Programmation C++ (débutant)/Les classes

Le cours du chapitre 11 : Les classes

Une évolution des structures

Une fois introduite la notion de structures, on s'aperçoit qu'il est totalement naturel de créer des fonctions permettant de manipuler ces structures. La notion de classe est donc une notion plus puissante que la notion de structures. Une classe va permettre de regrouper en une seule entité des données membres et des fonctions membres appelées méthodes.

Cependant, contrairement au langage C, les structures du C++ permettent cela également. La différence entre une structure et une classe est que les attributs et méthodes d'une structure sont par défaut publics, alors qu'ils sont par défaut privés dans une classe. Une autre différence est que les structures ne peuvent utiliser l'héritage.

Notion de classe

Une classe regroupera donc :

- des données membres.
- des méthodes membres qui seront des fonctions.

Un premier exemple de classe

- On veut manipuler des points définis par une abscisse et une ordonnée (des réels).
- Sur un point, on peut calculer la distance entre 2 points et le milieu de 2 points.
- Nous allons donc définir une classe Point définie par un fichier .h et un fichier .cpp.

Exemple 1 : la classe Point

Le fichier Point.h

```
#ifndef POINT_H
#define POINT_H

class Point
{
  public:
    double x,y;
    double distance(const Point &P);
    Point milieu(const Point &P);
};
#endif
```

Explications

On définit dans ce fichier la classe Point : elle contient 2 données de type double x et y et 2 méthodes membres distance qui calcule la distance entre ce point et un autre Point et milieu qui calcule le milieu du segment composé de ce point et d'un autre Point.

On remarque l'utilisation des directives de compilation #ifndef, #define et #endif pour gérer les inclusions multiple du fichier header.

Le fichier Point.cpp

```
#include "Point.h"
#include <cmath>

double Point::distance(const Point &P)
{
    double dx,dy;
    dx = x - P.x;
    dy = y - P.y;
    return sqrt(dx*dx + dy*dy);
}

Point Point::milieu(const Point &P)
{
    Point M;
    M.x = (P.x+x) /2;
    M.y = (P.y+y) /2;
    return M;
}
```

Explications

- Il contient l'implémentation de chaque méthode de la classe Point.
- On fait précéder chaque méthode de Point::
- On a inclut le fichier cmath afin de pouvoir utiliser la fonction sqrt de cmath (racine carrée).
- A l'intérieur de la classe Point, on peut accéder directement à l'abscisse du point en utilisant la donnée membre x.
- On peut accéder à l'abscisse du paramètre P d'une méthode en utilisant P.x.

Le fichier main.cpp

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

int main()
{
    Point A, B, C;
    double d;
    cout << "SAISIE DU POINT A" << endl;
    cout << "Tapez l'abscisse : "; cin >> A.x;
    cout << "Tapez l'ordonnée : "; cin >> A.y;
    cout << endl;
```

```
    cout << "SAISIE DU POINT B" << endl;
    cout << "Tapez l'abscisse : "; cin >> B.x;
    cout << "Tapez l'ordonnée : "; cin >> B.y;
    C = A.milieu(B);
    d = A.distance(B);
    cout << endl;
    cout << "MILIEU DE AB" << endl;
    cout << "L'abscisse vaut : " << C.x << endl;
    cout << "L'ordonnée vaut : " << C.y << endl;
    cout << endl;
    cout << "La distance AB vaut :" << d << endl;
    return 0;
}
```

Explications

- Une fois inclus le fichier d'en-tête Point.h, on peut définir 3 points A, B et C.
- A, B et C sont 3 objets qui sont des instances de la classe Point.
- Les données membres étant publiques, on peut accéder à l'abscisse et à l'ordonnée de A en dehors de la classe en écrivant A.x et A.y.
- Les méthodes membres distance et milieu étant publiques, on peut écrire directement A.milieu(B) ou A.distance(B).

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
SAISIE DU POINT A
Tapez l'abscisse : 3.2
Tapez l'ordonnée : 1.4
SAISIE DU POINT B
Tapez l'abscisse : 5
Tapez l'ordonnée : 6
MILIEU DE AB
L'abscisse vaut : 4.1
L'ordonnée vaut : 3.7
La distance AB vaut : 4.93964
```

