David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

Compte-rendu SMP-TP 7

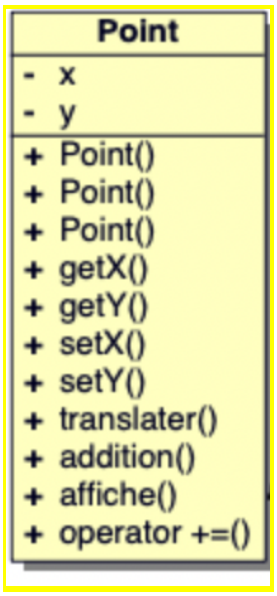
Introduction

L'objectif de ce TP est de se familiariser avec l'utilisation des classes en C++. Nous allons réaliser des classes nous permettant de travailler sur des formes géométriques.

1. Création d'une classe Point

Dans cette partie, il s'agit de créer une classe Point contenant les informations suivantes, listées par le cahier des charges :
— la méthode translater() qui prend en argument un autre Point
ou une paire de réels
— un constructeur sans paramètres qui initialise un point à l'origine
— un constructeur qui prend deux paramètres réels
— 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

Voici le diagramme de classe correspondant au cahier des charges :



Voici les spécifications des différentes méthodes de la classe Point ainsi que les tests associés :

- Méthode `translater()`

David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

Cette méthode va modifier les attributs de l'instance de la classe Point (x et y) sur laquelle est appelée la méthode.

Méthode : translater(const Point &a) Paramètres d'entrée : référence vers une instance a de la classe Point qui va donner les coordonnées de décalage Résultat : modification des coordonnées du point sur lequel est appliquée la méthode.

```
Valeur de p1 initiale :
(2, 3)
Test de la translation de (42,43):
(44, 46)
```

Nous avons bien le résultat attendu. En effet, x = 2 + 42 = 44 et y = 3 + 43 = 46, ce qui correspond bien aux valeurs obtenues.

- Constructeur initialisant un point à l'origine

Ce constructeur initialise les attributs de la classe Point à 0.

Constructeur : Point() Résultat : mise à 0 des champs x et y de la classe Point

```
Constructeur sans paramètres :
(0, 0)
```

- Constructeur prenant deux paramètres réels

Ce constructeur initialise les attributs de la classe Point aux valeurs passées en paramètre.

Constructeur : Point(float x, float y) Paramètres d'entrée : deux flottants x et y Résultat : initalisation des champs x et y de l'objet de la classe Point à leur valeur respective

```
Constructeur avec paramètres :
(2, 3)
```

- Constructeur de recopie

Ce constructeur initialise les valeurs des champs de l'objet à la même valeur que celle d'un objet Point déjà créé et passé en paramètre

Constructeur : Point(Point const &p) Paramètres d'entrée : une référence vers le Point p Résultat : initalisation des champs x et y de l'objet de la classe Point aux valeurs x et y respectives de p

David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

```
Coordonnées du point de référence :
(44, 46)
Constructeur avec référence à un autre point :
(44, 46)
```

- Accesseurs

Nous avons réalisé un accesseur par champ mais ils respectent les mêmes spécifications. Voici donc les spécifications de l'accesseur sur x.

Accesseur : float getX() const Paramètre de sortie : float correspondant à la valeur de x

```
Valeur de p1.x :
44
Valeur de p1.y :
46
```

- Mutateurs

Nous avons réalisé un mutateur par champ mais ils respectent les mêmes spécifications. Voici donc les spécifications du mutateur sur x.

Mutateur : void setX(float val) Paramètre d'entrée : nouvelle valeur du champ x sous forme de float Résultat : la valeur du champ de l'objet sur lequel est appliqué le mutateur est modifiée est vaut val

```
Modification du champ x : 44 -> 76 :
(76, 46)
Modification du champ y : 46 -> 95 :
(76, 95)
```

2. Surcharge des opérateurs

Dans cette partie, on souhaite surcharger les opérateurs `<<` et `+=`. Pour ce faire, nous allons utiliser le type `ostream`.

