SMP

TP 7

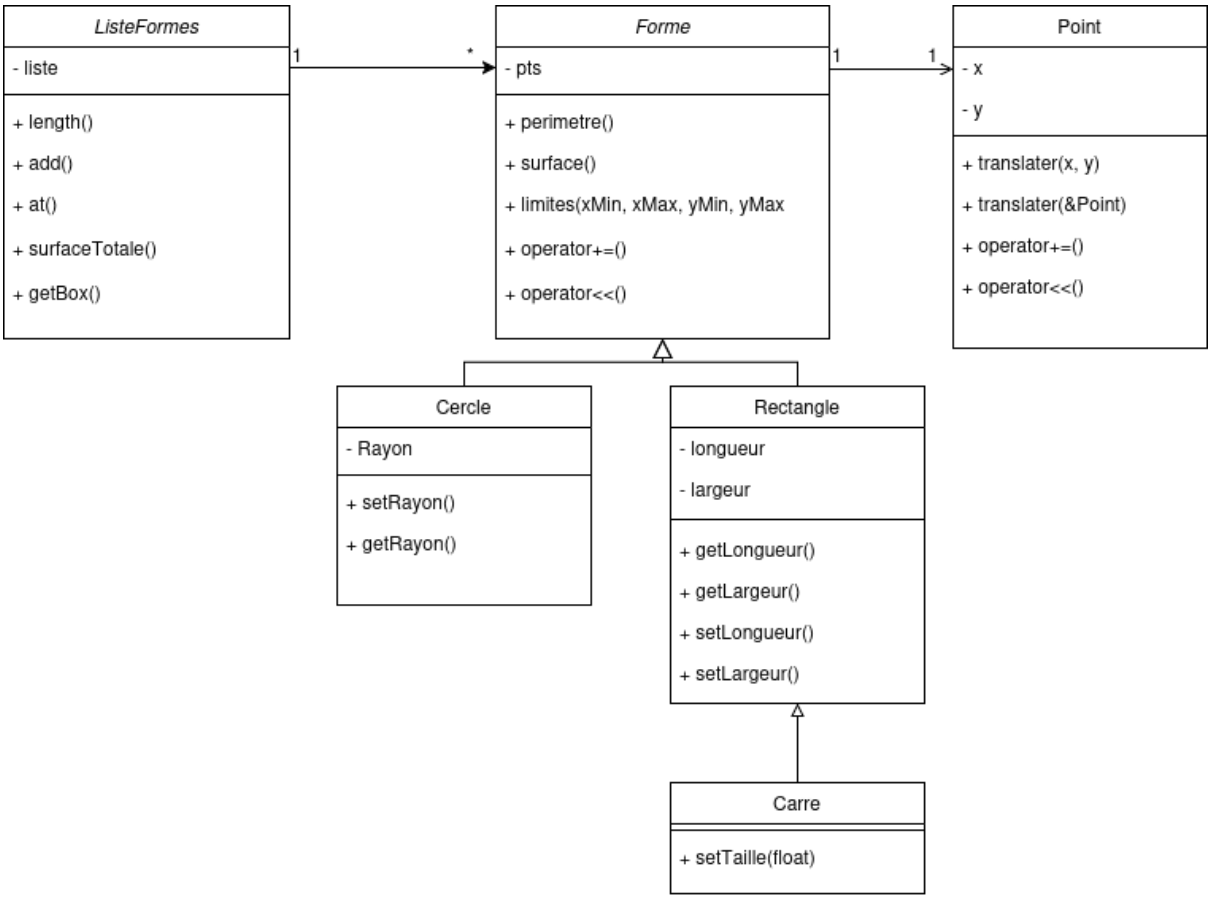
Jack HASCOET

Nathan ANDRE

2 février 2025

Système Embarqué Communicant 2024/2025

Diagramme de classe



Classe Point

La classe Point à deux attributs, X et Y. Ils désignent la position du point dans un repère orthonormé. Ces attributs sont des flottants et chacun d'eux ont des getter et setter et d'autres fonctions:

- Translater - Additionne deux points entre eux
- Surcharge += - Additionne deux points entre eux
- Surcharge << - Simplifie l'affichage d'un Point

src > Point.cpp > ...

```cpp
#include "Point.hpp"

/**
 * @brief Constructeur par défaut
 * Initialise un point à l'origine
 */
Point::Point()
{
    this->x = 0;
    this->y = 0;
}

/**
 * @brief Surcharge de constructeur
 * Initialise un point à ses paramètres
 *
 * @param x Abscisse
 * @param y Ordonnée
 */
Point::Point(float x, float y)
{
    this->x = x;
    this->y = y;
};

/**
 * @brief Surchage de constructeur par recopie
 *
 * @param point Point à recopier
 */
Point::Point(const Point &point)
{
    this->x = point.x;
    this->y = point.y;
}

/**
 * @brief Récupère la valeur de d'abscisse du point
 *
 * @return float - Abscisse
 */
float Point::getX() const
{
    return this->x;
}
```

```cpp
 * @return float - Ordonnée
 */
float Point::getY() const
{
    return this->y;
}

/**
 * @brief Défini la valeur d'abscisse du point
 *
 * @param x Le nouvel abscisse
 */
void Point::setX(float x)
{
    this->x = x;
}

/**
 * @brief Défini la valeur d'ordonnée du point
 *
 * @param y Le nouvel ordonnée
 */
void Point::setY(float y)
{
    this->y = y;
}

/**
 * @brief Destructeur du point
 */
Point::~Point()
{
}

/**
 * @brief Translate un point en fonction de ses paramètres
 *
 * @param x Translation en abscisse
 * @param y Translation en y
 */
void Point::translater(float x, float y)
{
    this->x += x;
    this->y += y;
}
```

```cpp
/**
 * @brief Translate un point par rapport à un autre
 *
 * @param pts Référence du point avec lequel faire la translation
 */
void Point::translater(Point &pts)
{
    this->x += pts.getX();
    this->y += pts.getY();
}

/**
 * @brief Surcharge de l'opérateur +=
 *
 * @param pts Point avec lequel faire l'addition
 * @return Point& - La référence à this
 */
Point &Point::operator+=(const Point &pts)
{
    this->x += pts.x;
    this->y += pts.y;
    return *this;
}

/**
 * @brief Surcharge de l'opérateur <<
 *
 * @param os outputstream
 * @param pts Référence du point à afficher
 * @return ostream& l'output textuelle
 */
ostream &operator<<(ostream &os, const Point &pts)
{
    os << "Point(" << pts.x << ", " << pts.y << ")";
    return os;
}
```