Encapsulation

- Il faut éviter de donner un accès extérieur aux données membres d'un objet quelconque.
- On va interdire l'accès à certaines données membres d'une classe ou certaines méthodes en utilisant le mot clé `private`.
- On ne peut accéder à une variable (ou une méthode membre) privée que par l'intérieur de la classe.
- par contre, on peut accéder librement à toutes les données membres ou méthodes membres publiques.
- Cette technique fondamentale permet d'empêcher au programmeur de faire n'importe quoi : il ne pourra accéder à ces données que par les méthodes publiques.

Interface et boîte noire

- Vu de l'extérieur, on ne peut accéder à un objet donné que grâce à ces méthodes publiques.
- Ceci permet entre autre de protéger l'intégrité des données.
- L'ensemble des méthodes publiques est appelée l'interface de l'objet.
- De l'extérieur, l'objet peut être vu comme une boîte noire qui possède une interface d'accès.
- On cache ainsi à l'utilisateur de cette classe comment cette interface est implémentée : seul le comportement de l'interface est important.

Accesseurs et mutateurs

- On pourra accéder aux valeurs des données membres privées grâce à des méthodes spécifiques appelée accesseurs. Les accesseurs seront publics.
- On pourra même modifier ces valeurs grâce à des fonctions spécifiques appelées mutateurs.
- Cela permet au programmeur d'avoir un contrôle complet sur ces données et sur des contraintes en tout genre qu'il veut imposer à ces données.

Exemple 2 : accesseurs et mutateurs

Le fichier Point.h

```
#ifndef POINT_H
#define POINT_H

class Point
{
  public:
    void setX(double x);
    void setY(double y);
    double getX();
    double getY();
    double distance(const Point &P);
    Point milieu(const Point &P);
    void saisir();
    void afficher();

  private:
    double x,y;
};
#endif
```

Explications

- La méthode void setX(double x) est un mutateur qui permet de modifier la donnée membre privée x.
- Idem pour void setY(double y) avec la donnée membre privée y.
- Les méthodes double getX() et double getY() sont des accesseurs qui permettent d'accéder aux valeurs respectives des données membres privées x et y.
- Les méthodes saisir() et afficher() permettent respectivement de saisir et d'afficher les coordonnées des points.

Le fichier Point.cpp

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

void Point::setX(double x)
{
    this->x = x;
}
void Point::setY(double y)
{
    this->y = y;
}

double Point::getX()
{
    return x;
}

double Point::getY()
{
    return y;
}

double Point::distance(const Point &P)
{
    double dx, dy;
    dx = x - P.x;
    dy = y - P.y;
    return sqrt(dx*dx + dy*dy);
}

Point Point::milieu(const Point &P)
{
    Point M;
    M.x = (P.x + x) /2;
    M.y = (P.y + y) /2;
    return M;
}
```

```
void Point::saisir()
{
    cout << "Tapez l'abscisse : "; cin >> x;
    cout << "Tapez l'ordonnée : "; cin >> y;
}

void Point::afficher()
{
    cout << "L'abscisse vaut " << x << endl;
    cout << "L'abscisse vaut " << y << endl;
}
```

Explications

- Dans la méthode setX(...), il y a une utilisation du mot-clé this. Le mot clé this désigne un pointeur vers l'instance courante de la classe elle même. this->x désigne donc la donnée membre de la classe alors que x désigne le paramètre de la méthode void setX(double x);
- Le mutateur double getX(); se contente de renvoyer la valeur de x.

Le fichier main.cpp

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

int main()
{
    Point A, B, C;
    double d;
    cout << "SAISIE DU POINT A" << endl;
    A.saisir();
    cout << endl;
    cout << "SAISIE DU POINT B" << endl;
    B.saisir();
    cout << endl;
    C = A.milieu(B);
    d = A.distance(B);
    cout << "MILIEU DE AB" << endl;
    C.afficher();
    cout << endl;
    cout << "La distance AB vaut :" << d << endl;
    return 0;
}
```

