Compte rendu CPP n°3

Templates

STEPHANT André-Louis, COUSSEAU Yanis

École Centrale de Nantes - Systèmes embarqués Communicants (SEC)

Table des matières

1 Introduction

Ce troisième TP a pour objectif de nous faire travailler une nouvelle notion en C++ : les templates. Pour rappel, un template est une manière de simplifier un code, permettant à une fonction d'accepter différents types de paramètres en entrée.

Dans ce TP, nous allons utiliser cette notion de templates avec des classes, ainsi que des surcharges d'opérateurs. L'objectif est de mêler plusieurs concepts différents afin de mener à bien un projet.

Dans un premier temps, nous allons créer une classe Point template. Puis, nous coderons une classe forme, dérivée de la classe précédente. Ensuite, nous utiliserons une classe rectangle, héritée de la classe forme. Enfin, nous spécialiserons nos classes, puis créerons une liste de forme.

Nous conclurons ce rendu en mettant en lumière les différentes compétences sollicitées pour ce TP.

2 Création d'une classe Point

Dans cette première partie, nous allons initialiser le premier template de notre projet : la classe Point. Pour rappel, voici le cahier des charges associé à cette classe :

Écrire une classe pour un point du plan. Outre les attributs d'abscisse et d'ordonnée, elle devra comporter :

- une méthode `translater()` qui prend en argument respectivement une paire déléments du même type que les coordonnées
- un constructeur qui prend deux paramètres
- un constructeur de recopie qui devra prendre en paramètre une référence constante à un point, c'est-à-dire un `point const &`. La référence est en réalité un pointeur mais s'utilise comme si on avait affaire à un objet passé par valeur
- des accesseurs et des mutateurs donnant accès aux valeurs des attributs et permettant de changer ces valeurs
- Surcharger l'opérateur « sous forme de fonction amie de façon à permettre l'utilisation de `cout` avec des points.

Vous expliciterez la classe `point` sous forme de diagramme de classes dans votre rapport. Vous écrirez l'ensemble du code dans `point.hpp` ainsi quun programme principal permettant de tester vos différentes méthodes. Vous présenterez vos tests sous forme de jeux d'essais dans votre compte rendu.

Figure 1: Cahier des charges partie 1

Par conséquent, on crée un fichier .hpp afin de répondre au cahier des charges :

```
1  #pragma once
2
3  // Bibliotheques
4  #include <iostream>
5
6  template <typename T>
7  class PointT;
8
9  template <typename T>
10 std::ostream &operator<<(std::ostream &, PointT<T> const &);
11
12 // Classe Point, a partir d'un template 'T'
13 template <typename T>
14 class PointT
15 {
16 protected:
17     T x;
18     T y;
19
20 public:
21     // Constructeurs
22     PointT(T x, T y) // Creation d'un point en fonction de x et de y
```

```
23      {
24          this->x = x;
25          this->y = y;
26      }
27
28      //Constructeur de copie
29      PointT(const PointT<T> &arg) {
30          this->x = arg.x;
31          this->y = arg.y;
32      }
33
34      // Accesseurs
35      T getX() const { return x; }
36      T getY() const { return y; }
37
38      // Mutateurs
39      T setX(T new_x) { x = new_x; }
40      T setY(T new_y) { y = new_y; }
41
42      // Methodes
43      // Fonction de translation de la classe PointT, prend en argument deux
        coordonnees du meme type
44      T translater(const T &dx, const T &dy)
45      {
46          this->x += dx;
47          this->y += dy;
48      }
49  };
50
51  // Surcharge operateur cout, pour afficher un point
52  template <typename T>
53  std::ostream &operator<<(std::ostream &o, const PointT<T> &arg)
54  {
55      o << "(" << "Nom :" << arg.getP() << ", " << "X :" << arg.getX() << ", " << "Y :"
         << arg.getY() << ")" << std::endl;
56      return o;
57  }
```

Listing 1: Fichier PointT.hpp

Le code devant être entièrement dans le fichier .hpp pour pouvoir compiler, nous prenons la liberté d'implémenter les fonctions directement à la déclaration.