Test

Constructeur: Nous testons le constructeur par défaut, la surcharge avec les coordonnées et par recopie.

```
void test_pointConstructeur()
{
    Point pts_1 = Point();
    Point pts_2 = Point(1, 2);
    Point pts_3 = pts_2;

    // Test du constructeur par défaut
    assert(pts_1.getX() == 0);
    assert(pts_1.getY() == 0);

    // Test du constructeur surcharger (float x, float y)
    assert(pts_2.getX() == 1);
    assert(pts_2.getY() == 2);

    // Test du constructeur par recopie
    assert(pts_3.getX() == 1);
    assert(pts_3.getY() == 2);

    cout << "test_pointConstructeur()\t : \tOK" << endl;
}
```

Translater: Nous testons la translations par coordonnées et par Point

```
void test_pointTranslater()
{
    Point pts_1 = Point();       // Point initialisé en (0, 0)
    Point pts_2 = Point(5, 5);   // Point initialisé en (5, 5)
    Point pts_3 = Point(2, 3);   // Point initialisé en (2, 3)
    Point pts_4 = Point(5, 6);   // Point initialisé en (5, 6)
    Point pts_5 = Point(1, 3);   // Point initialisé en (1, 3)
    Point pts_6 = Point(-3, -1); // Point initialisé en (-3, -3)

    pts_1.translater(3, 4);   // Translate en (3,4)
    pts_2.translater(-6, -6); // Translate en (-1, -1)
    pts_3.translater(-2, -3); // Translate en (0, 0)
    pts_4.translater(pts_5);  // Translate en (6, 9)
    pts_5.translater(pts_6);  // Translate en (-2, 2)

    // Test point_1
    assert(pts_1.getX() == 3);
    assert(pts_1.getY() == 4);

    // Test point_2
    assert(pts_2.getX() == -1);
    assert(pts_2.getY() == -1);

    // Test point_3
    assert(pts_3.getX() == 0);
    assert(pts_3.getY() == 0);

    // Test point_4
    assert(pts_4.getX() == 6);
    assert(pts_4.getY() == 9);

    // Test point_5
    assert(pts_5.getX() == -2);
    assert(pts_5.getY() == 2);

    cout << "test_pointTranslater()\t\t : \tOK" << endl;
}
```

Surcharge +=

```cpp
void test_pointOperateurPlusEgale()
{
    Point pts_1 = Point();       // Point initialisé en (0, 0)
    Point pts_2 = Point(5, 5);   // Point initialisé en (5, 5)
    Point pts_3 = Point(2, 3);   // Point initialisé en (2, 3)
    Point pts_4 = Point(1, 3);   // Point initialisé en (1, 3)
    Point pts_5 = Point(-3, -1); // Point initialisé en (-3, -3)

    pts_1 += pts_2; // (0, 0) + (5, 5) = (5, 5)
    pts_2 += pts_3; // (5, 5) + (2, 3) = (7, 8)
    pts_4 += pts_5; // (1, 3) + (-3, -1) = (-2, 2)

    assert(pts_1.getX() == 5);
    assert(pts_1.getY() == 5);

    assert(pts_2.getX() == 7);
    assert(pts_2.getY() == 8);

    assert(pts_4.getX() == -2);
    assert(pts_4.getY() == 2);

    cout << "test_pointOperateurPlusEgale()\t : \tOK" << endl;
}
```

Surcharge <<

```cpp
void test_pointOperateurCout()
{
    Point pts_1 = Point();       // Point initialisé en (0, 0)
    Point pts_2 = Point(5, 5);   // Point initialisé en (5, 5)
    Point pts_3 = Point(-3, -1); // Point initialisé en (-3, -3)

    cout << "test_pointOperateurCout - Affichage des points" << endl;
    cout << pts_1 << endl;
    cout << pts_2 << endl;
    cout << pts_3 << endl;
}
```

Résultat

```
-- Tests de la classe Point --
test_pointConstructeur()          :      OK
test_pointTranslater()            :      OK
test_pointOperateurPlusEgale()    :      OK
test_pointOperateurCout - Affichage des points
Point(0, 0)
Point(5, 5)
Point(-3, -1)
```

Classe Forme

Classe définissant une Forme. Elle a pour attribut une instance de la classe Point qui représente son Point d'ancrage.

