**Semaine 2**
**Introduction aux classes - 2/2**

Contenu : constructeur de copie, opérateur d'affectation, et les autres opérateurs (sur la classe complexe).

Sources liées à ce TD :
- `Complexe_a_distribuer.ps` : pour les étudiants
- `Complexe_profs.ps` : compléments pour les enseignants

# 1 La classe `Complexe` (TP1 et travail personnel)

"Corriger" brièvement le travail sur la classe Complexe. En ramasser 2 ou 3 ? Avant de distribuer une version de base commune à tous, remarquer que l'on peut suivre le même cheminement que pour les rationnels, sauf : pas de complexe réduit, en revanche une fonction module et une pour le conjugué. On peut faire écrire au tableau le produit de deux complexes et le conjugué d'un complexe. Si quelqu'un sait écrire l'inverse, ok, sinon ils verront sur le corrigé :
distribuer et commenter `Complexe_a_distribuer.ps`.

# 2 Constructeur de copie

Nous avons vu précédemment que le passage par valeur d'un objet de la classe `Point`, ou `Complexe` ici, provoquait une recopie de l'objet dans une instance locale de l'objet. Or, dans notre classe, nous n'avons pas défini comment doit se faire cette recopie. Par défaut, elle se fait membre à membre, ici, cela nous convient bien. Cependant, très bientôt, nous serons amenés à faire de l'allocation dynamique de mémoire, et dans ces cas-là, ce fonctionnement peut s'avérer dramatique (partage d'un même espace mémoire par deux instances différentes).
Ainsi, pour programmer correctement, il faut définir un **constructeur de copie**.

**Exercice 7 :** Pouvez-vous devinez le prototype du constructeur de copie ? Ecrivez aussi le corps de cette méthode.
*Réponse :*

Fichier `.h`

```
class Complexe
{
  private :
    ...
  public :
    ...
    Complexe( const Complexe & z );
};
```

Fichier .cc

```
Complexe::Complexe( const Complexe & z )
{
   cout << "Complexe::constructeur de copie" << endl;
   m_reel = z.m_reel;
   m_img = z.m_img;
}
```

Le constructeur de copie est également appelé lors de la déclaration suivante pour `z` :

```
Complexe z1( 4, 5 );
Complexe z = z1;
```

et aussi lors du retour d'un complexe dans une fonction, comme nous le verrons plus loin, ou comme par exemple dans la fonction `additionner`, si on retourne un complexe plutôt que de le passer par référence :

```
Complexe additionner(const Complexe & z) const;
```

## 3 L'opérateur d'affectation =

Cet opérateur est défini implicitement par le compilateur pour toutes les classes et structures (comme le constructeur par défaut ou le constructeur de copie). Pour une classe `X`, il est appelé à chaque fois que l'on écrit une affectation avec un objet de la classe `X` dans la partie gauche de l'affectation. Ainsi,

```
X obj; // appel du constructeur par defaut de la classe X
X obj2( "chaine qcq" ); // appel du constructeur de X prenant une chaine.
obj = obj2; // appel de l'operateur d'affectation de X.
X obj3 = obj2; // appel du constructeur par copie de X.
X obj3( obj2 ); // equivalent
X obj3 = X( obj2 ); // equivalent
```

Son prototype est de la forme : `X & operator=( const X & autre ) ;`

Pour la classe `Complexe`, cela donne :

Fichier .h

```
class Complexe
{
  private :
    ...
  public :
    ...
    Complexe & operator=( const Complexe & z );
};
```

Fichier .cc

```
Complexe &
Complexe::operator=( const Complexe & z )
{
  cout << "Complexe::Operateur affectation" << endl;
  if ( this != &z ) {
    m_reel = z.m_reel;
    m_img = z.m_img;
  }
  return *this;
}
```

Afin d'améliorer la compréhension de la mise en oeuvre de l'opérateur d'affectation, tracez aux tableaux l'exécution des instructions ci-dessous (on peut utiliser une représentation graphique où chaque objet est représenté par une référence et 2 cases mémoire (partie imaginaire, partie réelle)) :

```
Complexe a,b;
Complexe c(2,2);

1. a = c;
2. a = b = c;
```

Pour la première instruction c'est l'opérateur d'affectation de l'objet $a$ qui est appelé avec pour paramètre $c$. Les valeurs des parties réelles et imaginaires de $c$ sont affectées aux parties réelles et imaginaires de $a$. Le retour de la valeur courante à la fin de la méthode (return *this) est inutile dans ce cas.
Pour la deuxième instruction, c'est l'opérateur d'affectation de l'objet $b$ qui est d'abord appelé avec pour paramètre $c$. Les valeurs des parties réelles et imaginaires de $c$ sont affectées aux parties réelles et imaginaires de $b$. Le retour de méthode de ce premier appel est l'objet b modifié. Dans un deuxième temps, l'opérateur d'affectation de l'objet $a$ est appelé avec pour paramètre $b$ qui a été retourné après le premier appel.

Chaque fois que vous définissez un constructeur par copie pour une classe, il faudrait théoriquement toujours fournir l'opérateur d'affectation de cette classe.

Maintenant que nous disposons d'un constructeur par copie et d'un opérateur d'affectation, nous pouvons sans danger écrire des fonctions qui retournent un Complexe.

