Séance 5
Mise au point d'une classe Polygone

1 Objectif du TP :

— Valeurs / entités
— Notion de composition / agrégation
— allocat1ion-libération d'un tableau
— La "règle de trois" : destructeur, constructeur de copie, affectation

2 Thème : points et polygones

On souhaite disposer d'une classe Polygone, un polygone étant défini à partir de Points qui sont ses sommets. Exemple de code :

```
1 Point a {0,0}, b {30,0}, c {0,50};
2 Polygone triangle;
3 triangle.ajouterSommet(a);
4 triangle.ajouterSommet(b);
5 triangle.ajouterSommet(c);
6
7 cout << "Le triangle : " << triangle.toString() <<
    endl;
```

Et ce code devrait afficher :

```
Le triangle : { (0,0) (30,0) (0 50) }
```

3 Sémantique de valeur et d'entité

Attention, cet exemple, innocent en apparence, se prête à deux interprétations bien différentes. Si on fait :

```
1 a.deplacer(100,100);
2 cout << triangle.toString() << endl;
```

Le triangle s'est il déformé ou non ?

3.1 Sémantique de valeur

On peut penser raisonnablement qu'il s'affiche (0,0) (30,0) (0 50) parce qu'on a construit le Polygone à partir des valeurs des informations (coordonnées) contenues dans les points. Bouger ensuite un point n'a aucun effet sur le polygone.

On parle dans ce cas de **sémantique de valeur**.

Les coordonnées des sommets appartiennent au Polygone, qui est donc une composition de Points.

3.2 Sémantique d'entité

Tout aussi raisonnablement, on peut imaginer qu'il s'affiche (100,100) (30,0) (0 50) parce qu'on considère qu'un polygone se réfère à des entités (Points). Au moment de la construction, on s'intéresse au fait que le premier sommet est le point a, pas que ce sommet est en (0,0).

Dans ce cas, il s'agit d'un **sémantique d'entité**.

Les Points n'appartiennent pas au Polygone, ils sont en relation d'agrégation (association "Un point est un sommet d'un polygone").

3.3 Différences et Egalité

La différence porte en particulier sur la notion d'égalité (identité).

— En sémantique de valeur, deux Points sont égaux, à un moment donné, s'ils ont les mêmes coordonnées (ils contiennent les **mêmes champs**).
— En sémantique d'entité, deux Points sont identiques si et seulement s'il s'agit de la **même donnée**, à la même adresse en mémoire. Deux points peuvent être momentanément à la même position, ça n'en fait pas un même point.

3.4 Implémentation

— Dans le cas de la sémantique de valeur, la classe Polygone pourra être définie comme ceci (on suppose qu'il y a au plus 10 sommets)

```
1 class Polygone {
2 private:
3   Point m_sommets[10]; // valeurs
4   int m_nb_sommets;
5   ...
6 };
```

— et pour la sémantique d'entités, il faut se référer à des points, donc on stockera des pointeurs

```
1 class Polygone {
2 private:
3   Point * m_sommets[10];
4   int m_nb_sommets;
5   ...
6 };
```

4 Polygone de taille maximale fixe

Pour ce travail on s'intéresse à la seconde version (sémantique d'entité pour les points).

4.1 Préparation

Créez un projet à partir des classes Point, Polygone et du programme principal fournis.

4.2 Programmation : test_ajout()

1. Lancez le programme. Lisez le source de main.cpp pour imaginer ce que devrait faire `test_ajout()`.
2. Compilez, exécutez, et constatez que ça ne le fait pas.
   C'est normal, les entêtes et les déclarations des fonctions sont fournies de façon à ce que le compilateur accepte le source, mais elles ne font pas le travail. C'est ce qu'on appelle des "stubs" (voir article "Bouchon (informatique)" sur Wikipedia). C'est à vous de compléter pour que ça fonctionne, de préférence en suivant l'ordre des tests. C'est la technique de développement dite "développement piloté par les tests" (TDD, test driven development).
3. Complétez la fonction `Polygone::ajouterSommet`, qui note l'adresse de son paramètre (un point) dans le tableau `m_sommets` du polygone.
4. Complétez la fonction `Polygone::toString` qui doit retourner la description d'un Polygone (concaténer dans une chaine les représentations des Points contenus dans le tableau).
5. Exécutez et vérifiez que les affichages sont corrects.
6. Exécutez avec Valgrind : cet utilitaire permet d'indiquer une (ou plusieurs) éventuelle fuite mémoire (i.e. une donnée a été allouée et jamais libérée). Pour cela : onglet "Analyser" (sur la gauche), puis choisir Valgrind dans le menu déroulant en bas de la partie centrale). Vérifiez qu'il n'y a pas de fuite mémoire.

5 Polygone de taille maximale variable

On souhaite indiquer, dans le constructeur de Polygone, sa capacité (le nombre de sommets maximum qu'il peut avoir). Pour deux raisons :

— Parfois, la limite de 10 est insuffisante

— Parfois, elle est excessive (gaspillage de place).

Nous allons donc remplacer le tableau de taille 10 par un tableau alloué sur le tas, de la taille demandée. Nous allons voir que cela va entraîner des modifications en chaine.

5.1 Préparation

Dans la fonction main(), mettez en commentaire les appels à `test_copie` et `test_affectation`.