Explications

- On n'a plus le droit d'accéder aux données membres x et y sur les instances de Point A, B et C en utilisant A.x ou B.y : il faut obligatoirement passer là une des méthodes publiques.
- Pour saisir la valeur de A, il suffit d'écrire A.saisir();
- Pour afficher la valeur de A, il suffit d'écrire A.afficher();

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
SAISIE DU POINT A
Tapez l'abscisse : 3.2
Tapez l'ordonnée : 1.4
SAISIE DU POINT B
Tapez l'abscisse : 5
Tapez l'ordonnée : 6
MILIEU DE AB
L'abscisse vaut : 4.1
L'ordonnée vaut : 3.7
La distance AB vaut : 4.93964
```

Utiliser les opérateurs >> et <<

- Pour pouvoir saisir un Point au clavier, on pourrait écrire tout simplement cin >> A; où A est une instance de la classe Point.
- Pour écrire un Point à l'écran, on peut écrire tout simplement cout << B.
- Nous allons utiliser pour cela les fonctions operator>> et operator<<.
- Dans l'exemple 3, on utilisera une manière de procéder assez personnelle sans utiliser de fonctions amies;
- Dans l'exemple 4, nous verrons la méthode qui semble plus classique basée sur les fonctions amies.

Exemple 3 : l'opérateur >> et l'opérateur <<

Fichier Point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
using namespace std;

class Point
{
  public:
    void setX(double x);
    void setY(double y);
    double getX();
    double getY();
    double distance(const Point &P);
    Point milieu(const Point &P);
    void operator>>(ostream &out);
    void operator<<(istream &in);
```

```
  private:
    double x, y;
};
#endif
```

Explications

- On définit une méthode `operator>>(ostream &out);` qui permet d'afficher le point en utilisant le `ostream out` (le plus souvent ce sera `cout`).
- On définit une méthode `operator<<(istream &in);` qui permet de saisir le point au clavier en utilisant le `istream in` (le plus souvent ce sera `cin`).

Le fichier Point.cpp

Les fonctions setX(), setY(), getX(), getY(), distance() et milieu sont identiques à celle de l'exemple 2. On définit les 2 opérateurs de la manière suivante :

```
void Point::operator>>(ostream &out)
{
    out << "L'abscisse vaut " << x << endl;
    out << "L'ordonnée vaut " << y << endl;
}

void Point::operator<<(istream &in)
{
    cout << "Tapez l'abscisse : "; in >> x;
    cout << "Tapez l'ordonnée : "; in >> y;
}
```

Fichier main.cpp

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

int main()
{
    Point A, B, C;
    double d;
    cout << "SAISIE DU POINT A" << endl;
    A << cin;
    cout << endl;
    cout << "SAISIE DU POINT B" << endl;
    B << cin;
    cout << endl;
    C = A.milieu(B);
    d = A.distance(B);
    cout << "MILIEU DE AB" << endl;
    C >> cout;
    cout << endl;
    cout << "La distance AB vaut :" << d << endl;
```

```
    return 0;
}
```

Explications :

On peut directement saisir les coordonnées d'un point par A<<cin ou l'afficher à l'écran par A>>cout.

Exécution :

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
SAISIE DU POINT A
Tapez l'abscisse : 3.2
Tapez l'ordonnée : 1.4
SAISIE DU POINT B
Tapez l'abscisse : 5
Tapez l'ordonnée : 6
MILIEU DE AB
L'abscisse vaut : 4.1
L'ordonnée vaut : 3.7
La distance AB vaut : 4.93964
```