La surcharge de << se fait hors de la classe. Nous avons réalisé les tests pour la surcharge de << à l'affichage des points :

```
cout << p1 << endl;
```

```
Test de la surcharge de <<, affichage de p1
(76, 95)
```

La surcharge de l'opérateur += se fait à l'intérieur de la classe Point. Voici le test réalisé pour s'assurer du bon fonctionnement de la surcharge :

```
p1 += pTest; //p1 = (76, 95) et pTest = (42, 43)
cout << p1 << endl;
```

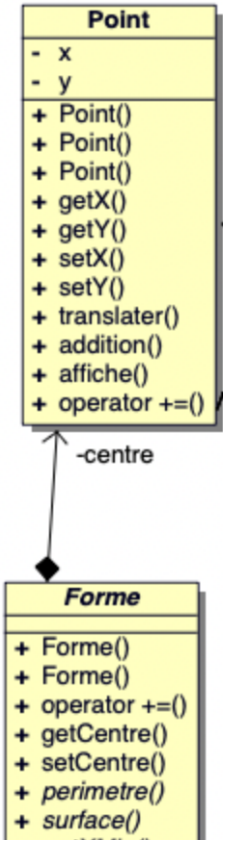
```
Test de la surcharge de += sur p1 et pTest
(118, 138)
```

On trouve donc bien le résultat souhaité.

3. Formes géométriques abstraites

Dans cette partie, il s'agit de créer une classe Forme qui représente une forme géométrique fermée centré sur un point particulier. Cette nouvelle classe contient un membre *centre* de type point qui est le point central de la forme. Ainsi que les deux méthodes *perimetre* et *surface* qui permettent d'obtenir le périmètre et la surface de la forme. Et enfin on surcharge les opérateurs += et <<.

Voici le diagramme de classe correspondant au cahier des charges :



Classe forme avec un Point central

La classe Forme représente une forme géométrique abstraite définie par la position de son centre, stocké dans l'attribut centre de type Point. Elle dispose d'un constructeur prenant un point en paramètre afin d'initialiser le centre à une position donnée, ainsi que d'un constructeur par défaut qui initialise le centre à l'origine du repère (0,0). Cette classe fournit également des méthodes d'accès et de modification du centre.

David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

Voici l'implémentation de cette classe :

```
class Forme {
private:
   Point centre;

public:
   Forme(const Point& centre) {
       this->centre = centre;
   }

   Forme() {
       this->centre = Point(0, 0);
   }

   Point getCentre() const;
   void  setCentre(const Point &p);
};
```

Surcharge de l'opérateur +=

Cette fonction surcharge l'opérateur += afin de permettre la **translation d'une forme**. Elle prend en paramètre un point p représentant le vecteur de déplacement et applique cette translation au centre de la forme via la méthode translater, ce qui déplace la forme entière sans en modifier la géométrie.

```
void Forme::operator+=(Point &p) {
   centre.translater(p);
}
```

Surcharge de l'opérateur <<

Cette fonction surcharge l'opérateur << afin de permettre l'affichage d'un objet Forme avec std::cout. Elle affiche le périmètre et la surface de la forme en appelant les fonctions virtuelles perimetre() et surface(), ce qui permet un comportement polymorphe selon le type réel de la forme. Ainsi, la classe Forme est abstraite et sert de classe de base virtuelle pour les formes dérivées.