src > Forme.cpp > ...

```cpp
#include "Forme.hpp"

/**
 * @brief Constructeur de la classe Forme
 * Initialise le point en (0, 0)
 *
 */
Forme::Forme(void)
{
    this->pts = new Point;
    this->pts->setX(0);
    this->pts->setY(0);
}

/**
 * @brief Surcharge du constructeur de la classe Forme
 *
 * @param p Point d'ancrage de la forme
 */
Forme::Forme(Point *p)
{
    this->pts = p;
}

/**
 * @brief Surcharge du constructeur de la classe Forme
 *
 * @param x valeur X du point d'ancrage
 * @param y valeur Y du point d'ancrage
 */
Forme::Forme(float x, float y)
{
    this->pts = new Point;
    this->pts->setX(x);
    this->pts->setY(y);
}
```

```cpp
/**
 * @brief Surcharge du constructeur de la classe Forme par recopie
 *
 * @param p
 */
Forme::Forme(const Forme &p)
{
    // Creation d'un nouveau point car si on supprime la forme copie
    // on veut pouvoir garder le point copie
    // Donc on n'enregistre pas l'addr du point de la forme copie
    this->pts = new Point;
    this->pts->setX(p.pts->getX());
    this->pts->setY(p.pts->getY());
}

/**
 * @brief Destructeur de l'objet
 *
 */
Forme::~Forme()
{
    delete this->pts;
}

/**
 * @brief Surcharge de l'opérateur +=
 *
 * @param A Référence du point
 * @return Forme&
 */
Forme &Forme::operator+=(const Point &A)
{
    this->pts->setX(A.getX());
    this->pts->setY(A.getY());
    return *this;
}

/**
 * @brief Récupère le pointeur de la Forme
 *
 * @return Point*
 */
Point *Forme::getPoint() const
{
    return this->pts;
}
```

Classe Cercle

Classe définissant la forme Cercle, héritant de la classe Forme. Elle a pour attribut un nombre flottant “rayon” représentant le rayon du cercle.
La classe possède des surcharges des méthodes perimetre, surface et limites déclaré abstrait dans la classe mère Forme.

Cercle.hpp :

```
#ifndef CERCLE_H
#define CERCLE_H_

#include "Forme.hpp"
#include <cmath>

class Cercle : public Forme
{
private:
   float rayon;

public:
   Cercle(void);
   Cercle( float rayon);
   Cercle(float rayon, float x, float y);
   Cercle(const Cercle &c);

   void setRayon(float rayon);
   float getRayon(void);

   float perimetre(void);
   float surface(void);
   void limites(float* xMin, float* xMax, float* yMin, float* yMax);

   friend ostream &operator<<(ostream &os, const Cercle &forme);
};

#endif // CERCLE_H_
```

Constructeurs

```cpp
Cercle::Cercle(void) : Forme()
{
    this->rayon = 0;
}

Cercle::Cercle(float rayon) : Forme()
{
    if (rayon <= 0)
    {
        throw invalid_argument("Le rayon doit être supérieur à 0");
    }
    this->rayon = rayon;
}

Cercle::Cercle(float rayon, float x, float y) : Forme(x, y)
{
    if (rayon <= 0)
    {
        throw invalid_argument("Le rayon doit être supérieur à 0");
    }
    this->rayon = rayon;
}

Cercle::Cercle(const Cercle &c) : Forme(c.getPoint()->getX(), c.getPoint()->getY())
{
    this->rayon = c.rayon;
}
```

Getter et setter de rayon :

```cpp
void Cercle::setRayon(float rayon)
{
    if (rayon <= 0)
    {
        throw invalid_argument("Le rayon doit être supérieur à 0");
    }
    this->rayon = rayon;
}
```

```cpp
float Cercle::getRayon(void)
{
    return this->rayon;
}
```

Surcharge perimetre, surface et limites :

```cpp
float Cercle::perimetre(void)
{

    if (this->rayon <= 0)
    {
        throw runtime_error("Le rayon doit être supérieur à 0");
    }

    return round(M_PI * (this->rayon * 2) * 100) / 100;
}
float Cercle::surface(void)
{
    if (this->rayon <= 0)
    {
        throw runtime_error("Le rayon doit être supérieur à 0");
    }
    return round((M_PI * this->rayon * this->rayon) * 100) / 100;
}

void Cercle::limites(float* xMin, float* xMax, float* yMin, float* yMax) {
    *xMin = this->getPoint()->getX() - this->rayon;
    *xMax = this->getPoint()->getX() + this->rayon;
    *yMin = this->getPoint()->getY() - this->rayon;
    *yMax = this->getPoint()->getY() + this->rayon;
}
```

Surcharge pour “cout << …” :

```cpp
ostream &operator<<(ostream &os, const Cercle &cercle)
{
    os << "Cercle:\n\t- Centré au " << *cercle.getPoint() << "\n\t- Rayon " << cercle.rayon;
    return os;
}
```

Test

Constructeur - Nous testons les différents constructeurs. Nous vérifions que la création de cercles se fait correctement, et que les valeurs enregistrées correspondent bien à celle renseignée dans le constructeur le cas échéant.

```
void test_cercleConstructeur()
{
    Cercle cercle_1 = Cercle();        // Initialisation cercle par défaut
    Cercle cercle_3 = Cercle(1);       // Initialisation cercle de rayon 1
    Cercle cercle_4 = Cercle(2, 1, 1); // Initialisation cercle de rayon 2 au point (1, 1)
    Cercle cercle_5 = cercle_4;        // Copie de cercle 4
    try
    {
        Cercle cercle_2 = Cercle(0);
    }
    catch (const exception &e)
    {
        cerr << "Exception caught: " << e.what() << endl;
    }

    try
    {
        Cercle cercle_2 = Cercle(-1);
    }
    catch (const exception &e)
    {
        cerr << "Exception caught: " << e.what() << endl;
    }

    // Initialisation cercle par défaut
    assert(cercle_1.getRayon() == 0);
    assert(cercle_1.getPoint()->getX() == 0);
    assert(cercle_1.getPoint()->getY() == 0);

    // Initialisation cercle de rayon 1
    assert(cercle_3.getRayon() == 1);
    assert(cercle_3.getPoint()->getX() == 0);
    assert(cercle_3.getPoint()->getY() == 0);

    // Initialisation cercle de rayon 2 au point (1, 1)
    assert(cercle_4.getRayon() == 2);
    assert(cercle_4.getPoint()->getX() == 1);
    assert(cercle_4.getPoint()->getY() == 1);

    // Copie de cercle_4
    assert(cercle_5.getRayon() == 2);
    assert(cercle_5.getPoint()->getX() == 1);
    assert(cercle_5.getPoint()->getY() == 1);

    cout << "test_cercleConstructeur():\t\t:\tOK" << endl;
}
```

