

TP 3 : Introduction aux classes, surcharge d'opérateurs, classes et fonctions amies

On créera et manipulera les classes 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`. On écrit alors le fichier `Point.hpp` suivant (appelé *fichier d'en-tête*).

```
1 #ifndef POINT_HPP
2 #define POINT_HPP
3
4 class Point {
5     public:
6         Point(double x = 0, double y = 0); //constructeur
7         Point(const Point &p); //constructeur par copie (attention :
8         // son argument est toujours une référence
9         // vers un objet constant)
10        ~Point(); //destructeur
11        void afficher ();
12        double distance (Point p);
13
14    private:
15        double abs, ord;
16 };
17 #endif
```

NB : Un mot pour expliquer la présence de `#ifndef POINT_HPP`, `#define POINT_HPP` et `#endif` : on peut mettre plusieurs header en en-tête 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 à ce qui suit `#endif`. On définit alors les méthodes dans le fichier `Point.cpp` suivant.

```
1 #include <cmath>
2 #include <iostream>
3 #include "Point.hpp"
4 using namespace std;
5
6 Point::Point(double x, double y) : abs(x), ord(y) {};
7
8 Point::Point(const Point &p) : abs(p.abs), ord(p.ord) {};
9
10 Point::~~Point() {};
11
12 void Point::afficher() {
13     cout<<"("<<abs<<" "<<ord<<" "<<endl;};
14
15 double Point::distance(Point p) {
16     double d=sqrt((abs-p.abs)*(abs-p.abs)+(ord-p.ord)*(ord-p.ord));
17     return d;
18 };
```

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,

```
1 #include <iostream>
2 #include <cmath>
3 #include "Point.hpp"
4 using namespace std;
5
6 int main() {
7     Point p=Point(1,1);
8     Point q;
9     Point r=Point(q);
10    q.afficher();
11    cout<<p.distance(r)<<endl;
12    return 0;
13 }
```

on compile comme suit :

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

1 *Que va afficher l'exécution du programme `test-Point` ce dessus ?*

Rappel : Pour "allouer de la mémoire" (c'est à dire remplir des cases mémoire) à un pointeur `p` vers un objet de type `nom_du_type` (et éventuellement le créer), on utilise :

```
nom_du_type *p = new nom_du_type
```

On peut éventuellement initialiser la valeur pointée par une syntaxe du type :

```
double *p = new double(3.14)
```

Si l'on veut allouer un tableau de n objets, on utilise une syntaxe du type :

```
char *p = new char[n] ( $n$  peut être inconnu à la compilation, mais il doit l'être à l'exécution).
```

On restitue ensuite la mémoire de la façon suivante : si `p` a été obtenu par un appel de `new`, on écrit :

```
delete p si ce que point p n'est pas un tableau,
```

et

```
delete [] p si p est un tableau.
```

Le fonctionnement précis de `new` est assez délicat, les exercices suivants permettent de mieux le comprendre.

2 *Que va afficher l'exécution du programme `test_constr_defaut.cpp` ce dessous ?*

```
1 #include <cmath>
2 #include <iostream>
3 using namespace std;
4
5 class Complex {
6 public :
7     Complex(double x=0, double y=0) : Re(x), Im(y) {};
8     double partie_reelle() {return Re;}
9 private:
10    double Re, Im;
11 };
12
13 int main() {
14    Complex *p=new Complex;
15    cout<<p->partie_reelle()<<endl;
16
17    Complex *q=new Complex[5];
18    cout<<q[2].partie_reelle()<<endl;
19    return 0;}
20 }
```

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

```
1 #include <cmath>
2 #include <iostream>
3 using namespace std;
4
5 class Complex {
6 public :
7     Complex(double x, double y) : Re(x), Im(y) {};
8 private:
9     double Re, Im;
10 };
11
12 int main() {
13     // possibilite 1 :
14     // Complex z;
15
16     // possibilite 2 :
17     // Complex *p;
18
19     // possibilite 3 :
20     // Complex *q=new Complex;
21
22     // possibilite 4 :
23     // Complex *q=new Complex[5];
24     return 0;}
```

4 (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 `Point` 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 : la création et la copie de chaînes de caractères est résumée dans le petit programme ci-dessous.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     char *c = new char;
6     char d[]="abcdef";
7     cout<< d <<endl;
8     cout<< strlen(d) <<endl; //longueur de la chaine
9     strcpy(c,d); //copie de la chaine c dans la chaine d
10    cout<< c <<endl;
11    cout<<c[0]<<" " <<c[3]<<endl;
12    return 0;};
```

5 (Création d'une classe, surcharge d'opérateurs) Construire une classe `Fraction` déclarée de façon suivante. La tester.

