Semaine 3
Classes, objets et allocation dynamique

Contenu : Correction Polygone, allocation dynamique d'un objet, puis allocation dynamique au sein d'un objet...

Sources liées à ce TD :

- `Polygone_a_distribuer.ps` : pour les étudiants
- `Polygone_dyn_Profs.ps` : pour les enseignants
- `Executions_a_distribuer.ps` : pour les étudiants

1 Classe `Polygone` : correction et commentaires

Distribuer la correction `Polygone_a_distribuer.ps` et commenter bien sûr l'ensemble du code, mais en particulier les points suivants :

- **Résultat d'exécution :**
  cheminement des appels aux constructeurs, destructeurs (Polygone ET Point) et autres fonctionnalités.
- **static :** membres statiques dans une classe (dans Point et Polygone, 2 exemples différents).
  Le mot clé **static** signifie que la variable n'existe qu'en un seul exemplaire, elle est globale à la classe en quelque sorte. Autrement, chaque objet du type de la classe dispose de sa propre copie.
  Pour initialiser ce type de variable : si c'est une constante entière, on peut le faire dans la classe (ex : Polygone), sinon, si pas entière ou pas constante, ça se fait dans le fichier .cc de la classe (ex : Point, sauf que là, comme c'est une constante, on aurait pu aussi le faire dans la classe). (Infos issues de la faq C++ de developpez.com).
- **assert :** ici utilisé dans ajoutSommet et getPoint.
  Voici quelques explications sur la fonction assert. Elle prend en paramètre un booléen. On doit considérer cela comme une "supposition" : si le booléen est vrai, le programme continue normalement son exécution. Sinon, le programme s'arrête et la ligne à laquelle il a stoppé est affichée avec un message d'erreur.
  Cela permet de déboguer plus facilement. Signaler qu'il existe d'autres mécanismes plus "professionnels", comme les exceptions, qui seront vues plus tard, en Semestre 3 en Java.
- **surcharge du flux « :**
  Commenter + ou - longtemps, selon si fait ou pas sur Complexe au précédent TD (cf fin feuille TD02).
- **Prototype méthode getPoint :** (un peu compliqué, ne pas trop insister... voir en fonction du public !)
  Si `Point getPoint(int idx) const` : appel du constructeur par copie à chaque retour de la fonction.

Si `const Point & getPoint(int idx) const` : évite l'appel au constructeur par copie en retournant une référence constante sur un Point ; c'est OK car c'est un Point qui existe dans le tableau contenu dans l'objet Polygone appelant, et non pas un Point créé dans la fonction getPoint et qui serait détruit à la fin de la fonction.

2 Allocation dynamique d'un objet

Remarque : pas d'idée d'exercice particulièrement intelligent ou pédagogique pour cette section... Agrémenter si besoin de plus d'exemples et surtout de schémas au tableau.

2.1 Instanciation de classe ou objet

Un objet est une instance d'une classe (i.e., une variable dont le type est une classe). Un objet est donc la réalisation effective d'une classe. Un objet occupe de l'espace en mémoire. Il peut être alloué :

statiquement : c'est ce qu'on a vu pour l'instant. Comme les variables de type de base, on écrit le nom de la classe (i.e. le type) suivi du nom que l'on veut donner à l'objet (i.e. le nom de la variable), éventuellement suivi par les arguments d'appels données à un constructeur de classe. Ex :

```
Point pt;
Point pt2(3,7);
Point tab[10];
```

Comme toute variable déclarée statiquement, la durée de vie d'un objet est limitée au contexte dans lequel il a été déclaré, en général le corps d'une fonction ou méthode.

dynamiquement : cela est effectué par l'intermédiaire des opérateurs `new` et `delete`. Un pointeur sur la zone mémoire du tas où l'objet a été alloué est retourné. L'objet déclaré dynamiquement est persistant. Lorsqu'on n'en a plus besoin, on le désalloue en donnant le pointeur sur cette zone allouée à l'opérateur `delete`. Ex :

```
// Le constructeur a 2 parametres de l'objet est appele
Point* ptr_pt = new Point(3,7);
ptr_pt->deplace(1,1);  // Modifie abs et ord de l'objet pointe
delete ptr_pt;  // L'objet pointe est désalloue (destructeur est appele)
```

2.2 Accès aux attributs ou méthodes d'un objet

Etant donnée une instance d'un objet, on accède à ses attributs et à ses méthodes grâce à la notation pointée ".", dans la limite de visibilité bien sûr (public/private). Pour les pointeurs, cela se fait au travers de la notation fléchée "->", comme dans l'exemple ci-dessus pour l'appel à la méthode `deplace`.

3 Allocation dynamique au sein d'un objet

Toutes les classes que nous avons étudiées pour l'instant (Point, Rationnel, Complexe, Polygone) ont pour attributs des objets alloués statiquement. Ainsi, la destruction d'une

instance de ces classes entrainent automatiquement la destruction des objets qui la composent, en appelant leur destructeur si besoin.
Nous avons pu constater ce mécanisme grâce aux affichages ajoutés dans les destructeurs de Point et Polygone (cf section 1).

Si un ou plusieurs attributs d'une classe sont alloués et gérés dynamiquement dans plusieurs méthodes, il faudra bien que lors de la desctruction de l'objet, ces attributs soient désalloués explicitement !... D'où le rôle primordial du destructeur dans ce contexte ! C'est lui qui devra se charger en général de ces désallocations...