Perimetre Nous testons la récupération de périmètre sur un cercle où le rayon vaut 0 Nous testons les cas normaux ensuite.

```
void test_cerclePerimetre()
{
    Cercle cercle_1 = Cercle();        // Initialisation cercle par défaut
    Cercle cercle_2 = Cercle(1);       // Initialisation cercle de rayon 1
    Cercle cercle_3 = Cercle(2, 1, 1); // Initialisation cercle de rayon 2 au point (1, 1)

    try
    {
        cercle_1.perimetre();
    }
    catch (const exception &e)
    {
        cerr << "Exception caught: " << e.what() << endl;
    }

    assert(cercle_2.perimetre() == (float)6.28);
    assert(cercle_3.perimetre() == (float)12.57);
    cout << "test_cerclePerimetre():\t\t:\tOK" << endl;
}
```

Surface Nous testons la récupération de la surface sur un cercle où le rayon vaut 0 Nous testons les cas normaux ensuite.

```
void test_cercleSurface()
{
    Cercle cercle_1 = Cercle();        // Initialisation cercle par défaut
    Cercle cercle_2 = Cercle(1);       // Initialisation cercle de rayon 1
    Cercle cercle_3 = Cercle(4, 1, 1); // Initialisation cercle de rayon 2 au point (1, 1)

    try
    {
        cercle_1.perimetre();
    }
    catch (const exception &e)
    {
        cerr << "Exception caught: " << e.what() << endl;
    }

    assert(cercle_2.surface() == (float)3.14);
    assert(cercle_3.surface() == (float)50.27);
    cout << "test_cercleSurface():\t\t:\tOK" << endl;
}
```

Operateur<<

```
void test_cercleOperateurCout()
{
    Cercle cercle_1 = Cercle(1);       // Initialisation cercle de rayon 1
    Cercle cercle_2 = Cercle(4, 1, 1); // Initialisation cercle de rayon 2 au point (1, 1)

    cout << "Surcharge opérateur <<" << endl;
    cout << cercle_1 << endl;
    cout << cercle_2 << endl;
}
```

Resultat des tests

```
-- Tests de la classe Cercle --
test_carreConstructeur():       :       OK
Exception caught: La largeur doit être supérieur à 0
test_carreSetTaille():          :       OK
Surcharge opérateur <<
Carre:
        - Centré au Point(0, 0)
        - Taille 0
Carre:
        - Centré au Point(0, 0)
        - Taille 3
Carre:
        - Centré au Point(5, 4)
        - Taille 7
```

Classe Rectangle

Classe définissant un Rectangle, elle hérite de Forme. Elle a deux attributs, longueur et largeur.

src > Rectangle.cpp > ...

```cpp
#include "Rectangle.hpp"
#include <stdexcept>

/**
 * @brief Constructeur de la classe Rectangle
 * Initialise le rectangle en longueur 0 et largeur 0
 */
Rectangle::Rectangle(void) : Forme()
{
    this->largeur = 0;
    this->longueur = 0;
}

/**
 * @brief Surcharge du contructeur de la classe Rectangle
 *
 * @param longeur longueur du rectangle
 * @param largeur largeur du rectangle
 */
Rectangle::Rectangle(float longueur, float largeur) : Forme()
{
    if (largeur <= 0 || longueur <= 0)
    {
        throw invalid_argument("La longueur et la largeur doivent être supérieur à 0");
    }
    this->longueur = longueur;
    this->largeur = largeur;
}

/**
 * @brief Surcharge du constructeur de la classe Rectangle
 *
 * @param longeur Longueur du rectangle
 * @param largeur Largeur du rectangle
 * @param ptsX Coordonnée X du rectangle
 * @param ptsY Coordonnée Y du rectangle
 */
Rectangle::Rectangle(float longueur, float largeur, float ptsX, float ptsY) : Forme(ptsX
{
    if (largeur <= 0 || longueur <= 0)
    {
        throw invalid_argument("La longueur et la largeur doivent être supérieur à 0");
    }
    this->longueur = longueur;
    this->largeur = largeur;
}
```

src > Rectangle.cpp > ...

```cpp
/**
 * @brief Surcharge du constructeur de la classe Rectangle par recopie
 *
 * @param rect Rectangle à recopier
 */
Rectangle::Rectangle(const Rectangle &rect) : Forme(rect.getPoint()->getX(), rect.getPoi
{
    this->largeur = rect.largeur;
    this->longueur = rect.longueur;
}

/**
 * @brief Défini la longueur du rectangle
 *
 * @param longeur Nouvelle longueur
 */
void Rectangle::setLargeur(float largeur)
{
    if (largeur > 0)
    {
        this->largeur = largeur;
    }
    else
    {
        throw invalid_argument("La largeur doit être supérieur à 0");
    }
}

/**
 * @brief Défini la largeur du Rectangle
 *
 * @param largeur Nouvelle largeur
 */
void Rectangle::setLongeur(float longueur)
{
    if (longueur > 0)
    {
        this->longueur = longueur;
    }
    else
    {
        throw invalid_argument("La longueur doit être supérieur à 0");
    }
}
```

