affectation =, pointeur this

2.2.1 Surcharge de l'opérateur d'affectation =

L'opérateur d'affectation `=` est celui qui permet, lorsque `x` et `y` sont des objets d'un même type, d'affecter à `x` la valeur de `y` en utilisant la syntaxe

```
x=y;
```

(lorsque `x` et `y` n'ont pas le même type, c'est parfois possible aussi, via des conversions, comme entre les objets `double` et les objets `int`). Attention, il s'agit dans `x` d'effacer proprement l'ancienne valeur (nettoyage de la mémoire) et ensuite d'y créer une duplication du contenu de `y` : ce n'est pas seulement une copie de l'adresse globale.

L'opérateur d'affectation ne se contente pas d'affecter, il rend une variable. En effet la commande `x=y` procède à l'affectation de la valeur de `y` à `x`, mais cette commande rend aussi une valeur en elle même (la valeur commune de `x` et `y`), ce qui permet d'écrire `t=x=y` ou `s=t=x=y`... (toutes les variables sont à chaque fois affectées de la valeur de `y`).

De même que pour les constructeurs, un opérateur d'affectation par défaut est toujours généré par le compilateur (cf le programme suivant).

3. Un *opérateur* est une opération à deux arguments `x` et `y`, qui se note `x` *symbole* `y`, comme `+`, `*`, `=`...

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

class point {public : double abs;
        void translate(double x) {abs+=x;};
        void affiche() {cout<<abs<<endl;};};

int main(){point p,q,r;
        p.translate(2.5);
        r=q=p;
        q.affiche();
        r.affiche();}
```

(nom du fichier : `Chapitre3/prog32.cpp`*)*

Néanmoins, il n'est pas toujours satisfaisant lorsque certains membres sont des pointeurs (cf le programme suivant) car il copie une adresse sans créer une copie du contenu qui se cache derrière et il peut laisser des fuites de mémoire.

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

class p_int {public : int *p;
        p_int(int n=0) : p(new int(n)) {};
        void ajoute(int m) {(*p)+=m;};
        void affiche_contenu() {cout<<*p<<endl;};};

int main(){p_int r(5), q;
        q=r;
        r.affiche_contenu();
        q.affiche_contenu();
        r.ajoute(2);
        r.affiche_contenu();
        q.affiche_contenu();}
```

(nom du fichier : `Chapitre3/prog33.cpp`*)*

Dans ce cas, on écrit la surcharge de cet opérateur dans la classe concernée. La syntaxe est toujours du même type :

```
ma_classe& operator=(const ma_classe& x) {
.
.
.
return *this;};
```

2.2.2 Le pointeur `this`

Dans une méthode d'une classe, `this` désigne un pointeur vers l'objet courant (les méthodes peuvent donc retourner l'objet courant, une référence à celui-ci ou son adresse). En effet, quand l'on appelle une méthode du type `x.faire(y)`, ce sont deux "arguments" qui sont utilisés : l'objet `x` sur lequel la méthode est appelée et l'éventuel argument complémentaire `y`. Lors de la définition de la méthode néanmoins, on ne connaît pas le nom de l'objet qui sera utilisé plus tard mais on peut avoir à y faire référence : on utilise alors `this` comme pointeur sur cet objet.

Exemple :

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

class p_int {public : int *p;
        p_int(int n=0) : p(new int(n)) {};
        p_int& operator=(const p_int& r) {*p=*(r.p);
                return *this;};
        void ajoute(int m) {(*p)+=m;};
        void affiche_contenu() {cout<<*p<<endl;};};

int main(){p_int r(5), q;
        q=r;
        r.affiche_contenu();
        q.affiche_contenu();
        r.ajoute(2);
        r.affiche_contenu();
        q.affiche_contenu();}
```

(nom du fichier : `Chapitre3/prog34.cpp`*)*

2.2.3 Nécessité de l'usage de `this` dans l'opérateur d'affectation

Bien qu'on n'écrive jamais `x=x;` dans un programme, il est possible par le jeu des références que le même objet en mémoire se retrouve à gauche et à droite du signe égal sans que l'on s'en rende directement compte à l'écriture du code :