On ajoute ensuite des jeux de test afin de valider le bon fonctionnement du programme :

Résultat attendu :

```
Point p1 (constructeur parametre) : (X :2, Y :3)

Point p2 (constructeur de copie a partir de p1) : (X :2, Y :3)

Abscisse de p1 : 2
Ordonnee de p1 : 3
p1 apres modification via les mutateurs : (X :10, Y :20)
```

```
8
9  Translation de p1 de ( -3 , +5 )... p1 apres translation : (X :7, Y :25)
10
11 p3 : (X :0, Y :0)
12
13 p3 apres translation (4.5, -2.5) : (X :4.5, Y :-2.5)
```

Résultat obtenu :

```
1  Point p1 (constructeur parametre) : (X :2, Y :3)
2
3  Point p2 (constructeur de copie a partir de p1) : (X :2, Y :3)
4
5  Abscisse de p1 : 2
6  Ordonnee de p1 : 3
7  p1 apres modification via les mutateurs : (X :10, Y :20)
8
9  Translation de p1 de ( -3 , +5 )... p1 apres translation : (X :7, Y :25)
10
11 p3 : (X :0, Y :0)
12
13 p3 apres translation (4.5, -2.5) : (X :4.5, Y :-2.5)
```

On peut donc valider que la classe fonctionne comme attendu.

3 Formes géométriques abstraites

Dans cette seconde partie, nous allons créer une classe afin de manipuler une forme, constituée de points. L'objectif est de réaliser une forme centrée sur un point particulier (créé dans la partie précédente). Pour rappel, voici les instructions données dans le sujet de TP associées à cette classe :

1. Écrire une classe `forme` qui aura ce point comme attribut (centre de la forme) avec le constructeur approprié.
2. Surcharger l'opérateur « de façon à permettre l'utilisation de `cout` avec des formes, toujours sous forme d'une fonction amie
3. Déclarer deux méthodes abstraites `perimetre()` et `surface()`. Attention les types de retour de ces deux fonctions devront être génériques mais pas forcément le même que celui de l'abscisse et de l'ordonnée des points (pour éviter les pertes de précision quand on utilise des entiers) .

Figure 2: Cahier des charges partie 2

Par conséquent, on crée un fichier .hpp afin de répondre au cahier des charges :

```hpp
#pragma once

#include "PointT.hpp"
#include <iostream>
#include <array>

// C++ 14: types par défaut dans les templates
template <typename TPoint, typename TPerimetre = float, typename TSurface = float>
class Forme;

template <typename T>
std::ostream &operator<<(std::ostream &, Forme<T> const &);

// Classe forme
template <typename TPoint, typename TPerimetre, typename TSurface>
class Forme
{
private:
    PointT<TPoint> centre; // Centre de la forme

public:
    // Constructeurs
    Forme(PointT<TPoint> const & p); // Création d'une forme en fonction d'un point P

    // Accesseurs
    PointT<TPoint> getPoint() {return centre;}

    // Mutateurs
    void setCentre(PointT<TPoint> value);

    // Méthodes
```

```
32      virtual TPerimetre perimetre() = 0; // Périmètre de la forme, fonction virtuelle
33
34      virtual TSurface surface() = 0; // Aire de la surface, fonction virtuelle
35
36      virtual std::array<PointT<TPoint>, 4> boundingBox() = 0; // Liste des points,
        fonction virtuelle
37
38      friend std::ostream &operator<< <TPoint>(std::ostream &, Forme const &);
39  };
40
41  // Création d'une forme en fonction d'un point P
42  template <typename TPoint, typename TPerimetre, typename TSurface>
43  Forme<TPoint, TPerimetre, TSurface>::Forme(PointT<TPoint>const & p) : centre(p)
44  {
45  }
46
47  // Surcharge de l'opérateur cout, pour afficher un point
48  template <typename TPoint, typename TPerimetre, typename TSurface>
49  std::ostream &operator<<(std::ostream &o, Forme<TPoint, TPerimetre, TSurface> const &
        R)
50  {
51      o << "(" << R.getPoint().getX() << "," << R.getPoint().getY() << ")" << std::endl
        ;
52      return o;
53  }
```

Listing 2: Fichier Forme.hpp

Cette classe étant abstraite, elle ne peut pas être instanciée et donc testée directement.

