TD 1 : **Bases de programmation orientée objet en C++**

DUT 1ère Année, Module d'IHM

Contact : Frédéric Koriche (koriche@cril.fr)

La conception d'interfaces fait appel à la programmation orientée objet. Les exercices suivants ont pour objectif de se familiariser avec les classes et les objets en C++.

Exercice 1. Considérons la déclaration de la classe `Point` décrite dans le listing suivant. Quels sont les attributs et les méthodes ? Quels sont les constructeurs et destructeurs ? Quel est l'opérateur de copie ? Construire un fichier en-tête (`point.hpp`) contenant cette classe.

```
#ifndef POINT_HPP
#define POINT_HPP

#include <iostream>
using namespace std;

class Point{
        float x;
        float y;

public:
        Point();
        Point(const float a, const float b);
        Point(const Point& p);
        ~Point() {}

        friend ostream &operator <<(ostream &output, const Point& p);
};

#endif
```

Exercice 2. Construire un fichier code (`point.cpp`) contenant le corps des méthodes, décrit de la manière suivante :

```
#include "point.hpp"

Point::Point()
{
}

Point::Point(const float a, const float b):
x(a),
y(b)
{
}

Point::Point(const Point& p)
{
}

ostream &operator <<(ostream &output, const Point& p)
{
output << "(" << p.x << ":" << p.y << ")" << endl;
return output;
}
```

Remplir le corps des méthodes de telle manière que :

- le point par *défaut* possède les coordonnées $(0, 0)$,
- un point puisse être copié à partir d'un autre point.

Exercice 3. Nous ajoutons au code la fonction principale qui construit deux objets de la classe `Point`.

```
int main()
{
Point p(1,0);
Point q(4,5);
}
```

Etendre la fonction principale pour afficher les points à l'écran (en utilisant l'operateur `<<`).

Exercice 4. Nous ajoutons au fichier entête `point.hpp` les méthodes suivantes permettant de modifier les coordonnées d'un point :

```
void setCoordinates(const float a, const float b);
```

Ajouter au fichier `point.cpp` la définition (code) correspondant à cette méthode, et tester cette méthode sur l'objet `p` en modifiant seulement son abscisse.

Exercice 5. Nous ajoutons au fichier entête `point.hpp` la méthode suivante qui permet de calculer la distance entre l'objet courant et l'objet cible `q`.

```
float dist(const Point& q);
```

Ajouter au fichier `point.cpp` la définition (code) correspondant à cette méthode, et tester cette méthode sur l'objet `p` en utilisant `q` en argument.

Exercice 6. Construire un fichier en-tête `rectangle.hpp` associé à la déclaration de la classe `Rectangle` décrite dans le listing suivant.

```
#ifndef RECTANGLE_HPP
#define RECTANGLE_HPP

#include <iostream>
using namespace std;

class Rectangle{
        float xl, xr, yb, yt;

public:
        Rectangle();
        Rectangle(const float x_left, const float y_bot, const float x_right, const float y_top);
        Rectangle(const Rectangle& r);
        ~Rectangle() {};

        friend ostream &operator <<(ostream &output, const Rectangle& r);
};

#endif
```

Construire un fichier code (`rectangle.cpp`) contenant le corps des méthodes, ainsi qu'une fonction principale utilisant deux objets rectangles avec les coordonnées respectives $(1, 1, 2, 2)$ et $(0, 0, 3, 3)$. Dans l'opérateur `<<`, vous pouvez afficher les rectangles comme vous le souhaitez.

Exercice 7. Définir une méthode permettant de tester si le rectangle courant est inclus dans un rectangle `r` donné en argument. La méthode doit être déclarée dans l'entête et définie dans le code.

Exercice 8. Définir un constructeur de rectangle à partir de ses points extrêmes "bas-gauche" et "haut-droite" :

```
Rectangle(const Point& bg, const Point& hd);
```

Dans ce but, vous utiliserez la classe `Point`.