Aix-Marseille Université
Master 2 F.I.

Introduction au C++ - TP

A) Préambule

Ce TP a pour but de vous faire manipuler des classes C++ ainsi que les différentes notions associées expliquées en cour. Pour cela vous allez partir d'une classe de base à laquelle vous aurez à ajouter diverses fonctionnalités. Dans un second temps, vous verrez comment étendre cette classe et la faire communiquer avec d'autres classes que vous développerez aussi.
L'ensemble des sources, ainsi que les fichiers "projets" utilisés par `Code::Blocks` (*TP1.cbp* et *TP2.cbp*), nécessaires à ce TP peuvent être récupérés à l'adresse :

`https://github.com/GFuhr/M2_FI`

Partie 1 : La classe de base - la classe Point

B) Introduction

La classe Point est une classe simple définissant des points 2D : chaque point est représenté par deux membres, son abscisse $\_x$ et son ordonnée $\_y$, qui sont de type double. La déclaration de la classe est la suivante :

```
class Point {
    double  _x ;
    double  _y ;
    double norm;

public:
    Point(void) ;
    Point(const double val_x, const double val_y);
    Point(const Point &p);
    ~Point(void);

    void setX(const double val);
    void setY(const double val);
    double  getX(void) const ;
    double  getY(void) const ;
    double  distance(const Point &p) const;
 };
```

La déclaration de la classe se trouve dans le fichier *Classe_point.h*.

C) Définition des fonctions membres de la classe

Remarque : Votre code source ne pourra être compilé qu'une fois la fin de cette partie atteinte.
La définition des divers constructeurs, fonctions... de la classe ***Point*** seront à faire dans le fichier *Classe_point.cpp*.

1. Définissez le constructeur par défaut tel que les valeurs de $\_x$ et $\_y$ soient égales à 0.
2. Ajoutez ensuite à ce constructeur la ligne suivante :

```
std::cout << "appel du constructeur par defaut" << std::endl;
std::cout << "pour l'objet :"<<(long)this<<std::endl;
```

3. Que permet cette instruction et quel est son intérêt ?
4. Ajoutez la ligne suivante au corps du destructeur :

```
std::cout << "appel du destructeur " << std::endl;
std::cout << "pour l'objet :"<<(long)this<<std::endl;
```

5. Définissez le constructeur avec argument, permettant de stocker la valeur de l'argument val_x dans la donnée membre $\_x$ et la valeur de val_y dans $\_y$.
6. Définissez le constructeur par copie, permettant de copies les données membres de l'objet passé en argument (p) dans l'objet actuel.
7. Ajoutez pour chaque constructeur (constructeur avec arguments et constructeur par copie) une ligne permettant d'afficher à l'écran quel constructeur est appelé.
8. Étant donné le type des données membres de notre classe, devons nous faire autre chose dans le destructeur associé ? pourquoi ?

D) Exemple d'utilisation

Maintenant que nous avons déclaré et défini notre classe, nous allons voir comment l'utiliser. Pour cela nous allons nous intéresser à notre fonction *main* afin de créer plusieurs points. La fonction *main* a modifier se trouve dans le fichier *maint_Point.cpp*. Écrivez votre fonction *main* de telle sorte qu'elle contiendra 4 points.

1. un point A qui utilisera le constructeur par défaut
2. un point B qui utilisera le constructeur surchargé
3. un point C qui utilisera le constructeur par copie
4. Un point D qui sera une référence sur un des points définis avant
5. Un point E qui sera défini via un pointeur sur un point et l'opérateur new
6. Que constatez vous entre le nombre d'appel aux destructeurs et le nombre d'appel aux constructeurs ? Et concernant l'ordre des appels ?