```
int main() {
                double c=3;
                double & x=c;
                double & y=c;
                x=y;
}
```

(nom du fichier : `Chapitre3/prog35.cpp`*)*

Lors du nettoyage de l'ancienne valeur du membre de gauche dans `=`, si on ne prend pas de précaution, il est alors possible de tout perdre sur l'objet d'origine ! Pour pallier cet inconvénient, dès que les objets contiennent des allocations mémoires `new` derrière des pointeurs, on opte systématique pour le schéma suivant pour une classe `C` quelconque :

```
C & operator=(const C & y)      {
        // 1/ test si l'objet sous-jacent est le meme:
        if (this == &y) return *this;
        // 2/ Nettoyage des champs de this->...
        ...;
        // 3/ copie des champs de y dans les champs de *this
        ...;
        return *this;
}
```

(nom du fichier : `Chapitre3/prog36.cpp`*)*

Exercice 29. *Que donne le programme suivant ? Que donnerait-t-il si la ligne 15 était absente ?*

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

class Tableau {
                int n;
                int * t;
        public:
                Tableau(int u0=0,int u1=0,int u2=0): n(3), t(new int[3])
                        {t[0]=u0; t[1]=u1; t[2]=u2;}
                Tableau & operator=(const Tableau &);
                int operator [](const int i) const {return t[i];};
};

Tableau & Tableau::operator=(const Tableau & V){
        if (this == &V) return *this;
// purge de la memoire:
        delete [] this->t;
// copie de V:
        this->n= V.n;
        this->t= new int[n]();// les parentheses initialisent a zero
        for (int i = 0; i < n; i++) this->t[i]=V.t[i];
        return *this;
}

int main(void) {
        Tableau T(1,2,3);
        Tableau & U=T;
        Tableau & V=T;
        U=V;
        cout << "T: " <<T[0] << " " << T[1] << " " << T[2] << endl;
}
```

(nom du fichier : `Chapitre3/prog37.cpp`*)*

En conclusion, la première ligne de test sur le `this` est fondamentale.

2.3 Surcharge des opérateurs usuels +, −, *...

Ces opérateurs s'implémentent dans les classes grâce à des fonctions amies.
Le prototype doit être du type suivant

```
friend ma_classe operator+ (const ma_classe&, const ma_classe&)
```

Exemple :

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

class complex {public : double Re, Im;
      complex(double x=0, double y=0) : Re(x), Im(y) {};
      friend complex operator+(const complex &, const complex &);
};

complex operator+(const complex & z1, const complex &z2) {
        return complex(z1.Re+z2.Re, z1.Im+z2.Im);};

int main(){complex z1(1,1), z2(3,4);
        cout<<(z1+z2).Re<<endl;}
```

(nom du fichier : `Chapitre3/prog38.cpp`*)*

2.4 Surcharge des opérateurs usuels +=, −=, *=...

Ces opérateurs s'implémentent dans les classes comme des méthodes.
Le prototype doit être du type suivant :

```
ma_classe& operator+= (const ma_classe&)
```

et le rendu doit être `*this`, de façon à permettre la transitivité.

Exemple :

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

class complex {public : double Re, Im;
      complex(double x=0, double y=0) : Re(x), Im(y) {};
      complex& operator+=(const complex &z){Re+=z.Re;Im+=z.Im;return *this;};
};

int main(){complex z1(1,1), z2(3,4);
        cout<<(z1+=z2).Re<<endl;}
```

(nom du fichier : `Chapitre3/prog39.cpp`*)*

2.5 Surcharge des opérateurs de flux « et »

Les opérateurs de flux `<<` et `>>` sont des opérateurs binaires. Afin de pouvoir écrire (pour le flux `<<`) ou lire (pour le flux `>>`) les objets d'une classe sur la console ou un fichier externe, on les surcharge donc comme des fonctions amies. Les prototypes doivent être les suivants :

```
friend ostream& operator<< (ostream&, const ma_classe&)
```

et

```
friend istream& operator>> (istream&, ma_classe&)
```

(attention, pas de `const` en lecture !).