Voici son implémentation :

```
ostream& operator<<(std::ostream &os, Forme &f)
{
   os << "Forme:" << endl;
   os << "  | Perimetre = " << f.perimetre()
      << "  | Surface = " << f.surface();
```

```
    return os;
}
```

Déclaration de deux méthodes abstraites perimetre() et surface()

Les méthodes perimetre() et surface() doivent être déclarées abstraites car la classe Forme est une classe mère générique qui ne représente aucune forme géométrique concrète. Il n'existe pas de formule unique permettant de calculer le périmètre ou la surface d'une forme abstraite : ces calculs dépendent du type réel de la forme (cercle, rectangle, carré, etc.). En les déclarant virtuelles pures (= 0), on impose aux classes dérivées de fournir leur propre implémentation, tout en permettant l'utilisation du polymorphisme

Voici un extrait de la déclaration de la classe Forme dans le fichier Forme.h :

```
virtual float perimetre() = 0;
virtual float surface()   = 0;
```

4. Formes géométriques concrètes

Afin de spécialiser la classe abstraite Forme, trois classes concrètes ont été implémentées : Cercle, Rectangle et Carre. Chacune de ces classes est définie dans des fichiers séparés .h et .cpp, conformément aux bonnes pratiques de programmation orientée objet en C++.

Classe Rectangle

La classe Rectangle hérite de la classe abstraite Forme. Elle est caractérisée par deux attributs : la largeur et la hauteur. Les côtés du rectangle sont supposés alignés avec les axes horizontal et vertical.

Cette classe redéfinit les méthodes abstraites perimetre() et surface() en utilisant les formules propres au rectangle. L'opérateur << est surchargé afin d'afficher le type de la forme ainsi que les valeurs de ses attributs.

Les tests suivants ont pour objectif de vérifier le bon fonctionnement de la classe Rectangle, tant sur la définition géométrique que sur les opérations associées. Dans un premier temps, deux rectangles r1 et r2 sont créés avec des sommets alignés sur les axes, mais définis dans un ordre différent, afin de valider que la classe calcule correctement le périmètre et la surface indépendamment de l'ordre d'affectation des coins. Les assertions confirment que les dimensions (2×1 puis 1×2) conduisent bien à un périmètre de 6 et une surface de 2. Ensuite, une translation de r1 par le vecteur (1,1) est effectuée via la surcharge de l'opérateur +=, ce qui permet de tester à la fois la modification correcte des coordonnées des sommets et l'invariance des propriétés géométriques (surface et périmètre inchangés après déplacement). Enfin, le rectangle r3, dont les côtés ne sont pas alignés avec les axes,

permet de vérifier que les méthodes surface() et perimetre() fonctionnent également pour un rectangle incliné. L'utilisation d'une tolérance (0.01f) dans les assertions tient compte des imprécisions liées aux calculs en nombres flottants.

```
// =======Test de la classe Rectangle =========

cout << "Test de la classe Rectangle" << endl;

Rectangle r1;

r1.setCoin2(0, 1); r1.setCoin3(2, 1);

r1.setCoin1(0, 0); r1.setCoin4(2, 0);

cout << r1;

assert(r1.perimetre() == 6.0);

assert(r1.surface() == 2.0);

Rectangle r2;

r2.setCoin2(0, 2); r2.setCoin3(1, 2);

r2.setCoin1(0, 0); r2.setCoin4(1, 0);

cout << r2;

assert(r2.perimetre() == 6.0);

assert(r2.surface() == 2.0);

// Traslation de r1 de (1, 1)

r1 += Point(1, 1);

assert(r1.getCoin2().getX() == 1 && r1.getCoin2().getY() == 2);
assert(r1.getCoin3().getX() == 3 && r1.getCoin3().getY() == 2);

assert(r1.getCoin1().getX() == 1 && r1.getCoin1().getY() == 1);
assert(r1.getCoin4().getX() == 3 && r1.getCoin4().getY() == 1);

assert(r1.perimetre() == 6.0);

assert(r1.surface() == 2.0);

// Rectangle avec ces côtés non alignés avec les axes x et y

Rectangle r3;

r3.setCoin2(0, 1); r3.setCoin3(2, 3);
```

```
r3.setCoin1(1, 0); r3.setCoin4(3, 2);

cout << r3;

assert(r3.surface()    - 4      < 0.01f);

assert(r3.perimetre() - 8.49f < 0.01f);
```