E) Fonctions d'interface

1. Définissez les fonctions : getX et getY qui renvoient respectivement la valeur de $\_x$ et $\_y$, et setX(/Y) qui permettent de modifier la valeur de $\_x$(/ $\_y$)

2. Définissez la fonction distance.
   Rappel : la distance entre 2 points (x_1, y_1) et (x_2, y_2) est définie par

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

   — Utilisez dans la fonction main les fonctions membres getX et getY pour afficher les coordonnées des points B et E.
   — Utilisez dans la fonction main les fonctions setX et setY pour modifier les coordonnées des points B et E.
   — Utilisez dans la fonction main la fonction membre distance pour calculer la distance entre les points suivants : A et B, C et D, E et A, E et D. Affichez le résultat obtenu.
   — Si, dans la déclaration et la définition de votre fonction distance, vous supprimez l'utilisation des références, cela va-t-il changer quelque chose à la définition de votre fonction ? et au niveau de l'utilisation, remarquez vous une différence ?

```
double distance(const Point &p);
```

devient,

```
double distance(const Point p);
```

3. Que devez vous changer dans la syntaxe d'appel de votre fonction membre lorsque vous l'utilisez depuis l'objet E ? Devez vous changer la syntaxe d'appel lorsque vous utilisez votre fonction membre depuis l'objet D ?

F) Gestion de la mémoire

1. Dans la fonction main(...), créez un tableau F de 3 Point (avec new). Appelez le destructeur du tableau.([] delete) Que remarquez-vous ?
2. Pour vérifier si la mémoire a bien été libérée, déclarez un nouveau tableau G de 3 Point après l'appel au destructeur de F, et affichez l'adresse de ce tableau ainsi que celle de F[0], de F[1] et de F[2], dans le cas où F a été détruit via *// delete*.
   Remarque : pour afficher l'adresse de l'élément F[idx] utilisez l'opérateur (&) de la manière suivante avec *cout*

   ```
   std::cout<<"adresse de F[idx] :"<<(long)&(F[idx])<<std::endl
   ```

3. Sachant qu'un double occupe 8 octets en mémoire, et en utilisant les valeurs des adresses affichées, que pouvez-vous en déduire sur le stockage des éléments du tableau ?
4. Dans la fonction main(...), créez un tableau H de 3 Point (avec les vectors). Que remarquez-vous, en particulier au niveau des constructeurs/destructeurs ? Conclusion ?

G) Nouvelle fonctionnalité

Nous allons chercher à ajouter un autre élément à notre classe, à savoir le calcul de la norme du point. Pour un point P, de coordonnées x_i et y_i, on qualifie de norme la quantité suivante, il s'agit en fait de la distance entre notre point et l'origine :

$$norm(P) = \sqrt{(x_i)^2 + (y_i)^2}$$

1. ajoutez une variable membre pour stocker la valeur de la norme. Quel doit-être le type d'accès à cette variable membre : public, private, protected ?
2. ajoutez une fonction membre à la classe **point** pour calculer cette norme. Cette fonction devra uniquement calculer la valeur et la stocker dans votre variable membre sans renvoyer de valeur.
3. Ajoutez une fonction permettant de renvoyer la valeur de la norme.
4. Modifiez votre classe de manière à ce que, dès qu'un objet est crée ou alors dès que la valeur de x et/ou y est modifiée, la norme soit recalculée automatiquement.

Partie 2 : La classe Graphe - Notion de Composition et de tableaux

H) Préambule

Dans cette seconde partie, nous allons reprendre la classe ***Point*** comme base pour une nouvelle classe ***Graphe***. Un graphe est constitué d'un ensemble de points (représentés par la donnée membre ArrayPoint) qui vont correspondre à un ensemble de coordonnées (x_i, y_i). Nous allons dans ce TP déclarer cette nouvelle classe ***Graphe*** avec de nouvelles fonctionnalités par rapport à la classe ***Point***. Dans toute la suite nous ferons l'hypothèse que un graphe de N points sera constitué des points : Point0, Point1... Point(N-1) Cette seconde partie va vous permettre de manipuler des tableaux, aussi bien via la notion d'allocations dynamiques que via la notion de *vector* de la STL. Ceci vous permettant de constater les apports de la STL par rapport aux apports uniquement liés au langage.

Remarque générale :

- Pour chaque constructeur/destructeur que vous définirez dans la suite, vous ajouterez une ligne permettant d'afficher à l'écran quelle est la fonction utilisée.
  Exemple pour le constructeur par défaut de la classe ***Graphe***, vous ajouterez :
  `std::cout<<"constructeur par defaut pour la classe Graphe"<<std::endl;`
  `std::cout<<"pour le graphe : "<<(long)this<<std::endl;`
- Toutes les définitions devront être faites dans le fichier *Classe_ graphe.cpp* et les déclarations des fonctions membres supplémentaires dans le fichier *Classe_ graphe.h*

Rôle des données membres :

1. *ArrayPoint* : tableau de type vector qui contiendra la liste des points
2. *iNbPoint* : variable qui contiendra le nombre de points du graphe
3. *point_ dist* : tableau dynamique qui servira à stocker la distance entre chaque point et le point suivant du graphe.