src > Rectangle.cpp > ...

```cpp
/**
 * @brief Récupère la largeur du Rectangle
 *
 * @return float
 */
float Rectangle::getLargeur() const
{
    return this->largeur;
}

/**
 * @brief Récupère la longueur du Rectangle
 *
 * @return float
 */
float Rectangle::getLongeur() const
{
    return this->longueur;
}

/**
 * @brief Calcule le périmètre du rectangle
 *
 * @return float
 */
float Rectangle::perimetre()
{
    if (this->longueur <= 0)
    {
        throw runtime_error("La longueur doit être supérieur à 0");
    }
    if (this->largeur <= 0)
    {
        throw runtime_error("La largeur doit être supérieur à 0");
    }
    return (this->longueur + this->largeur) * 2;
}
```

src > Rectangle.cpp > ...

```cpp
/**
 * @brief Calcule la surface du rectangle
 *
 * @return float
 */
float Rectangle::surface()
{
    if (this->longueur <= 0)
    {
        throw runtime_error("La longueur doit être supérieur à 0");
    }
    if (this->largeur <= 0)
    {
        throw runtime_error("La largeur doit être supérieur à 0");
    }

    return this->largeur * this->longueur;
}

/**
 * @brief Calcule les limites du rectangle
 *
 * @param xMin valeur minimal de X
 * @param xMax valeur maximale de X
 * @param yMin valeur minimale de Y
 * @param yMax valeur maximale de Y
 */
void Rectangle::limites(float *xMin, float *xMax, float *yMin, float *yMax)
{
    *xMin = this->getPoint()->getX() - this->longueur;
    *xMax = this->getPoint()->getX() + this->longueur;
    *yMin = this->getPoint()->getY() - this->largeur;
    *yMax = this->getPoint()->getY() + this->largeur;
}

/**
 * @brief Surcharge de l'opérateur <<
 *
 * @param os output stream
 * @param forme Forme à afficher
 * @return ostream&
 */
ostream &operator<<(ostream &os, const Rectangle &rect)
{
    os << "Rectangle:\n\t- Centré au " << *rect.getPoint() << "\n\t- Longueur " << rect.longueur << "\n\t- Largeur " << rect.largeur;
    return os;
}
```

Test

Constructeur - Nous testons les constructeurs et les surcharges. Nous vérifions que nous ne pouvons pas instancier un rectangle ou la largeur et/ou la longueur est <= 0 (sauf constructeur par défaut)

```
273  void test_rectangleConstructeur()
274  {
275      Rectangle rect_1 = Rectangle();           // Constructeur par défaut
276      Rectangle rect_2 = Rectangle(3, 4);       // Définition de longueur, largeur
277      Rectangle rect_3 = Rectangle(3, 4, 2, 3); // Définition de longueur, largeur, X, Y
278      Rectangle rect_7 = rect_3;                // Par recopie
279
280      try
281      {
282          Rectangle rect_4 = Rectangle(0, 0, 2, 3); // Largeur et longueur à 0
283          Rectangle rect_5 = Rectangle(0, 3, 2, 3); // Longueur à 0
284          Rectangle rect_6 = Rectangle(3, 0, 2, 3); // Largeur à 0
285      }
286      catch (const exception &e)
287      {
288          cout << "Exception caught: " << e.what() << endl;
289      }
290
291      assert(rect_1.getLargeur() == 0);
292      assert(rect_1.getLongeur() == 0);
293      assert(rect_1.getPoint()->getX() == 0);
294      assert(rect_1.getPoint()->getY() == 0);
295
296      assert(rect_2.getLongeur() == 3);
297      assert(rect_2.getLargeur() == 4);
298      assert(rect_2.getPoint()->getX() == 0);
299      assert(rect_2.getPoint()->getY() == 0);
300
301      assert(rect_3.getLongeur() == 3);
302      assert(rect_3.getLargeur() == 4);
303      assert(rect_3.getPoint()->getX() == 2);
304      assert(rect_3.getPoint()->getY() == 3);
305
306      assert(rect_7.getLongeur() == 3);
307      assert(rect_7.getLargeur() == 4);
308      assert(rect_7.getPoint()->getX() == 2);
309      assert(rect_7.getPoint()->getY() == 3);
310      cout << "test_rectangleConstructeur():\t:\tOK" << endl;
311  }
```

SetLongueur Nous testons des valeurs normales et les valeurs inférieur ou égale à 0

```
313  void test_rectangleSetLongueur()
314  {
315      Rectangle rect_1 = Rectangle();
316      Rectangle rect_2 = Rectangle();
317      Rectangle rect_3 = Rectangle(3, 3);
318
319      rect_1.setLongeur(10);
320
321      try
322      {
323          rect_2.setLongeur(0);
324          rect_3.setLongeur(-1);
325      }
326      catch (const exception &e)
327      {
328          cerr << "Exception caught: " << e.what() << endl;
329      }
330
331      assert(rect_1.getLongeur() == 10); // Longueur 0 vers 10 => 10
332      assert(rect_2.getLongeur() == 0);  // Longueur 0 vers 0 => 0
333      assert(rect_3.getLongeur() == 3);  // Longueur 3 vers -1 => 3
334
335      cout << "test_rectangleSetLongueur():\t:\tOK" << endl;
336  }
```

SetLargeur Nous testons des valeurs normales et les valeurs inférieur ou égale à 0

```cpp
void test_rectangleSetLargeur()
{
    Rectangle rect_1 = Rectangle();
    Rectangle rect_2 = Rectangle();
    Rectangle rect_3 = Rectangle(3, 3);

    rect_1.setLargeur(10);

    try
    {
        rect_2.setLargeur(0);
        rect_3.setLargeur(-1);
    }
    catch (const exception &e)
    {
        cerr << "Exception caught: " << e.what() << endl;
    }

    assert(rect_1.getLargeur() == 10); // Largeur 0 vers 10 => 10
    assert(rect_2.getLargeur() == 0);  // Largeur 0 vers 0 => 0
    assert(rect_3.getLargeur() == 3);  // Largeur 3 vers -1 => 3

    cout << "test_rectangleSetLongueur():\t:\tOK" << endl;
}
```