La notation A<<cout

- On peut attacher l'opérateur de << à la classe Point comme dans l'exemple 3.
- Certains préfèrent toutefois écrire de manière plus usuelle : cout<<A;
- Ils argumentent parfois en disant qu'en plus cela permet d'enchaîner les affichages en écrivant : `cout<<A<<endl<<B;`
- Il faudrait donc normalement définir une méthode `operator<<(const Point &A)` sur la classe `ostream`.
- Or la classe `iostream` est déjà écrite !

Les fonctions amies

- Pour pouvoir écrire `cout<<A;` il faut écrire une fonction `ostream & operator<<(ostream &, const Point &P);`
- Le plus pratique est que cette fonction ait le droit d'accéder aux données membres privées de la classe Point.
- On va donc la créer en tant que fonction amie en utilisant le mot-clé friend.
- L'abus de fonctions amies rompt avec le principe d'encapsulation : à utiliser avec précaution.

Exemple 4 : les fonctions amies

Le fichier Point.h

```
#ifndef POINT_H
#define POINT_H
class Point
{
friend istream & operator>>(istream &, Point &P);
friend ostream & operator<<(ostream &, const Point &P);
public:
    void setX(double x);
```

```
    void setY(double y);
    double getX();
    double getY();
    double distance(const Point &P);
    Point milieu(const Point &P);
private:
    double x, y;
};
#endif
```

Explications

- Nous avons défini deux fonctions amies de la classe Point : ce ne sont pas des méthodes membres.
- Leurs définitions sont les suivantes :

```
friend istream & operator>>(istream &, Point &P);
friend ostream & operator<<(ostream &, const Point &P);
```

Elles renvoient respectivement une référence vers un istream et un ostream pour pouvoir enchaîner par exemple : `cout<<A<<B;`

Le fichier Point.cpp

Voici l'implémentation de ces fonctions amies :

```
ostream & operator<<(ostream & out, const Point &P)
{
    out << "L'abscisse vaut " << P.x << endl;
    out << "L'ordonnée vaut " << P.y << endl;
    return out;
}
istream & operator>>(istream & in,Point &P)
{
   cout << "Tapez l'abscisse : "; in >> P.x;
   cout << "Tapez l'ordonnée : "; in >> P.y;
   return in;
}
```

Explications

- Nos 2 fonctions renvoient respectivement in et out par une instruction `return` afin de pouvoir enchaîner les opérations de saisie et d'affichage.
- `cout` ne modifie pas notre `Point` : on passe donc une référence vers un `Point` constant.
- Pour `cin`, il faut passer en paramètre une référence vers un `Point`.

Le fichier main.cpp

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

int main()
{
```

```
    Point A, B, C;
    double d;
    cout << "SAISIE DU POINT A" << endl;
    cin >> A;
    cout << endl << "SAISIE DU POINT B" << endl;
    cin >> B;
    cout << endl;
    C = A.milieu(B);
    d = A.distance(B);
    cout << "MILIEU DE AB" << endl << C << endl;
    cout << "La distance AB vaut :" << d << endl;
    return 0;
}
```

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
SAISIE DU POINT A
Tapez l'abscisse : 3.2
Tapez l'ordonnée : 1.4
SAISIE DU POINT B

Tapez l'abscisse : 5
Tapez l'ordonnée : 6
MILIEU DE AB

L'abscisse vaut : 4.1
L'ordonnée vaut : 3.7
La distance AB vaut : 4.93964
```