```

2  #ifndef FRACTION_HPP
3  #define FRACTION_HPP
4
5  class Fraction {
6  public:
7      Fraction(int n = 0, int d = 1, char *s="");
8      Fraction(const Fraction &f);
9      ~Fraction();
10     void afficher () const;
11     double valeur_numerique() const;
12     int lire_num() const;
13     int lire_denom() const;
14     void change_num(const int n) ;
15     void change_denom(const int d) ;
16     void change_nom(const char *s) ;
17     void reduction() ;
18     Fraction& Fraction::operator=(const Fraction &z) ;
19     friend Fraction operator+(const Fraction &, const Fraction &);
20     friend Fraction operator-(const Fraction &, const Fraction &);
21     friend Fraction operator*(const Fraction &, const Fraction &);
22     friend Fraction operator/(const Fraction &, const Fraction &);
23     friend std::ostream& operator<<(std::ostream &, const Fraction &);
24     friend std::istream& operator>>(std::istream &, Fraction &);
25     Fraction valeur_absolue();
26     friend bool operator==(const Fraction &f, const Fraction &g);
27     bool a_meme_valeur_numerique_que(const Fraction &g);
28 private:
29     int num, denom;
30     char *nom;
31 };
32 #endif

```

6 (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)

```

2  #ifndef MOBILE_HPP
3  #define MOBILE_HPP
4  class Mobile {
5  public :
6      static int nombre_de_mobiles;
7      Mobile(double x = 0, double y = 0, double v_x=0, double v_y=0,
8          int num=0, char *s="");
9      Mobile(const Mobile &f);
10     ~Mobile();
11     void afficher () const;
12     double vitesse_numerique() const; //norme du vecteur vitesse
13     double lire_x() const;
14     double lire_y() const;
15     double lire_vitesse_x() const;
16     double lire_vitesse_y() const;
17     double lire_numero() const;
18     void deplace(const double u, const double v) ;// deplacement
19     // selon le vecteur (u,v)
20     void accelere(const double u, const double v) ;// addition
21     // du vecteur (u,v) au vecteur vitesse
22     void tourne_droite() ;// rotation -pi/2 du vecteur vitesse
23     void tourne_gauche() ;// rotation +pi/2 du vecteur vitesse
24     void change_nom(const char *s) ;
25     friend std::ostream& operator<<(std::ostream &, const Mobile &);
26     friend std::istream& operator>>(std::istream &, Mobile &);
27     friend bool operator==(const Mobile &f, const Mobile &g);
28     double Mobile::operator[](int n);
29     friend double distance_M(const Mobile &f, const Mobile &g);
30     //ne pas utiliser le nom distance, qui correspond deja a une fonction de la STL
31     friend bool sont_au_meme_endroit(const Mobile &f, const Mobile &g);
32     friend bool avancent_parallement(const Mobile &f, const Mobile &g);
33 private:
34     double position_x, position_y, vitesse_x, vitesse_y;
35     const int numero_serie;
36     char *nom;
37 };
38 #endif

```

7 (Classes amies) A rendre, par mail, en binôme, à florent.benaych@gmail.com. Sujet du mail : C++IFMA, Prénom 1 NOM 1, Prénom 2 NOM 2.

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.cpp qui ferait appel aux classes Particule et Polymere définies dans des fichiers Particule.cpp et Polymere.cpp et compilées antérieurement, on tape

```
> g++ Particule.o Polymere.o test_Polymere.cpp -o test_Polymere
```

```

2 #ifndef Particule_HPP
3 #define Particule_HPP
4 class Particule {
5 public :
6     static int nombre_de_particules;
7     Particule(double x = 0, double y = 0, double v_x=0, double v_y=0);
8     Particule(const Particule &f);
9     ~Particule();
10    void afficher () const;
11    double vitesse_numerique() const; //norme du vecteur vitesse
12    double lire_x() const;
13    double lire_y() const;
14    double lire_vitesse_x() const;
15    double lire_vitesse_y() const;
16    void deplace(const double u, const double v) ;// deplacement
17    // selon le vecteur (u,v)
18    void accelere(const double u, const double v) ;// addition
19    // du vecteur (u,v) au vecteur vitesse
20    void tourne_droite() ;// rotation -pi/2 du vecteur vitesse
21    void tourne_gauche() ;// rotation +pi/2 du vecteur vitesse
22    Particule& operator=(const Particule &z);
23    friend std::ostream& operator<<(std::ostream &, const Particule &);
24    friend std::istream& operator>>(std::istream &, Particule &);
25    friend bool operator==(const Particule &f, const Particule &g);
26    double operator[](int n);
27    friend double distanceP(const Particule &f, const Particule &g);
28    friend bool sont_au_meme_endroit(const Particule &f,
29                                     const Particule &g);
30    friend bool avancent_parallement(const Particule &f,
31                                     const Particule &g);
32    friend class Polymere;
33    friend class Dimere;
34 private:
35    double position_x, position_y, vitesse_x, vitesse_y;
36 };
37 #endif

```

```

2 #ifndef Dimere_HPP
3 #define Dimere_HPP
4 class Dimere {
5 public :
6     static int nombre_de_dimeres;
7     Dimere(const Particule &p, const Particule &q);
8     Dimere(const Dimere &f);
9     ~Dimere();
10    void afficher() const;
11    void inverser(); // permute Part1 et Part2
12    double longueur() const;
13    void projette_sur_x(); // projette ortho. les 2 part. sur l'axe des x
14 private:
15    Particule Part1, Part2;
16 };
17 #endif

```

```
1 #ifndef Polymere_HPP
2 #define Polymere_HPP
3
4 class Polymere {
5 public :
6     static int nombre_de_polymeres;
7     Polymere();
8     Polymere(const Particule &p);
9     Polymere(const Polymere &pol, const Particule &p);
10    Polymere(const Particule &p, const Polymere &pol);
11    Polymere(const Polymere &f);
12    ~Polymere();
13    int Nombre_monomeres () const;
14    void afficher() const;
15 private:
16    int n;
17    Particule *nuage;
18 };
19 #endif
```