Exemple :

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

class complex {public : double Re, Im;
      complex(double x=0, double y=0) : Re(x), Im(y) {};
      friend ostream& operator<< (ostream&, const complex&);
      friend istream& operator>> (istream& , complex&);
};

ostream& operator<< (ostream& flux_de_sortie, const complex& z) {
        flux_de_sortie<<z.Re<<"+i*"<<z.Im<<endl;
        return flux_de_sortie;//ce choix permet les expressions
                              //du type cout<<z1<<z2<<...
};

istream& operator>> (istream& flux_de_lecture, complex& z) {
        double x,y;
        flux_de_lecture>>x>>y;
        z.Re=x; z.Im=y;
        return flux_de_lecture;//idem
};

int main(){complex z1, z2;
        cout<<z1;
        cin>>z2;
        cout<<z2.Re*z2.Im<<endl;}
```

(nom du fichier : `Chapitre3/prog40.cpp`*)*

Dans l'affichage des informations, le programmeur est maître du jeu. Dans la lecture de l'information, au contraire, le programmeur doit se prémunir des erreurs d'entrée du futur utilisateur qui *a le droit de se tromper ou de ne pas savoir* : ainsi, derrière `>>`, il est nécessaire de mettre toute une couche de vérification pour être certain que l'information entrée a été comprise et est utilisable.

2.6 Autres opérateurs que l'on peut surcharger

On peut, de même, surcharger les opérateurs de comparaison `==`, `!=`, `<`, de conversion, d'incrémentation `++`...

2.7 Exercices sur la surcharge d'opérateurs

Exercice 30. *Surcharger les opérateurs* `+`, `*`, `+=`, `*=`, `++`, `--`, `<<`, `>>`, `<`, `>`, `==`, *etc. dans la classe suivante :*

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

class ptr_double {double* p;
public :          ptr_double(double x=0) : p(new double){*p=x;}
                  ptr_double(const ptr_double& q): p(new double(*q.p)) {}
                  ~ptr_double(){delete p;}
                  double contenu() {return *p;} };
```

(nom du fichier : `Chapitre3/prog41.cpp`*)*

Exercice 31 (Surcharge des opérateurs)**.** *Quelles versions des fonctions doit-on décommenter dans le programme suivant ? Aura-t-on alors* `c==d` *? Pourrait-on, avec cette fonction* `main`*, changer*

```
friend ostream& operator<<(ostream &, const Complex &);
```

pour

```
friend void operator<<(ostream &, const Complex &);
```

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

class Complex { double x,y;
  public:
    Complex(double x=0, double y=0) : x(x), y(y) {};
    friend ostream& operator<<(ostream &, const Complex &);
    friend istream& operator>>(istream &, Complex &);
    friend bool operator==(const Complex &f, const Complex &g);};

//ostream &operator<<(ostream &out, const Complex &z) {
//      out << z.x <<" "<< z.y<<"\t";
//    return out;};
//
//ostream &operator<<(ostream &out, const Complex &z) {
//      out << z <<"\t";
//    return out;};
//
//istream &operator>>(istream &in, Complex &z) {
//      double a, b;
//    in >> a >> b;
//    z.x=a; z.y=b;
//    return in;};
//
//istream &operator>>(istream &in, Complex &z) {
//    in >> z.x >> z.y;
//    return in;};
//
//bool operator==(const Complex &f, const Complex &g) {
//      return (f.x==g.x && f.y==g.y);};
//
//bool operator==(const Complex &f, const Complex &g) {
//      return (f==g);};

int main() {Complex a(2,3), b(5,1), c(0,0), d, e;
        cout<<a<<b;
        if (c==d) cin>>e;}
```

(nom du fichier : `Chapitre3/prog42.cpp`*)*

3 Exercices sur le chapitre

Exercice 32 (Classes, manipulation de variables aléatoires). *On cherche à simuler la v.a.* $\sum_{i=1}^{N} X_i$*, où* N *suit une loi de Poisson de paramètre* 1 *et* $(X_i)_{i\geq 1}$ *est une suite de v.a.i.i.d. de loi uniforme sur* $[0,1]$*, indépendante de* N*. Le programme suivant va-t-il afficher des réalisations d'une telle v.a. ? (on ne cherchera pas à vérifier la fonction* `double operator()()` *de la classe* `Poisson`*, qui est juste). Comment modifier le programme pour que l'affichage corresponde à ce que l'on souhaite ?*

Remarque : en changeant a en $a = e^{-\lambda}$, cette méthode permet de simuler une loi de Poisson avec $\lambda > 0$.