Périmètre Nous testons la récupération de périmètre sur un rectangle où la largeur et/ou la longueur vaut 0
Nous testons les cas normaux ensuite.

```cpp
void test_rectanglePerimetre()
{
    Rectangle rect_1 = Rectangle();
    Rectangle rect_2 = Rectangle(3, 3);
    Rectangle rect_3 = Rectangle(7, 5);

    try
    {
        assert(rect_1.perimetre() == 0); // longueur et largeur à 0 => Exception
    }
    catch (const exception &e)
    {
        cerr << "Exception caught: " << e.what() << endl;
    }

    assert(rect_2.perimetre() == 12); // Longueur 3 et largeur 3 => 12
    assert(rect_3.perimetre() == 24); // Longueur 7 et largeur 5 => 24

    cout << "test_rectanglePerimetre():\t:\tOK" << endl;
}
```

Surface Nous testons la récupération de la surface sur un rectangle où la largeur et/ou la longueur vaut 0
Nous testons les cas normaux ensuite.

```
void test_rectangleSurface()
{
    Rectangle rect_1 = Rectangle();
    Rectangle rect_2 = Rectangle(3, 3);
    Rectangle rect_3 = Rectangle(7, 5);

    try
    {
        assert(rect_1.surface() == 0); // longueur et largeur à 0 => Exception
    }
    catch (const exception &e)
    {
        cerr << "Exception caught: " << e.what() << endl;
    }

    assert(rect_2.surface() == 9);  // Longueur 3 et largeur 3 => 9
    assert(rect_3.surface() == 35); // Longueur 7 et largeur 5 => 24

    cout << "test_rectanglePerimetre():\t:\tOK" << endl;
}
```

Surcharge <<

```
void test_rectangleOperateurCout()
{
    Rectangle rect_1 = Rectangle();
    Rectangle rect_2 = Rectangle(3, 3);
    Rectangle rect_3 = Rectangle(7, 5, 4, 5);
    cout << "Surcharge opérateur <<" << endl;
    cout << rect_1 << endl;
    cout << rect_2 << endl;
    cout << rect_3 << endl;
}
```

Résultat des tests:

```
-- Tests de la classe Rectangle --
Exception caught: La longueur et la largeur doivent être supérieur à 0
test_rectangleConstructeur():   :       OK
Exception caught: La longueur doit être supérieur à 0
test_rectangleSetLongueur():    :       OK
Exception caught: La largeur doit être supérieur à 0
test_rectangleSetLongueur():    :       OK
Exception caught: La longueur doit être supérieur à 0
test_rectanglePerimetre():      :       OK
Exception caught: La longueur doit être supérieur à 0
test_rectanglePerimetre():      :       OK
Surcharge opérateur <<
Rectangle:
        - Centré au Point(0, 0)
        - Longueur 0
        - Largeur 0
Rectangle:
        - Centré au Point(0, 0)
        - Longueur 3
        - Largeur 3
Rectangle:
        - Centré au Point(4, 5)
        - Longueur 7
        - Largeur 5
```

Classe Carre

Classe définissant un carré, elle hérite de Rectangle.

src > Carre.cpp > ...

```cpp
#include "Carre.hpp"

/**
 * @brief Constructeur de la classe Carre
 *
 */
Carre::Carre() : Rectangle()
{
}

/**
 * @brief Surcharge du constructeur de la classe Carre
 *
 * @param taille Taille du carré
 */
Carre::Carre(float taille) : Rectangle(taille, taille)
{
}

/**
 * @brief Surcharge du constructeur de la classe Carre
 *
 * @param taille Taille du carré
 * @param ptsX Position X du Point
 * @param ptsY Position Y du Point
 */
Carre::Carre(float taille, float ptsX, float ptsY) : Rectangle(taille, taille, ptsX, ptsY)
{
}

/**
 * @brief Surcharge du constructeur de la classe Carre par recopie
 *
 * @param carre Carre à recopier
 */
Carre::Carre(const Carre &carre) : Rectangle(carre.getLongeur(), carre.getLargeur(), carre.getPoint()->getX(),
                                             carre.getPoint()->getY())
{
}
```

```cpp
/**
 * @brief Défini la taille du carré
 *
 * @param taille Nouvelle taille
 */
void Carre::setTaille(float taille)
{
    this->setLargeur(taille);
    this->setLongeur(taille);
}

/**
 * @brief Surcharge de l'opérateur <<
 *
 * @param os outputstream
 * @param carre Carré à afficher
 * @return ostream&
 */
ostream &operator<<(ostream &os, const Carre &carre)
{
    os << "Carre:\n\t- Centré au " << *carre.getPoint() << "\n\t- Taille " << carre.getLargeur();
    return os;
}
```