**Exercice 8 :** Ecrivez l'addition de deux complexes sous forme d'une fonction retournant un Complexe :

*Réponse :*

```
Complexe Complexe::somme(const Complexe & z) const{
  Complexe z1(m_reel+z.m_reel, m_img+z.m_img);
  return z1;
}
```

**Remarque :** A la fin de la fonction le constructeur par copie est automatiquement appelé pour récupérer le retour. La variable locale z1 est ensuite détruite. On ne peut donc

absolument pas écrire

```
Complexe & Complexe::somme(const Complexe & z)
```

car on retourne une référence sur du vide dans ce cas... Le mécanisme des pointeurs peut éviter les copies lors du retour du résultat de fonctions, on peut en faire la remarque.

**Utilisation :**

```
Complexe z1(1.2, -3), z2(sqrt(2), sqrt(3)) ;
Complexe z3 = z1.somme(z2); // z3 construit par copie
// z3 vaut 1.61 - 1.27i
z2 = z3.somme(z1); // affectation
// z2 vaut maintenant 2.81 - 4.27i
```

**Exercice 9 :** Même exercice pour la fonction inverse ;

*Réponse :*

```
Complexe Complexe::inverse() const{
// 1/(a + ib) = a/(a^2 + b^2) - b/(a^2 + b^2)
  float module_carre = pow(m_reel,2) + pow(m_img,2);
  Complexe inv (m_reel/module_carre, -m_img/module_carre);
  return inv;
}
```

Remarque : on aurait pu le faire dès le départ, puisque nous faisons ici de la copie membre à membre.

# 4 Opérateurs et surcharge

**Exercice 10 :** On a écrit des fonctions d'égalité (points, rationnels, complexes ; leur utilisation : `if (x.egal(y)) ...`, on aimerait bien écrire plutôt : `if (x == y) ...`
Pour cela, il faut surcharger l'opérateur de comparaison : la fonction à surcharger s'appelle `operator==`.

Fichier `.h`

```
class Complexe
{
  private :
    ...
  public :
    ...
    bool operator==( const Complexe & z ) const;
};
```

Fichier .cc

```
...
bool
Complexe::operator==( const Complexe & z ) const
{
  return m_reel == z.m_reel && m_img == z.m_img;
}
```

**Exercice 11 :** Surcharger les opérateurs : +, *

*Réponse :*

```
Complexe Complexe::operator+(const Complexe & autre) const{
  Complexe somme(m_reel + autre.m_reel,m_img + autre.m_img);

  return somme;
}

Complexe Complexe::operator*(const Complexe & autre) const{
  Complexe produit(m_reel * autre.m_reel - m_img * autre.m_img,
           m_reel * autre.m_img + m_img * autre.m_reel);

  return produit;
}
```

## 5 Surcharge de fonctions non membres

La surcharge des fonctions n'est pas réservée aux classes. Il peut être utile de surcharger des fonctions en dehors de la classe. Par exemple nous n'avons pas surchargé les opérateurs de division et de soustraction ; nous pourrions souhaiter écrire

```
z1 = z2 / z3;
```

Une solution serait évidemment de surcharger ces opérateurs dans la classe. Mais c'est ici l'occasion de réfléchir sur un point très important ; dans une situation d'apprentissage (la vôtre en ce moment), on est souvent à la fois le concepteur et l'utilisateur de la classe. Mais dans la pratique réelle, ces deux rôles sont souvent disjoints. On utilise par exemple une classe sans pouvoir en aucune façon la modifier (exemple : classe string). Donc nous allons faire comme si nous ne pouvions pas modifier la classe Complexe.

**Exercice 12 :** Surchargez les opérateurs / et -

```
// Attention, fonctions NON membres
// dans le fichier essaicomplexe.cc
Complexe operator/(const Complexe & z1, const Complexe & z2){
  return z1 * z2.inverse();
}
```

```
Complexe operator-(const Complexe & z1, const Complexe & z2){
  // pas de get, il faut utiliser les services proposés par la classe
  Complexe difference;
  z1.soustraction(z2, difference);
  return difference;
}
```

S'il existe une version fonctionnelle de la soustraction :

```
Complexe operator-(const Complexe & z1, const Complexe & z2){
  // pas de get, il faut utiliser les services proposés par la classe

  return z1.soustraction(z2);
}
```

**Exercice 13 :** Surcharge des flux d'entrée-sortie (<<, >>)

On aimerait bien aussi écrire

```
cout << z1 << " + " << z2 << " = " << z1+z2 ;
```

Il est donc nécessaire de surcharger l'opérateur <<. Par contre, cet opérateur n'est pas un opérateur de la classe Complexe. en effet, c'est l'opérateur << du flux cout qui prend en paramètre un complexe (z1) et non l'inverse.

Nous allons donc définir un opérateur de sérialisation non-membre (mais néanmoins amie (*friend*) [1]) associé à la classe Complexe.

*Réponse :*

```
// dans le fichier essaicomplexe.cc si l'on est utilisateur de la classe ;
// mais en tant que concepteur de la classe, on peut le mettre dans Complexe.h
// (après la définition de la classe) et dans Complexe.cc


ostream & operator<<(ostream & out, const Complexe & z){
  out << z.toString();
  return out;
}
```

[1] La fonction peut accéder à la partie privée de la déclaration de la classe