Voici la sortie de ces tests qui valide bien la classe Rectangle :

```
Test de la classe Rectangle
Rectangle:
  | Perimetre = 6
  | Surface = 2
  | Centre = (0, 0)
  | Coin1 = (0, 0)
  | Coin2 = (0, 1)
  | Coin3 = (2, 1)
  | Coin4 = (2, 0)
Rectangle:
  | Perimetre = 6
  | Surface = 2
  | Centre = (0, 0)
  | Coin1 = (0, 0)
  | Coin2 = (0, 2)
  | Coin3 = (1, 2)
  | Coin4 = (1, 0)
Rectangle:
  | Perimetre = 8.48528
  | Surface = 4
  | Centre = (0, 0)
  | Coin1 = (1, 0)
  | Coin2 = (0, 1)
  | Coin3 = (2, 3)
  | Coin4 = (3, 2)
```

Classe Carre

La classe Carre hérite de la classe Rectangle. Elle représente un cas particulier de rectangle dont la largeur et la hauteur sont égales.

David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

Elle réutilise ainsi les fonctionnalités de Rectangle tout en garantissant l'égalité des côtés via son constructeur. Les méthodes perimetre() et surface() sont adaptées au cas du carré, et l'opérateur << est également surchargé pour afficher le type Carre et la longueur de son côté.

Les tests suivants visent à valider le bon comportement de la classe Carre, en vérifiant à la fois les contraintes géométriques spécifiques au carré et les fonctionnalités héritées (notamment si Carre dérive de Rectangle). Dans un premier temps, l'objet car1 est défini avec des sommets alignés sur les axes pour former un carré de côté 2. Les assertions permettent de confirmer que les méthodes perimetre() et surface() retournent respectivement 8 et 4, ce qui valide le calcul correct des dimensions et le respect de l'égalité des côtés. Ensuite, une translation de vecteur (1,1) est appliquée via l'opérateur +=, afin de vérifier que toutes les coordonnées sont correctement mises à jour et que les propriétés géométriques du carré (longueur des côtés, surface et périmètre) restent invariantes après déplacement. Enfin, le carré car2, dont les côtés ne sont pas parallèles aux axes, permet de tester la robustesse des calculs pour une configuration inclinée. L'utilisation d'une tolérance (0.01f) dans les assertions prend en compte les éventuelles imprécisions liées aux calculs en virgule flottante, notamment pour le périmètre basé sur des distances calculées avec des racines carrées.

```
// =======Test de la classe Carre =========

cout << "Test de la classe Carre" << endl;

Carre car1;

car1.setCoin2(0, 2); car1.setCoin3(2, 2);

car1.setCoin1(0, 0); car1.setCoin4(2, 0);

cout << car1;

assert(car1.perimetre() == 8.0);

assert(car1.surface() == 4.0);

// Translation de car1 de (1, 1)

car1 += Point(1, 1);

assert(car1.getCoin2().getX() == 1 && car1.getCoin2().getY() ==
3); assert(car1.getCoin3().getX() == 3 && car1.getCoin3().getY()
== 3);

assert(car1.getCoin1().getX() == 1 && car1.getCoin1().getY() ==
1); assert(car1.getCoin4().getX() == 3 && car1.getCoin4().getY()
== 1);

assert(car1.perimetre() == 8.0);
```

```
assert(car1.surface() == 4.0);

// Carré avec ces côtés non alignés avec les axes x et y

Carre car2;

car2.setCoin2(0, 1); car2.setCoin3(1, 2);

car2.setCoin1(1, 0); car2.setCoin4(2, 1);

cout << car2;

assert(car2.perimetre() - 5.65f < 0.01f);

assert(car2.surface()   - 2.0f  < 0.01f);
```