Test

Constructeur - Nous testons les différents constructeurs. Le constructeur utilisé est le super constructeur (celui de Rectangle)

```
416  void test_carreConstructeur()
417  {
418      Carre carre_1 = Carre();        // Constructeur par défaut
419      Carre carre_2 = Carre(3);       // Définition de la taille
420      Carre carre_3 = Carre(7, 4, 5); // Définition de la taille et positionX et positionY
421      Carre carre_4 = carre_3;        // Par recopie
422
423      assert(carre_1.getLargeur() == 0);
424      assert(carre_1.getLongeur() == 0);
425      assert(carre_1.getPoint()->getX() == 0);
426      assert(carre_1.getPoint()->getY() == 0);
427
428      assert(carre_2.getLargeur() == 3);
429      assert(carre_2.getPoint()->getX() == 0);
430      assert(carre_2.getPoint()->getY() == 0);
431
432      assert(carre_3.getLargeur() == 7);
433      assert(carre_3.getPoint()->getX() == 4);
434      assert(carre_3.getPoint()->getY() == 5);
435
436      assert(carre_4.getLargeur() == 7);
437      assert(carre_4.getPoint()->getX() == 4);
438      assert(carre_4.getPoint()->getY() == 5);
439      cout << "test_carreConstructeur():\t:\tOK" << endl;
440  }
```

SetTaille - Nous testions la définition de la taille du carré

```
442  void test_carreSetTaille()
443  {
444      Carre carre_1 = Carre();
445      Carre carre_2 = Carre(3);
446      Carre carre_3 = Carre(7, 4, 5);
447
448      carre_1.setTaille(10); // Cas normal
449      carre_3.setTaille(1);  // Cas normal
450
451      try
452      {
453          carre_2.setTaille(0);  // Taille impossible
454          carre_2.setTaille(-1); // Taille impossible
455      }
456      catch (const exception &e)
457      {
458          cerr << "Exception caught: " << e.what() << endl;
459      }
460
461      assert(carre_1.getLargeur() == 10);
462      assert(carre_1.getLargeur() == 10);
463
464      assert(carre_2.getLargeur() == 3);
465      assert(carre_2.getLargeur() == 3);
466
467      assert(carre_3.getLargeur() == 1);
468      assert(carre_3.getLargeur() == 1);
469      cout << "test_carreSetTaille():\t\t:\tOK" << endl;
470  }
```

Surcharge <<

```
472   void test_carreOperateurCout()
473   {
474       Carre carre_1 = Carre();
475       Carre carre_2 = Carre(3);
476       Carre carre_3 = Carre(7, 5, 4);
477
478       cout << "Surcharge opérateur <<" << endl;
479       cout << carre_1 << endl;
480       cout << carre_2 << endl;
481       cout << carre_3 << endl;
482   }
```

Résultat

```
-- Tests de la classe Carre --
test_carreConstructeur():       :       OK
Exception caught: La largeur doit être supérieur à 0
test_carreSetTaille():          :       OK
Surcharge opérateur <<
Carre:
        - Centré au Point(0, 0)
        - Taille 0
Carre:
        - Centré au Point(0, 0)
        - Taille 3
Carre:
        - Centré au Point(5, 4)
        - Taille 7
```

Classe ListeFormes

Classe définissant une liste de Forme, avec des méthodes pour ajouter/récupérer les formes, ainsi que pour calculer la surface totale ou obtenir le rectangle englobant toutes les formes

```cpp
#ifndef LISTE_FORMES_H_
#define LISTE_FORMES_H_

#include <vector>

#include "Forme.hpp"
#include "Rectangle.hpp"

class ListeFormes
{
private:

    vector<Forme*> liste;

public:
    ListeFormes(void);
    ~ListeFormes();

    unsigned int length(void);
    void add(Forme* forme);
    Forme* at(unsigned int index);
    unsigned int surfaceTotale(void);
    Rectangle getBox(Rectangle* box);

};


#endif // LISTE_FORMES_H_
```

```cpp
ListeFormes::ListeFormes(/* args */) {

}

ListeFormes::~ListeFormes() {

}
```

Add - La fonction add permet d'ajouter une forme à la fin de la liste

```cpp
void ListeFormes::add(Forme* forme) {
    this->liste.push_back(forme);
}
```

At - La fonction "at" permet de retourner une forme à un index donné

```cpp
Forme* ListeFormes::at(unsigned int index) {
    if (index >= this->liste.size()) {
        return nullptr;
    }

    return this->liste.at(index);
}
```

length - Cette fonction retourne la taille du vecteur, donc le nombre de forme dans la liste

```cpp
unsigned int ListeFormes::length(void) {
    return this->liste.size();
}
```

surfaceTotale - La fonction surfaceTotale calcule la somme des surfaces de toutes les formes dans la liste.

```
unsigned int ListeFormes::surfaceTotale(void) {
    int surfaceTotal = 0;
    for (Forme *forme : this->liste)
    {
        surfaceTotal += forme->surface();
    }

    return surfaceTotal;
}
```

getBox - La fonction getBox retourne le rectangle englobant toutes les formes dans l'espace. Elle regarde donc jusqu'où s'étend toutes les formes, afin de récupérer les valeurs minimale et maximale en x et y, afin de calculer la longueur et la largeur du cadre englobant les formes, puis le centre de ce cadre.

```
void ListeFormes::getBox(Rectangle* box) {
    /**
     * Pour calculer la boite qui englobe toutes les formes,
     * On observe quelle forme s'étend le plus sur les quatres directions
     *
     */
    float xMin, xMax, yMin, yMax;
    float xMin_tmp, xMax_tmp, yMin_tmp, yMax_tmp;

    float longueur, largeur, centreX, centreY;


    this->liste.at(0)->limites(&xMin, &xMax, &yMin, &yMax);

    for (unsigned int i = 1; i < this->liste.size(); i++) {
        this->liste.at(i)->limites(&xMin_tmp, &xMax_tmp, &yMin_tmp, &yMax_tmp);

        if (xMin_tmp < xMin) {
            xMin = xMin_tmp;
        }
        if (xMax_tmp > xMax) {
            xMax = xMax_tmp;
        }
        if (yMin_tmp < yMin) {
            yMin = yMin_tmp;
        }
        if (yMax_tmp > yMax) {
            yMax = yMax_tmp;
        }
    }


    longueur = xMax - xMin;
    largeur = yMax - yMin;
    centreX = xMin + (longueur / 2);
    centreY = yMin + (largeur / 2);


    box->setLongeur(longueur);
    box->setLargeur(largeur);
    box->getPoint()->setX(centreX);
    box->getPoint()->setY(centreY);

}
```