Constructeurs et initialisation des données membres

- Par défaut, les données membres d'un objet ne sont pas initialisées. Il existe toutefois un constructeur par défaut qui se contente de créer ces données membres. Il est appelé dès qu'une instance de la classe est crée. **Les données membres sont créées mais ne sont pas initialisées** : elles sont à une valeur aléatoire correspondant à ce qu'il y avait en mémoire à ce moment là.
- Il est vivement recommandé de définir un constructeur par défaut pour une classe donnée qui initialisera les données membres : ainsi, il n'existera pas d'instance de la classe avec des données membres non initialisées.
- On peut également définir d'autres constructeurs qui permettront d'initialiser les données membres d'un objet avec certaines valeurs.
- Le programmeur sera alors certain que ces données sont dans un état cohérent.
- Le constructeur peut éventuellement allouer dynamiquement de la mémoire pour une classe complexe : il faut désallouer cette mémoire dès que l'objet n'existe plus.

Les destructeurs

- Le destructeur est appelé automatiquement dès qu'un objet est détruit. Il peut avoir (entre autres) pour rôle de libérer par exemple la mémoire allouée au cours de l'utilisation de la classe.

Syntaxe des constructeurs et des destructeurs

- Le constructeur par défaut de la classe A sera noté A() . Il ne peut rien renvoyer par un return.
- On peut créer d'autres constructeur qui seront identifiés par leurs paramètres : par exemple un constructeur de la classe A peut s'écrire A(int x, int y).
- Le destructeur de la classe A sera noté ~A(). Il ne renvoie rien par un return et ne peut pas avoir de paramètres.
- Il existe un destructeur par défaut qui se contente de détruire les données membres de l'objet.

Exemple 5 : constructeurs et destructeurs

Le fichier Point.h

```
#ifndef POINT_H
#define POINT_H
#include <iostream>
using namespace std;

class Point
{
  public:
    Point();
    Point(double x, double y);
    void setX(double x);
    void setY(double y);
    double getX();
    double getY();
    double distance(const Point &P);
    Point milieu(const Point &P);
    void operator>>(ostream &out);
    void operator<<(istream &in);

  private:
    double x,y;
};
#endif
```

Explications

Par rapport à l'exemple précédent, nous avons rajouté 2 constructeurs :

- Le constructeur par défaut Point();
- Un autre constructeur Point(double x, double y);
- Le constructeur par défaut va initialiser l'abscisse et l'ordonnée de notre Point à 0.
- Le second constructeur va initialiser l'abscisse et l'ordonnée de notre Point respectivement à x et à y.

Ficher Point.cpp

Voici l'implémentation des 2 constructeurs :

```
Point::Point()
{
    x = 0;
    y = 0;
}

Point::Point(double x, double y)
{
    this->x = x;
    this->y = y;
}
```

Explications

- Le constructeur par défaut initialise x et y à la valeur 0.
- Pour le deuxième constructeur this->x désigne la données membre x de la classe et x désigne le paramètre du constructeur.
- Idem pour this->y.

Fichier main.cpp

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

int main()
{
    Point A, B(3.4, 5.6);
    cout << "Coordonnee du point A :" << endl;
    A >> cout;
    cout << endl << endl;
    cout << "Coordonnee du point B :" << endl;
    B >> cout;
    cout << endl << endl;
    return 0;
}
```

Explications

- Lorsqu'on déclare un objet par Point A; c'est le constructeur par défaut de la classe Point qui est appelé : l'abscisse et l'ordonnée de A sont initialisées à 0.
- Lorsqu'on déclare un objet par Point B(3.4,5.6); c'est le deuxième constructeur qui est appelé l'abscisse de B est initialisée à 3.4 et l'ordonnée à 5.6.

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
COORDONNEES DU POINT A
L'abscisse vaut : 0
L'ordonnée vaut : 0
COORDONNEES DU POINT B
L'abscisse vaut : 3.4
L'ordonnée vaut : 5.6
```

Liste d'initialisation

- Dans un constructeur, on peut intialiser des données dans la liste d'initialisation : il est d'ailleurs préférable de les initialiser à cet endroit.
- Certaines données ne peuvent être initialisées qu'à cet endroit : les références par exemple.
- Lorsqu'on étudiera l'héritage, nous reparlerons de cette liste d'initialisation.