Voici la capture d'écran de la sortie des tests qui valide la classe Carre :

```
Test de la classe Carre
Carre:
  | Perimetre = 8
  | Surface = 4
  | Centre = (0, 0)
  | Coin1 = (0, 0)
  | Coin2 = (0, 2)
  | Coin3 = (2, 2)
  | Coin4 = (2, 0)
Carre:
  | Perimetre = 5.65685
  | Surface = 2
  | Centre = (0, 0)
  | Coin1 = (1, 0)
  | Coin2 = (0, 1)
  | Coin3 = (1, 2)
  | Coin4 = (2, 1)
```

Classe Cercle

La classe Cercle hérite directement de la classe Forme. Elle est définie par un attribut rayon.

Elle implémente les méthodes abstraites perimetre() et surface() à l'aide des formules mathématiques associées au cercle. Comme pour les autres formes, l'opérateur << est surchargé afin d'afficher le type de la forme ainsi que la valeur de son rayon.

Voici le code permettant de tester la classe Cercle :

David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

```
cout << "Test de la classe Cercle" << endl;
Cercle c1;
c1.setCentre(Point(1, 1));

assert(c1.getCentre().getX() == 1);
assert(c1.getCentre().getY() == 1);

c1.setRayon(2);
cout << "Cercle c1: " << c1 << endl;
assert(c1.perimetre() == static_cast<float>(2.0 * M_PI * 2.0));
assert(c1.surface()   == static_cast<float>(M_PI * 2 * 2));
```

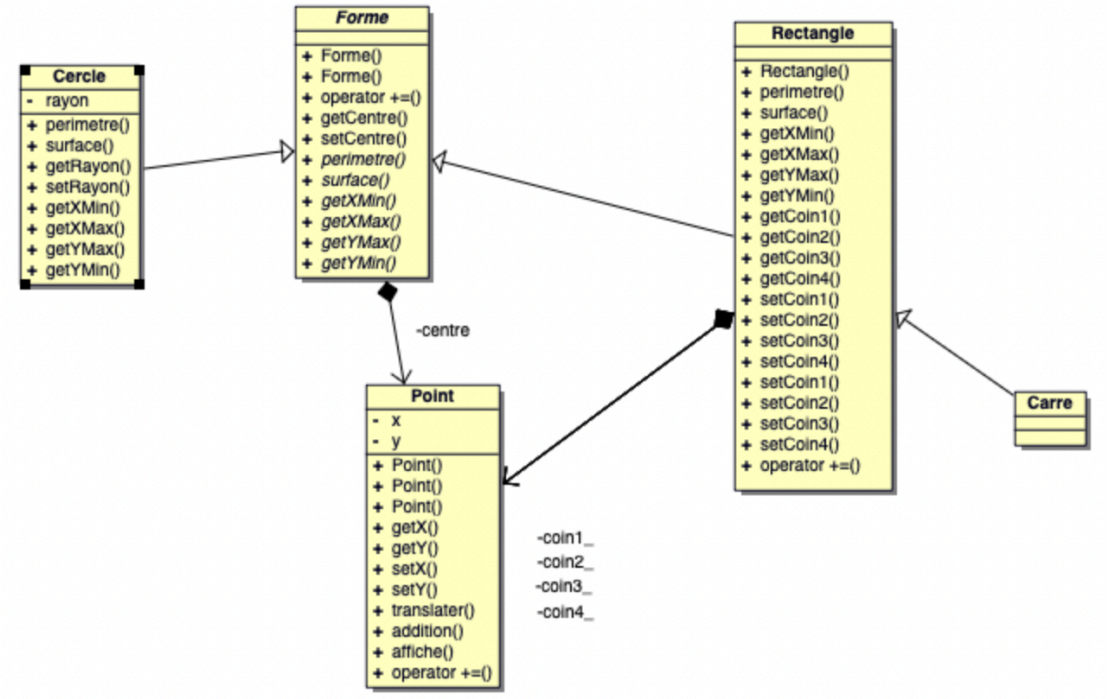
Ce code réalise un jeu d'essais permettant de vérifier le bon fonctionnement de la classe Cercle. Il commence par créer un objet Cercle par défaut, puis modifie la position de son centre en le plaçant au point (1,1). Des assertions sont ensuite utilisées pour vérifier que les coordonnées du centre ont bien été mises à jour. Le rayon du cercle est fixé à 2, puis l'objet est affiché à l'aide de l'opérateur << surchargé. Enfin, les deux dernières assertions permettent de valider la justesse des calculs du périmètre et de la surface en comparant les valeurs retournées par les méthodes perimetre() et surface() aux formules mathématiques théoriques du cercle.

```
Test de la classe Cercle
Cercle c1: Cercle:
  | Perimetre = 12.5664
  | Surface = 12.5664
  | Centre = (1, 1)
  | Rayon = 2
```