Test

Construction

Dans ce test, nous vérifions la création d'une liste de forme.
On vérifie que les formes s'ajoutes bien.

```
void test_listeFormesConstructeur(void)
{
    Cercle *cerc = new Cercle(2, 5, 6);
    Rectangle *rect_1 = new Rectangle(3, 3);
    Rectangle *rect_2 = new Rectangle(7, 5);

    ListeFormes *liste = new ListeFormes();

    cout << "==> Test ListeFormes" << endl;

    cout << "Taille liste vide : " << liste->length() << endl;

    cout << "Ajout cercle" << endl;
    liste->add(cerc);
    assert(liste->length() == 1);
    cout << "Taille avec 1 cercle : " << liste->length() << endl;
    cout << "Surface Cercle : " << liste->at(0)->surface() << endl;

    cout << "Ajout 2 rectangles" << endl;
    liste->add(rect_1);
    liste->add(rect_2);
    assert(liste->length() == 3);
    cout << "Nouvelle taille de la liste : " << liste->length() << endl;
    cout << "Perimetre du rectangle(3, 3) : " << liste->at(1)->perimetre() << endl;

    cout << "Fin test ListeFormes" << endl;
    delete cerc;
    delete rect_1;
    delete rect_2;
    delete liste;
}
```

SurfaceTotal

On teste si avec une liste connue de forme, on obtient bien la surface totale attendue et connue.

```
void test_listeFormesSurfaceTotal()
{
    ListeFormes liste;
    Rectangle *rect_2 = new Rectangle(3, 3);
    Rectangle *rect_3 = new Rectangle(7, 5);

    cout << "==> Test surface totale" << endl;
    cout << "Ajout rectangles (3, 3) et (7, 5)" << endl;

    liste.add(rect_2);
    liste.add(rect_3);

    assert(liste.surfaceTotale() == 44);
    cout << "Surface totale attendu/obtenu : 44/" << liste.surfaceTotale() << endl;

    delete rect_2;
    delete rect_3;
}
```

Boite englobante

Dans ce test, on crée une liste puis on teste la récupération de la boîte englobante.
On vérifie ensuite visuellement sur la console si les résultats sont bon

```
void test_listeFormesBoiteEnglobante(void) {
    ListeFormes liste;
    Cercle* cerc = new Cercle(2, 5, 6);
    Rectangle *rect_1 = new Rectangle(3, 3);
    Rectangle *rect_2 = new Rectangle(7, 5);

    Rectangle box;

    float xMin, xMax, yMin, yMax;

    liste.add(cerc);
    liste.add(rect_1);
    liste.add(rect_2);

    cout << "==> Test boite englobante" << endl;

    liste.at(0)->limites(&xMin, &xMax, &yMin, &yMax);
    cout << *cerc << endl;
    cout << "Limites (xmin, xmax, ymin, ymax) : " << xMin << ", " << xMax << ", " << yMin << ", " << yMax << endl;
    liste.at(1)->limites(&xMin, &xMax, &yMin, &yMax);
    cout << *rect_1 << endl;
    cout << "Limites (xmin, xmax, ymin, ymax) : " << xMin << ", " << xMax << ", " << yMin << ", " << yMax << endl;
    liste.at(2)->limites(&xMin, &xMax, &yMin, &yMax);
    cout << *rect_2 << endl;
    cout << "Limites (xmin, xmax, ymin, ymax) : " << xMin << ", " << xMax << ", " << yMin << ", " << yMax << endl;

    liste.getBox(&box);
    cout << "Boite englobante point: " << endl;
    cout << box.getPoint() << endl;
    cout << "Boite englobante: " << endl;
    cout << box << endl;
    box.limites(&xMin, &xMax, &yMin, &yMax);
    cout << "Limites (xmin, xmax, ymin, ymax) : " << xMin << ", " << xMax << ", " << yMin << ", " << yMax << endl;

    delete cerc;
    delete rect_1;
    delete rect_2;
}
```

Resultats :

```
==> Test ListeFormes
Taille liste vide : 0
Ajout cercle
Taille avec 1 cercle : 1
Surface Cercle : 12.57
Ajout 2 rectangles
Nouvelle taille de la liste : 3
Perimetre du rectangle(3, 3) : 12
Fin test ListeFormes
==> Test surface totale
Ajout rectangles (3, 3) et (7, 5)
Surface totale attendu/obtenu : 44/44
==> Test boite englobante
Cercle:
        - Centré au Point(5, 6)
        - Rayon 2
Limites (xmin, xmax, ymin, ymax) : 3, 7, 4, 8
Rectangle:
        - Centré au Point(0, 0)
        - Longueur 3
        - Largeur 3
Limites (xmin, xmax, ymin, ymax) : -3, 3, -3, 3
Rectangle:
        - Centré au Point(0, 0)
        - Longueur 7
        - Largeur 5
Limites (xmin, xmax, ymin, ymax) : -7, 7, -5, 5
Boite englobante point:
Point(0, 1.5)
Boite englobante:
Rectangle:
        - Centré au Point(0, 1.5)
        - Longueur 14
        - Largeur 13
Limites (xmin, xmax, ymin, ymax) : -14, 14, -11.5, 14.5
```

Conclusion

Le TP fonctionne dans son entièreté. Nous n’avons rencontré aucun problème particulier. Nous avons réussi à mettre en oeuvre le polymorphisme