3.1 Classe Polygone de taille dynamique

Illustrons cela en proposant de modifer la classe Polygone pour qu'elle implémente un polygone de taille dynamique, c'est-à-dire dont la taille n'est pas bornée par une constante MAX.

En terme de fonctionnement, dans un premier temps, on se donne les règles suivantes :

- par défaut, un polygone sera créé vide (sans point),

- la taille du tableau de points représentant le polygone suivra exactement la taille du polygone, c'est-à-dire que ce tableau sera redimensionné à chaque ajout de point au polygone.

Exercice 14 : Pouvez-vous deviner le contenu de l'interface (.h) de la classe Polygone ?
Réponse : Seule la partie privée change, les prototypes des méthodes sont les mêmes.

Fichier `.h`

```
class Polygone
{
  private :
    Point * my_tab; // tableau qui sera alloué dynamiquement
    int my_taille;
  public :
    ...
};
```

Exercice 15 : Identification du code à modifier.

1. Quelles que soient les modifications que nous allons apporter à la classe Polygone, doit-on modifier le main (TestePolygone) qui nous permet de tester nos fonctions ?
   Réponse : non, car l'interface publique va rester la même, avec les mêmes fonctionnalités.
2. Quelles sont les méthodes de la classe qui devront être modifiées ? (les autres resteront telles quelles)
   Réponse : constructeur par défaut, destructeur, constructeur par copie, opérateur d'affectation, saisie, ajoutSommet, litFichier.

Exercice 16 : Ecrire les méthodes modifiées.
(En fonction du temps, choisir l'ordre pédagogique qui semble opportun, peut-être ne pas traiter toutes les méthodes, choisir les plus représentatives)

Réponse :

```
Polygone::Polygone()
{
  cout << "Polygone::Constructeur par defaut" << endl;
  // polygone vide
  my_taille = 0;
  my_tab = NULL;
}

Polygone::~Polygone()
{
  cout << "Polygone::Destructeur" << endl;
  if( my_tab!=NULL )  // Attention, si my_tab était vide, rien à libérer
    delete [] my_tab;
}

Polygone::Polygone( const Polygone & poly )
{
  cout << "Polygone::Constructeur par copie" << endl;
  my_taille = poly.my_taille;
  my_tab = new Point[my_taille];
  for(int i=0; i<my_taille; i++)
    my_tab[i] = poly.my_tab[i];
}

Polygone &
Polygone::operator=( const Polygone & poly )
{
  cout << "Polygone::Operateur affectation" << endl;
  if( this != &poly )
  {
    if( my_tab!=NULL ) // Attention, si my_tab était vide, rien à libérer
      delete [] my_tab;

    my_taille = poly.my_taille;
    my_tab = new Point[my_taille];
    for(int i=0; i<my_taille; i++)
      my_tab[i] = poly.my_tab[i];
  }
  return *this;
}
```

```
void
Polygone::saisie()
{
  int taille_reelle;
  float valx, valy;

  my_taille=0;

  do {
    cout << "nb de sommets : ";
    cin >> taille_reelle;
  } while (taille_reelle <0);

  for (int i=0; i<taille_reelle; i++) {
    cout << i << " : abs ? ";
    cin >> valx;
    cout << i << " : ord ? ";
    cin >> valy;
    ajoutSommet( Point(valx,valy) );
    // l'incrémentation de my_taille est réalisée
    // dans la fonction ajoutSommet
  }
}

void
Polygone::ajoutSommet (const Point & p)
{
  my_taille++;
  Point * tmp = new Point[my_taille];
  for(int i=0; i<my_taille-1; i++)
    tmp[i] = my_tab[i];
  tmp[my_taille-1] = p;

  if( my_tab!=NULL ) // Attention, si my_tab était vide, rien à libérer
    delete [] my_tab;

  my_tab = tmp;
}

// Le fichier peut contenir autant de Point qu'on veut maintenant
// On vide le polygone si besoin
void
Polygone::litFichier( string nom_fic )
{
  ...
  // polygone vide au depart
  my_taille = 0;
```

```
  if( my_tab!=NULL )
    delete [] my_tab;
  // pas nécessaire si appel à la fonction
  // ajoutSommet

  // lecture idem qu'avant
  ...
}
```

Exercice 17 : Exécution avec ce nouveau code.

Question : d'après vous, que se passe-t-il à l'exécution de `TestePolygone.cc` si, dans la classe `Polygone` dynamique, le programmeur oublie d'écrire le constructeur par copie et de l'opérateur d'affectation ?

Distribuer et commenter `Executions_a_distribuer.ps`. Faire remarquer que l'écriture du constructeur par copie et de l'opérateur d'affectation n'est plus optionnel ! C'est primordial.
En effet, la copie ou l'affectation membre à membre faite par défaut fait que les tableaux des polygones pointent sur la même zone mémoire ! Donc, conséquences :

- modifier un polygone modifie l'autre par effet de bord...
- au moment de la désallocation, on tente de désallouer plusieurs fois la même zone mémoire... Explosion du programme !

3.2 Classe Polygone de taille dynamique améliorée

Problème de complexité lié au redimensionnement systématique dans ajoutSommet.

==> leur faire prendre conscience du nombre d'opérations d'allocation/désallocation réalisées par exemple par l'ajout successif de 5 points à un polygone...

==> Les faire réfléchir à une meilleure solution.
Par exemple allocation par bloc d'une certaine taille, et réallouer que quand c'est nécessaire, c'est-à-dire qu'on dépasse la taille du tableau actuel.