4 Formes géométriques concrètes

On veut ici réaliser des formes géométriques concrètes. C'est-à-dire une forme constituée de points, avec une longueur et une hauteur donnée. On va créer ici deux classes : rectangle et carré.

On crée donc deux fichiers .hpp :

```hpp
#pragma once

#include "PointT.hpp"
#include "Forme.hpp"
#include <iostream>
#include <array>

template <typename TPoint, typename TPerimetre = float, typename TSurface = float>
class Rectangle;

template <typename T>
std::ostream &operator<<(std::ostream &, Rectangle<T> &);

// Classe Rectangle, dérivée de la classe forme
// C++ 14: types par défaut dans les templates
template <typename TPoint, typename TPerimetre, typename TSurface>
class Rectangle : public Forme<TPoint, TPerimetre, TSurface>
{
private:
    TPoint h; // Hauteur du rectangle
    TPoint l; // Largeur du rectangle

public:
    // Constructeurs
    Rectangle() : Forme<TPoint>(), h(TPoint()), l(TPoint()) {};
    Rectangle(PointT<TPoint> const &p, TPoint h, TPoint l) : Forme<TPoint>(p), h(h), l(l) {}

    // Accesseurs
    TPoint getH() const { return this->h; }
    TPoint getL() const { return this->l; }

    // Mutateurs
    void setH(TPoint h) { this->h = h; }
    void setL(TPoint l) { this->l = l; }

    // Méthodes
    std::array<PointT<TPoint>, 4> boundingBox() override
    {
        TPoint x0 = this->getPoint().getX();
        TPoint y0 = this->getPoint().getY();
        TPoint x1 = x0 + l;
        TPoint y1 = y0 + h;

        return {PointT<TPoint>(x0, y0),
                PointT<TPoint>(x1, y0),
                PointT<TPoint>(x1, y1),
                PointT<TPoint>(x0, y1)};
    }
```

```

    TPerimetre perimetre() override {
        return (2*h)+(2*l);
    }

    TSurface surface() override {
        return h*l;
    }
};

template <typename T>
std::ostream &operator<<(std::ostream &o, Rectangle<T> &R)
{
    o << "(" << R.getPoint().getX() << "," << R.getPoint().getY() << "), " << R.getH
    () << "," << R.getL() << std::endl;
    return o;
}
```

Listing 3: Fichier Rectangle.hpp

```
#pragma once

#include "Rectangle.hpp"
#include <iostream>

template <typename TPoint, typename TPerimetre = float, typename TSurface = float>
class Carre;

template <typename T>
std::ostream &operator<<(std::ostream &, Carre<T> const &);

// Classe Carré héritée de Rectangle, on force la largeur et la hauteur à la même
    valeur
template <typename TPoint, typename TPerimetre, typename TSurface>
class Carre : public Rectangle<TPoint, TPerimetre, TSurface>
{
public:
    // Constructeurs
    Carre() : Rectangle<TPoint>() {}
    Carre(PointT<TPoint> const &p, TPoint cote) : Rectangle<TPoint>(p, cote, cote) {}
    virtual ~Carre() = default;

    // Accesseur
    TPoint getCote() const { return this->getH(); }

    // Mutateur
    void setCote(TPoint cote)
    {
        Rectangle<TPoint>::setH(cote);
        Rectangle<TPoint>::setL(cote);
    }

    // Redéfinition des mutateurs de Rectangle pour conserver l'invariant du carré
    TPoint setH(TPoint h)
    {
        setCote(h);
        return h;
```

```
37      }
38      TPoint setL(TPoint l)
39      {
40          setCote(l);
41          return l;
42      }
43  };
44
45  // Surcharge de l'opérateur d'affichage pour la classe Carre
46  template <typename T>
47  std::ostream &operator<<(std::ostream &o, Carre<T> const &c)
48  {
49      o << "Carré: origine=(" << c.getPoint().getX() << "," << c.getPoint().getY()
50        << "), cote=" << c.getCote();
51      return o;
52  }
```