La sortie valide bien que l'implémentation répond au cahier des charges.

David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026



Le diagramme de classes met en évidence les relations d'héritage entre les différentes classes.

La classe Forme est une classe abstraite contenant un attribut centre de type Point et deux méthodes virtuelles pures perimetre() et surface().

Les classes Cercle et Rectangle héritent directement de Forme, tandis que la classe Carre hérite de Rectangle, illustrant une spécialisation supplémentaire. Ce diagramme permet de visualiser clairement les dépendances et la structure hiérarchique du modèle objet.

5. Liste de Formes

Dans cette partie, il est demandé d'utiliser la structure de données dynamique vector<> de la bibliothèque standard C++ afin de concevoir une classe ListeFormes. Cette classe a pour rôle de gérer un ensemble de pointeurs vers des objets de type Forme. En plus des fonctionnalités classiques telles que l'accès à une forme donnée et l'ajout d'une nouvelle forme dans la collection, la classe doit proposer deux traitements supplémentaires : le calcul de la surface totale correspondant à la somme des surfaces de toutes les formes stockées, ainsi que la détermination de la boîte englobante de la liste. Cette boîte englobante correspond au plus petit rectangle aligné avec les axes du repère capable de contenir l'intégralité des formes présentes dans la collection.

La solution consiste à définir : un attribut privé qui est un vector qui stocke une référence vers chaque forme, le mutateur et l'accesseur associé, une méthode qui ajoute une forme à la liste, une méthode qui calcule la surface totale de toute les formes, une méthode qui retourne la boîte englobante et on surcharge l'opérateur pour pouvoir afficher chaque forme :

David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

```
class ListeFormes{

   private:

        vector<Forme*> liste;

   public:

        //Accesseur

        Forme* getForme(int i) const;

        vector<Forme*> getFormes() const;

        void ajout(Forme *f);

        float surfaceTotale() const;

        Rectangle boiteEnglobante() const;

};

ostream &operator<<(std::ostream &s, ListeFormes &f);
```

Une première version de la surcharge de l'opérateur << appel la surcharge de cette opérateur par la classe Forme :

```
ostream& operator<<(ostream &os, ListeFormes &f)

{

   os << "ListeFormes:" << endl;

   for (Forme *forme : f.getFormes()) {

        os << *forme;

   }

   return os;

}
```

Voici l'affichage de la liste contenant r1, r2, c1 et car1 :