```
#include <iostream>
#include <cmath>
#include <ctime>
#include <cstdlib>
using namespace std;

inline double unif_rand() {return (random() + 0.5)/(RAND_MAX + 1.0);};

class Poisson {public : Poisson() {}
         double operator()(){
                double a=exp(-1);       int r=0.;
                double U=unif_rand();
                while (U>a){U = U * unif_rand();r++;}
                return double(r);}};

int main() {srandom(time(NULL));
        Poisson P;      double s=0;
        for(int i=0; i<P();i++) {s+=unif_rand();}
        cout<<s<<endl;}
```

(nom du fichier : `Chapitre3/prog43.cpp`*)*

Exercice 33 (classes, surcharges). *Construire une classe* `Fraction` *déclarée de la façon suivante et la tester.*

```
#ifndef FRACTION_HPP
#define FRACTION_HPP

class Fraction {
public:
        Fraction(int n = 0, int d = 1, char *s="");
        Fraction(const Fraction &f);
        ~Fraction();
        void afficher () const;
        double valeur_numerique() const;
        int lire_num() const;
        int lire_denom() const;
        void change_num(const int n) ;
        void change_denom(const int d) ;
        void change_nom(const char *s) ;
        void reduction() ;
        Fraction& Fraction::operator=(const Fraction &z) ;
        friend Fraction operator+(const Fraction &, const Fraction &);
        friend Fraction operator-(const Fraction &, const Fraction &);
        friend Fraction operator*(const Fraction &, const Fraction &);
        friend Fraction operator/(const Fraction &, const Fraction &);
        friend std::ostream& operator<<(std::ostream &, const Fraction &);
        friend std::istream& operator>>(std::istream &, Fraction &);
        Fraction valeur_absolue();
        friend bool operator==(const Fraction &f, const Fraction &g);
        bool a_meme_valeur_numerique_que(const Fraction &g);
private:
        int num, denom;
        char *nom;
};
#endif
```

(nom du fichier : `Chapitre3/Fraction.hpp`*)*

Exercice 34 (Création d'une classe, donnée membre constante, fonction amie et membres statiques)**.** *Construire une classe* `Mobile` *déclarée de façon suivante (voir les paragraphes correspondants du cours de H. Garreta). La tester. Est-il possible d'ajouter à cette classe une méthode du prototype suivant ?*

```
Mobile& Mobile::operator=(const Mobile &z)
```

```
#ifndef MOBILE_HPP
#define MOBILE_HPP

class Mobile {
public :
        static int nombre_de_mobiles;
        Mobile(double x = 0, double y = 0, double v_x=0, double v_y=0,
        int num=0, char *s="");
        Mobile(const Mobile &f);
        ~Mobile();
        void afficher () const;
        double vitesse_numerique() const; //norme du vecteur vitesse
        double lire_x() const;
        double lire_y() const;
        double lire_vitesse_x() const;
        double lire_vitesse_y() const;
        double lire_numero() const;
        void deplace(const double u, const double v) ;// deplacement
        // selon le vecteur (u,v)
        void accelere(const double u, const double v) ;// addition
        // du vecteur (u,v) au vecteur vitesse
        void tourne_droite() ;// rotation -pi/2 du vecteur vitesse
        void tourne_gauche() ;// rotation +pi/2 du vecteur vitesse
        void change_nom(const char *s) ;
        friend std::ostream& operator<<(std::ostream &, const Mobile &);
        friend std::istream& operator>>(std::istream &, Mobile &);
        friend bool operator==(const Mobile &f, const Mobile &g);
        double Mobile::operator[](int n);
        friend double distance_M(const Mobile &f, const Mobile &g);
        //ne pas utiliser le nom distance,
        // qui correspond deja a une fonction de la STL
        friend bool sont_au_meme_endroit(const Mobile &f, const Mobile &g);
        friend bool avancent_parallelement(const Mobile &f, const Mobile &g);
private:
        double position_x, position_y, vitesse_x, vitesse_y;
        const int numero_serie;
        char *nom;
};
#endif
```

(nom du fichier : `Chapitre3/Mobile.hpp`*)*

Exercice 35 (Classes amies)**.** *La classe* `Particule`*, dont la déclaration est ci-dessous, est une simplification de la classe* `Mobile`*. Écrire cette classe et la tester. Les classes* `Dimere` *et* `Polymere` *sont des classes amies de la classe* `Particule`*. Les écrire et les tester.*

Rappel : Pour compiler un programme `test_Polymere_TP3.cpp` qui ferait appel aux classes `Particule` et `Polymere` définies dans des fichiers `Particule.cpp` et `Polymere.cpp`, la démarche est la suivante.

a) on compile les classes `Particule` et `Polymere` par les commandes