Exemple 6 : la liste d'initialisation

Nous allons réécrire les constructeurs des fichiers Point.cpp de l'exemple 5.

Fichier Point.cpp

```
Point::Point() : x(0),y(0)
{
}

Point::Point(double x, double y) : x(x),y(y)
{
}
```

Explications

- Lors de l'écriture d'un constructeur, on peut initialiser une donnée membre à 0 en écrivant x(0). C'est ce qui est fait dans le constructeur par défaut Point() pour les données membres x et y.
- Pour le constructeur Point(int x, int y), on initialise la donnée membre x en écrivant x(x) : le premier x désigne la donnée membre de la classe x, le deuxième x désigne le paramètre du constructeur.

Les opérateurs `new` et `delete`

- Si vous avez un pointeur p déclaré ainsi : `A* p;` Pour que p pointe vers une nouvelle instance de la classe A, vous pouvez utiliser ainsi l'opérateur `new` : `p= new A();`
- Lorsque vous avez créé avec new une nouvelle instance d'un objet, vous êtes tenu de détruire cet objet avant la fin de votre programme grâce à l'opérateur `delete` : `delete p;`

`new` et les constructeurs

- Si la classe A possède plusieurs constructeurs par exemple `A();` et `A(int,int);` vous pouvez choisir le constructeur de votre choix :

```
A *x, *y;
x = new A();  // constructeur A();
y = new A(4, 8);  // constructeur A(int, int);
```

- Bien évidemment il ne faudra pas oublier de détruire les instances de la classe A créées en utilisant delete :

```
delete x;
delete y;
```

Exercices

EXERCICE 1

Écrire une classe Fraction dont le fichier d'en-tête est le suivant :

```
#ifndef FRACTION_H
#define FRACTION_H

#include<iostream>
using namespace std;

class Fraction
{
friend ostream & operator<<(ostream & out, const Fraction &f);
friend istream & operator>>(istream &in, Fraction &f);

 public:
    Fraction();
    Fraction(int i);
    Fraction(int num,int den);

    Fraction operator+(const Fraction & f);
    Fraction operator-(const Fraction & f);
    Fraction operator*(const Fraction & f);
    Fraction operator/(const Fraction & f);
 private:
    int num,den;
    int pgcd(int x, int y);
    void normalise();
};

#endif
```

Voici le rôle de chaque fonction :

- Fraction(); : le constructeur par défaut initialise la fraction à 0.
- Fraction(int); : initialise la fraction à l'entier i.

- Fraction(int num, int den); : initialise le numérateur et le dénominateur de la fraction.
- ostream & operator<<(ostream & out, const Fraction &f) : affiche à l'écran la fraction f.
- istream & operator>>(istream &in, Fraction &f) : saisit au clavier la fraction f.
- Fraction operator+(const Fraction & f); permet de faire la somme de 2 fractions.
- Fraction operator-(const Fraction & f); permet de faire la différence de 2 fractions.
- Fraction operator*(const Fraction & f); permet de faire la multiplications de 2 fractions.
- Fraction operator/(const Fraction & f); permet de faire la division de 2 fractions.
- int pgcd(int x, int y) : calcule le pgcd de 2 entiers.
- void normalise() : normalise la fraction. Le dénominateur doit être positif et la fraction irréductible.

Écrire un programme principal qui saisit au clavier 2 fractions f1 et f2 et qui affiche E=(f1+3/4-f2)/(f1*f2-5/8)+4.

Sources et contributeurs de l'article

Programmation C++ (débutant)/Les classes *Source*: http://fr.wikibooks.org/w/index.php?oldid=254565 *Contributeurs*: DavidL, Merrheim, Saamreivax, Sub, Tavernier, Trefleur, 37 modifications anonymes

Licence