Listing 4: Fichier Carre.hpp

On ajoute ensuite des jeux de test afin de valider le bon fonctionnement du programme :

Résultat attendu :

Rectangle r1 : (0,0), 10,5

Périmètre : 30 Surface : 50

Carré c1 : Carré: origine=(3,4), cote=6 Périmètre : 24 Surface : 36

Résultat obtenu :

Rectangle r1 : (0,0), 10,5

Périmètre : 30 Surface : 50

Carré c1 : Carré: origine=(3,4), cote=6 Périmètre : 24 Surface : 36

On peut donc valider que ces classes fonctionnent comme attendu.

5 Liste de Formes

On veut créer une classe stockant une liste de formes et permettant d'effectuer quelques opérations simples.

On crée donc un fichier .hpp :

```
#pragma once

#include <vector>
#include <limits>
#include "Forme.hpp"
#include "Rectangle.hpp"

template <typename TPoint, typename TPerimetre = float, typename TSurface = float>
class ListeFormes
{
private:
    std::vector<Forme<TPoint, TPerimetre, TSurface>*> formes;

public:

    // Ajouter une forme dans la liste
    void ajouter(Forme<TPoint, TPerimetre, TSurface>* f) {
        formes.push_back(f);
    }

    std::size_t taille() const {
        return formes.size();
    }

    Forme<TPoint, TPerimetre, TSurface>* operator[](std::size_t i) {
        return formes[i];
    }

    // Surface totale de toutes les formes
    TSurface surfaceTotale() const {
        TSurface total{};
        for (auto f : formes)
            total += f->surface();
        return total;
    }

    // Boîte englobante exprimée sous forme de rectangle
    Rectangle<TPoint, TPerimetre, TSurface> boundingBox() const {

        if (formes.empty()) {
            return Rectangle<TPoint, TPerimetre, TSurface>(
                PointT<TPoint>(0,0), 0, 0
            );
        }

        TPoint xmin = std::numeric_limits<TPoint>::max();
        TPoint ymin = std::numeric_limits<TPoint>::max();
        TPoint xmax = std::numeric_limits<TPoint>::lowest();
        TPoint ymax = std::numeric_limits<TPoint>::lowest();
```

```
50
51          // On visite la boundingBox (4 sommets) de chaque forme
52          for (auto f : formes) {
53              auto bb = f->boundingBox();   // array<4 points>
54
55              for (const auto& pt : bb) {
56                  xmin = std::min(xmin, pt.getX());
57                  ymin = std::min(ymin, pt.getY());
58                  xmax = std::max(xmax, pt.getX());
59                  ymax = std::max(ymax, pt.getY());
60              }
61          }
62
63          // Construire le rectangle final
64          return Rectangle<TPoint, TPerimetre, TSurface>(
65              PointT<TPoint>(xmin, ymin),     // Coin bas gauche
66              ymax - ymin,                    // Hauteur
67              xmax - xmin                     // Largeur
68          );
69      }
70  };
```

Listing 5: Fichier ListeFormes.hpp

On ajoute ensuite des jeux de test afin de valider le bon fonctionnement du programme :

Résultat attendu :

Boîte englobante : (0,0) (20,0) (20,11) (0,11)

Résultat obtenu :

Boîte englobante : (0,0) (20,0) (20,11) (0,11)

On peut donc valider que cette classe fonctionne comme attendu.

6 Conclusion

Ce TP nous a permis de découvrir et d'approfondir la notion de template en C++. Nous avons pu mettre en pratique cette notion à travers la création de plusieurs classes, ainsi que l'utilisation de l'héritage et des surcharges d'opérateurs.

Nous avons également renforcé notre compréhension des concepts liés aux classes, ainsi que du langage C++ d'une manière générale. Ces notions nous seront importantes pour l'avenir, notamment dans le cadre du projet de fin de module à venir.