```
g++ -c Particule.cpp
```

et

```
g++ -c Polymere.cpp
```

b) on tape

```
g++ Particule.o Polymere.o test_Polymere_TP3.cpp -o test_Polymere_TP3.exe
```

On peut aussi écrire le fichier suivant, appelé `Makefile`, puis taper `make`

```
CPP = g++
//CPPFLAGS = -I/Users/Florent/Informatique/boost_1_44_0/
//LDFLAGS = -L/Users/Florent/Informatique/boost_1_44_0/stage/lib/

PROG = mon_ex
HDRS = Polymere.hpp Particule.hpp
SRCS = test_Polymere_TP3.cpp Polymere.cpp Particule.cpp
OBJS = $(SRCS:.cpp=.o)

$(PROG) : $(OBJS)
	$(CPP) $(LDFLAGS) $(OBJS) -o $(PROG)

test_Polymere.o : test_Polymere_TP3.cpp Polymere.hpp Particule.hpp
Polymere.o : Polymere.cpp Polymere.hpp Particule.hpp
Particule.o : Particule.cpp Particule.hpp


mon_clean :
	rm -f $(PROG) $(OBJS)
```

```
#ifndef Dimere_HPP
#define Dimere_HPP

class Dimere {
public :
        static int nombre_de_dimeres;
        Dimere(const Particule &p, const Particule &q);
        Dimere(const Dimere &f);
        ~Dimere();
        void afficher() const;
        void inverser();// permute Part1 et Part2
        double longueur() const;
        void projette_sur_x();// projette ortho. les 2 part. sur l'axe des x
private:
        Particule Part1, Part2;
};
#endif
```

(nom du fichier : Chapitre3/Dimere.hpp*)*

```
#ifndef Polymere_HPP
#define Polymere_HPP

class Polymere {
public :
        static int nombre_de_polymeres;
        Polymere();
        Polymere(const Particule &p);
        Polymere(const Polymere &pol, const Particule &p);
        Polymere(const Particule &p, const Polymere &pol);
        Polymere(const Polymere &f);
        ~Polymere();
        int Nombre_monomeres () const;
        void afficher() const;
private:
        int n;
        Particule *nuage;
};
#endif
```

(nom du fichier : Chapitre3/Polymere.hpp*)*

```
#ifndef Particule_HPP
#define Particule_HPP

class Particule {
public :
        static int nombre_de_particules;
        Particule(double x = 0, double y = 0, double v_x=0, double v_y=0);
        Particule(const Particule &f);
        ~Particule();
        void afficher () const;
        double vitesse_numerique() const; //norme du vecteur vitesse
        double lire_x() const;
        double lire_y() const;
        double lire_vitesse_x() const;
        double lire_vitesse_y() const;
        void deplace(const double u, const double v) ;// deplacement
        // selon le vecteur (u,v)
        void accelere(const double u, const double v) ;// addition
        // du vecteur (u,v) au vecteur vitesse
        void tourne_droite() ;// rotation -pi/2 du vecteur vitesse
        void tourne_gauche() ;// rotation +pi/2 du vecteur vitesse
        Particule& operator=(const Particule &z);
        friend std::ostream& operator<<(std::ostream &, const Particule &);
        friend std::istream& operator>>(std::istream &, Particule &);
        friend bool operator==(const Particule &f, const Particule &g);
        double operator[](int n);
        friend double distanceP(const Particule &f, const Particule &g);
        friend bool sont_au_meme_endroit(const Particule &f,
                                                const Particule &g);
        friend bool avancent_parallelement(const Particule &f,
                                                const Particule &g);
        friend class Polymere;
        friend class Dimere;
private:
        double position_x, position_y, vitesse_x, vitesse_y;
};
#endif
```

(nom du fichier : Chapitre3/Particule.hpp*)*