5.2 Modification du constructeur

1. Modifiez la déclaration de la capacité et du tableau de sommets :

```
1  int m_capacite; // devient une variable
2                  // membre
3  Point ** m_sommets; // pointeur de pointeur
```

2. Ajoutez un constructeur paramétré

```
1 Polygone::Polygone (int capacite) {
2   m_sommets = new Point* [capacite];
3   m_capacite = capacite;
4   m_nbSommets = 0;
5 }
```

3. Modifiez le constructeur par défaut pour qu'il renvoie sur le constructeur paramétré (délégation de constructeur)

```
1 Polygone::Polygone ()
2   : Polygone(CAPACITE) {}
```

4. Compilez et exécutez.
5. Exécutez avec Valgrind. Qu'observez vous ?

5.3 Ajout d'un destructeur

Il y a bien un destructeur implicite généré par le compilateur, mais son rôle se limite à détruire les données membres : les 2 entiers et le pointeur sur le tableau de pointeurs.

Mais détruire le pointeur, ce n'est pas suffisant. Avant cela, il faut désallouer le tableau pointé.

L'ajout d'un destructeur explicite va résoudre le problème.

1. Pour cela, ajoutez un destructeur dans la classe, qui libèrera le tableau.

```
1 class Polygone {
2 ...
3 ~Polygone();
4 ...
5 };
```

   et ce destructeur fera le `delete` nécessaire.

2. Vérifiez que cela fonctionne, tant à l'exécution que par Valgrind.

5.4 Constructeur de copie

Décommentez l'appel à la fonction `test_copie`, et faites exécuter. **Observez que**

— Le programme plante manifestement à la fin de l'exécution. En remontant, vous trouverez la cause `double free or corruption`.

— Valgrind vous le confirmera, et vous permettra de savoir que c'est arrivé dans le destructeur de Polygone.

Ce qui s'est passé
— Dans `test_copie`, le polygone p2 est créé en appelant le constructeur de copie.
— Le compilateur a généré un constructeur de copie implicite, qui crée une copie en copiant les champs de p1 dans ceux de p2.
— Dans la première version du programme, ça marchait bien. Le tableau de sommet était copié, et p2 repartait avec sa copie.
— Maintenant, ça copie le pointeur. Le pointeur `m_sommets` de p2 pointe donc sur le même tableau que celui de p1
— à la fin de `test_copie`, les variables allouées (a, p1, p2, b, c) sont détruites, en ordre inverse (c, puis b, p2, p1, a).
— quand p2 est détruit, le destructeur est appelé, qui libère le tableau.
— quand p1 est détruit, même chose, mais comme le tableau a déjà été libéré, ce qui provoque le crash.

la cause est une erreur de programmation : chaque polygone doit posséder, en propre, son tableau de sommets.

Le constructeur de copie implicite ne fait pas le boulot, il faut en écrire un qui marche. Quand on construit p2, il faut
— allouer un nouveau tableau de même capacité que celui de p1
— copier les (pointeurs de) sommets de p1 dans ceux de p2.
— copier aussi la capacité et le nombre de sommets.
Ecrire le constructeur `Polygone(const Polygone & autre_poly)`.

5.5 Redéfinition de l'affectation

Décommentez maintenant `test_affectation`. Vous constaterez que le programme se termine anormalement, avec une erreur du même type.

La raison est similaire :
— dans le `test_affectation`, il y a une affectation p2 = p1 ;
— le compilateur fournit une opération d'affectation implicite, qui consiste à affecter les champs de p2 dans ceux de p1.
— comme précédemment, cela conduit à copier le pointeur `m_sommets`, et donc partager le même tableau de sommets
— il y a donc deux problèmes au moment de la destruction : double libération de la donnée partagée, et non-libération de celle qui a été oubliée (le tableau qui appartenait à p2).
Le remède est assez semblable : au lieu de copier le pointeur, il faut copier les sommets d'un polygone dans l'autre.

Pour cela, dans la classe Polygone, il faut définir l'opérateur d'affectation (operator=)

```
1 Polygone & Polygone::operator= (const Polygone &
     autre) {
2   // capacite suffisante ?
3   assert (m_capacite >= autre.m_taille);
4
```

```
5     // copier la taille
6     // transferer les elements
7
8     // l'affectation retourne, par reference,
9     // l'objet destinataire
10    return *this;
11  }
```

Vérifiez que cela fonctionne (Run + Valgrind).

6 Bilan : The Rule Of Three

La pratique de C++ montre que 3 choses sont généralement liées : si on doit écrire un destructeur, il faut souvent écrire aussi un constructeur par copie, et l'affectation. Et inversement.

C'est ce qu'on appelle "La règle des trois" (Marshall Cline, 1991).

L'explication est simple :

— si le destructeur généré automatiquement pour une classe ne suffit pas, c'est qu'il y a une ressource R qu'il faut libérer explicitement quand un objet O de cette classe sont détruits.

— Si O a la responsabilité de libérer R, c'est probablement qu'il en est propriétaire.

— S'il en est propriétaire, il faut savoir ce qui se passe avec R quand on crée une copie de O, ou qu'on fait une affectation. En général, le constructeur de copie et l'affectation implicites ne conviennent pas.

En général on a besoin de le faire quand les objets contiennent des structures de données complexes, des pointeurs, des références à des ressources externes, etc.

En C++11 "moderne" on parle de la règle des 5, puisqu'au trio déjà cité s'ajoutent :

— le move constructor, qui sert à initialiser des variables temporaires modifiables

— le move assignment operator (dans certains cas, le destinataire d'une affectation peut s'arroger les ressources détenues par la valeur transférée).

Ces nouveautés ont été introduites en C++11 pour permettre une gestion plus fine et plus efficace de la mémoire. Elles dépassent le cadre de ce cours.