I) Déclaration de la classe Graphe et de ses membres

1. Définissez les constructeurs suivants dans le fichier *Classe_ graphe.cpp* :
   - Un constructeur par défaut
   - Un constructeur pour lequel on spécifiera le nombre de points que possédera le ***Graphe*** et l'on initialisera tous les points avec des coordonnées au choix (x_i, y_i). Par exemple $y_i = \sin(x_i)$.
   - Un constructeur pour lequel on spécifiera le nombre de points ainsi qu'un tableau de ***Point*** qui constitueront le graphe
   - Un constructeur pour lequel on transmettra un tableau de ***vector<Point>***.
   - Un constructeur par copie.

   Dans chaque constructeur, toutes les données membre doivent être initialisées.
2. Est ce qu'un appel au constructeur de la classe ***Graphe*** entraîne automatiquement un appel à un des constructeur de la classe ***Point*** ?Si oui lequel ? Pourquoi ?
3. Définissez le destructeur

J) Déclarations et Définitions des fonctions annexes

Dans la liste des données membres de la classe, il y a une donnée qui n'a pas encore été utilisée, à savoir la variable ***point_ dist***.

1. Ajoutez la déclaration et la définition d'une fonction membre privée permettant de calculer pour chaque point N du polygone la distance entre ce point et son voisin, à savoir le point $N+1$. Pour le dernier point, vous calculez la distance entre ce point et le premier point. Ces valeurs seront stockées dans le tableau ***point_ dist***.

2. Une fonction permettant d'ajouter N ***Point*** à un ***Graphe***. Justifiez votre choix d'arguments pour cette fonction .
3. En vous servant des déclarations déjà écrites pour la classe, définissez les fonctions membres suivantes :
   — Une fonction permettant d'obtenir un ***Point*** du ***Graphe*** en spécifiant uniquement son index.
   — Une fonction permettant de récupérer le nombre de ***Point*** dans un ***Graphe*** .
   — Une fonction permettant d'afficher l'ensemble des ***Point*** du ***Graphe***.
   — Une fonction permettant de calculer le périmètre du polygone.
   — Une fonction permettant de calculer l'aire. Pour un polygone quelconque, l'aire du polygone peut-être obtenu via la formule,

$$Aire = \frac{1}{2}\left|\sum_{n=1}^{N}(x_n y_{(n+1)\%n} - y_n x_{(n+1)\%n})\right|$$

   Le % correspond à la fonction mathématique modulo.

K) Création de Graphe

1. Vous créerez dans votre fonction principale *main* plusieurs Graphes afin d'utiliser les fonctions définies dans votre classe :
   — Un cas avec le constructeur par défaut
   — Un cas ou vous générerez un tableau de Points que vous utiliserez pour initialiser votre tableau de Graphe
   — Un cas par copie d'un autre Graphe.
   — Un cas utilisant le constructeur prenant un objet de type vector en argument.
   — Á l'aide de votre classe Graphe, déclarez une variable dont le graphe correspond à un carré de côté 1 et un rectangle de longueur 4 et largeur 2.
   — Pour ces 2 cas, affichez l'aire et le périmètre associés, et vérifiez que les valeurs sont correctes.

Partie 3 : Extension de la classe graphe - Notion d'héritage

L) Classe Rectangle

Un rectangle peut être vu comme un cas particulier de graphe. À savoir un graphe de 4 cotés qui peut-être entièrement défini à partir des coordonnées du Point0 et de 2 longueurs L_x et L_y.

1. Définissez une classe rectangle qui héritera de la classe ***CGraphe*** qui contiendra
   - Un constructeur par défaut qui appelera le constructeur surchargé de la classe ***CGraphe*** en spécifiant le nombre de points et pour lequel il sera considéré que le rectangle a une largeur de 1 et une longueur de 2
   - Un constructeur surchargé pour lequel on spécifiera le Point0 ainsi que les longueurs L_x et L_y
   - Un destructeur
   - Une fonction membre qui renverra l'aire du rectangle et qui aura le même nom que la fonction que vous aurez défini dans la classe ***CGraphe***.