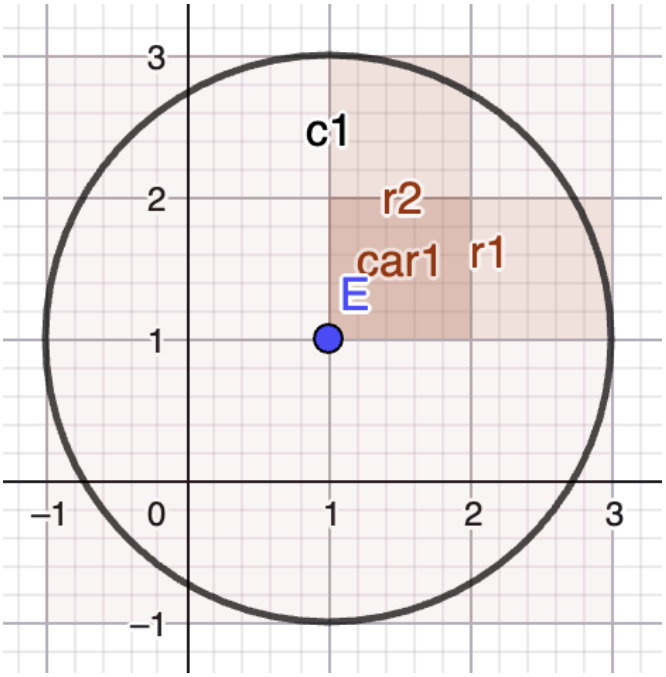
David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

```
ListeFormes:
Forme:
  | Perimetre = 6
  | Surface = 2
Forme:
  | Perimetre = 6
  | Surface = 2
Forme:
  | Perimetre = 12.5664
  | Surface = 12.5664
Forme:
  | Perimetre = 8
  | Surface = 4
```

Quant à la méthode `boiteEnglobante()` calcule le plus petit rectangle capable de contenir l'ensemble des formes stockées dans la liste. Pour cela, elle initialise d'abord quatre variables représentant les bornes extrêmes du futur rectangle : `xMin` et `yMin` sont initialisées à la plus grande valeur flottante possible afin de pouvoir être réduites lors des comparaisons, tandis que `xMax` et `yMax` sont initialisées à la plus petite valeur flottante positive afin de pouvoir être augmentées. La fonction parcourt ensuite toutes les formes de la collection et met à jour ces valeurs en comparant, pour chaque forme, ses coordonnées minimales et maximales (`getXMin()`, `getYMin()`, `getXMax()`, `getYMax()`). À la fin du parcours, `xMin` et `yMin` correspondent aux plus petites coordonnées trouvées, et `xMax` et `yMax` aux plus grandes. Ces quatre valeurs permettent alors de définir les quatre coins d'un rectangle aligné avec les axes du repère, qui constitue la boîte englobante. Le rectangle ainsi construit est finalement retourné par la fonction.

Voici la représentation de toutes les formes contenues dans la liste. Le rectangle sur fond rouge clair est la boite englobante.



David PROSPERIN
Riana CHAUVET RAHERIMALALA

28/01/2026

La capture d’écran ci-dessous valide bien notre méthode boite englobante :

```
Boîte englobante de la liste de formes:
Rectangle:
  | Perimetre = 16
  | Surface = 16
  | Centre = (0, 0)
  | Coin1 = (-1, -1)
  | Coin2 = (-1, 3)
  | Coin3 = (3, 3)
  | Coin4 = (3, -1)
```

Pour conclure, ce TP nous a permis de mettre en pratique les principes fondamentaux de la programmation orientée objet en C++, notamment l’encapsulation, l’héritage, le polymorphisme et la surcharge d’opérateurs. À travers la conception progressive des classes Point, Forme puis des formes concrètes Rectangle, Carre et Cercle, nous avons construit une architecture cohérente respectant le cahier des charges et illustrant clairement les relations d’héritage. La mise en œuvre de méthodes virtuelles pures dans la classe abstraite Forme nous a permis d’exploiter le polymorphisme, tandis que l’utilisation d’un vector<Forme*> dans ListeFormes a mis en évidence la gestion dynamique d’objets et la manipulation de collections polymorphes. Les nombreux tests réalisés à l’aide d’assertions ont validé la justesse des calculs géométriques ainsi que le bon fonctionnement des translations et des surcharges d’opérateurs. Ce TP nous a ainsi permis de consolider notre compréhension de la conception orientée objet en C++ et de mieux structurer un projet autour d’un modèle clair, évolutif et rigoureux.