Pour la définition de cette classe, le choix dans la partie 2 de n'utiliser que des variables privées était-il le plus judicieux ?

M) Classe Carré

De même qu'un rectangle peut-être considéré comme un polygone particulier, un carré peut-être considéré comme un rectangle particulier avec $L_x = L_y = L$.

1. Définissez une classe ***Carre*** qui sera une classe fille de la classe ***Rectangle*** qui contiendra, entre autre,
   - Un constructeur par défaut qui appelera le constructeur par défaut de la classe ***Rectangle***
   - Un constructeur surchargé pour lequel on spécifiera le Point0 ainsi que la longueur L. Est-il possible d'utiliser directement un des constructeur de la classe Rectangle pour cette fonction ? Si oui, comment.
   - Un destructeur
   - A-t-on besoin de définir une nouvelle fonction *calcAire* ou peut-on utiliser celle définie dans la classe ***Rectangle*** ? Pourquoi

N) Utilisation

1. Maintenant que vous avez définis vos nouvelles classes, déclarez de nouvelles variables correspondant à un carré de côté 1 et un rectangle de longueur 4 et largeur 2. La déclaration de ces variables est-elle differente du cas précédent ?
2. Si vous comparez le nombre d'appels à des constructeurs/destructeurs ainsi que l'ordre des appels entre ce cas la et le rectangne et le carré définis dans la partie précédente, que constatez vous ? Que pouvez vous dire sur l'ordre des appels engendrés ?
3. Quels avantages/inconvénients voyez vous à l'utilisation de cette nouvelle méthode ?
4. Concernant le calcul des aire, comment pouvez vous savoir si la fonction appelée est celle de la classe ***CGraphe*** où celle de la classe fille ?

Annexe 1 : Vector et STL

Soit la variable $varV$ correspondant à un vector d'un type T .

```
std::vector<T> varV;
```

- spécification de la taille du tableau lors de la déclaration de la variable :
  `std::vector<T> varV(taille_tableau);`
- nombre d'elements dans le tableau associé :
  `varV.size();`
- modifier la taille du tableau associé :
  `varV.resize(new_size);`
- accéder à l'élément idx :
  `varV.at(idx);`
  `varV[idx];`

Annexe 2 : Notation des sources

O) Préambule
Ce document liste les mauvaises pratiques qui serviront à évaluer les programmes écrits lors des TP. En priorité, le code doit compiler et s'éxecuter pour fournir les resultats attendus. Tous ces critères correspondent aux normes et bonnes pratiques de développement C++ pour la production de codes fonctionnels et réduisants le risque d'erreurs.
Ces critères vont être classés en 4 niveaux, avec un niveau de validé uniquement si toutes les conditions liées à ce niveau le sont.

Niveau 1
- Utiliser les bons types d'accès pour les fonctions/données membres d'une classe : public, private ou protected.
- Ne pas utiliser de variables globales et/ou de "#define".
- Utiliser les équivalent C++ de fonctions C *"classiques"*.

Niveau 2
- Initialiser toutes les variables lors des déclarations.
- Faire une allocation dynamique plutôt que déclarer des tableaux statiques de taille disproportionnée.
- Commenter le code.

Niveau 3
- Libérer la mémoire (via **delete** ou **delete[]**) pour un bloc mémoire alloué via **new**, **new []**.
- Tester les valeurs de retour des fonctions.

Niveau 4
- Faire de tests sur les fonctions globales/les fonctions membres pour s'assurer des résultats fournis.
- S'assurer de la "const correctness".

Rappel : commentaire d'une fonction

```
/**
 * definition :  type_retour nom_fonction(type_1 const param1, type_2 param2);
 * description :  But de  fonction
 *
 * param  : param1: variable de type type_1 qui va servir à...,
                     non modifié par la fonction
 * param  : param2: variable de type type_2 qui va servir à...,
                    modifié par la fonction pour stocker tel résultat
 * valeur renvoyée : variable de type, type_retour, qui  contiendra telle valeur.
                    Il peut s'agir d'un code erreur, du résultat d'un calcul...
 **/
```