Université de Nice-Sophia Antipolis
Polytech ELEC4

Lundi 23 septembre 2019
Durée: 2h

C++ Classe et Objet
Travaux Dirigés – Séance n. 3

---

1 Objectif

Dans les deux premiers TD, nous avons manipulé des objets définis dans l'environnement du langage C++ comme, par exemple, `string` ou `vector`. Dans ce TD, nous allons apprendre à déclarer des classes et les utiliser pour créer des objets.

2 Introduction

Dans le paradigme objet, un objet contient des données et des actions (méthodes). Un objet est une notion *dynamique*, c'est-à-dire à l'exécution du programme. Dans le modèle objet, un programme à l'exécution est *un ensemble d'objets dynamiques qui sont en interaction.*

C++ est un langage de *classe*, c'est-à-dire, que de façon *statique* (*i.e.* à la compilation), les objets sont définis par des classes qui les décrivent. En C++, une classe définit un *type*, et la déclaration d'une variable permettra de désigner des objets de ce type.

Dans le suite de ce TD, nous allons définir une classe `complexe` permettant de représenter des nombres complexes.

3 La classe complexe

3.1 Déclaration

D'une façon générale, la déclaration de la classe sera placée dans un fichier `.hpp` et le code des méthodes associées dans un fichier `.cpp`.

Un objet complexe est constitué de données : un partie réelle et une partie imaginaire, toutes deux de type réel. On peut donc commencer la déclaration de notre classe complexe :

```
class complexe {
 private:
  double preelle;
  double pimg;
};
```

La déclaration de la classe est introduite par le mot-clé **class** suivi de son nom. Elle définit deux données membres (ou attributs) `preellle` et `pimg` de type double. Le mot-clé **private** spécifie que ces deux données sont privées, *i.e.* uniquement visible dans la classe. Il existe également les mots-clés **public** et **protected** pour rendre une visibilité globale ou pour les classes héritières. Nous verrons la notion d'héritage ultérieurement. Les constructeurs et les méthodes que vous définirez dans la suite de ce TD doivent être déclarés **public**.

exercice 1) Recopiez la classe `complexe` précédente dans un fichier `complexe.hpp` et placée sous la directive `pragma` :


```
#pragma once

// déclaration de la classe complexe
```

La directive `pragma` garantit que le fichier ne pourra être inclus plusieurs fois.

exercice 2) Dans le fichier `testcomplexe.cpp`, incluez le fichier `complexe.hpp` et écrivez une fonction `main` dans laquelle vous déclarez un variable `c1` de type `complexe`. Compilez et exécutez votre programme.

3.2 L'objet courant : this

En programmation objet, les méthodes s'appliquent (généralement) sur les données d'un objet, appelé *l'objet courant*. En C++, il est désigné par le mot-clé **this** qui est un *pointeur*. Ainsi, l'accès à la variable membre `v` de l'objet courant s'écrira **this->v**.

3.3 Constructeur

Un constructeur est une fonction membre qui sert à l'*initialisation* des données membres d'un objet. Le constructeur porte le nom de la classe, et on *n'indique pas* de type de retour dans son en-tête.

Une classe sans constructeur possède un constructeur sans paramètre, dit *par défaut*, qui initialise les données membres de l'objet à 0 (ou convertibles en 0). À partir du moment où un constructeur est défini, le constructeur par défaut disparaît. Si on souhaite un constructeur qui initialise les données à 0, il faut alors le déclarer explicitement.

Si on veut déclarer des variables de type `complexe` initialisées à des valeurs de parties réelle et imaginaire particulières, il faudra ajouter un constructeur qui initialise les deux membres `preelle` et `pimg` :

```
complexe(const double r, const double i) {
    this->preelle = r;
    this->pimg = i;
}
```

exercice 3) Ajoutez le constructeur *public* précédent à votre classe, et modifiez la déclaration de la variable `c1` pour l'initialiser à la valeur du complexe $(2.1, -6.7)$. Testez votre programme.

exercice 4) Déclarez une variable `c2` de type `complexe` non initialisée. Compilez. Que constatez-vous ?

exercice 5) Ajoutez à votre classe `complexe`, un constructeur qui initialise à 0.0 les parties réelle et imaginaire.

Une autre façon de procéder est d'utiliser la notion de valeur par défaut à associer à la donnée membre si aucune valeur ne lui est transmise lors de l'appel constructeur. C'est la notion de *liste d'initialisation*. Ainsi on écrira :

```
complexe(const double r=0, const double i=0) : preelle(r), pimg(i) {}
```

Dans l'appel du constructeur, les paramètres effectifs absents seront remplacés par les valeurs par défaut spécifiées.

exercice 6) Dans votre classe `complexe`, remplacez les deux constructeurs par le constructeur précédent. Dans la méthode `main`, écrivez les 3 déclarations de variables `c1`, `c2` et `c3`.

```
complexe c1 = complexe();
complexe c2 = complexe(4.5);
complexe c3 = complexe(3.2, 8.9);
```

C++ propose également une autre syntaxe pour déclarer des variables et initialiser les objets. Les trois déclarations précédentes peut être récrites de la façon suivante :

```
complexe c1;
complexe c2(4.5);
complexe c3(3.2, 8.9);
```

exercice 7) Testez ces trois nouvelles déclarations.

3.4 Destructeur

Le *destructeur* est en quelque sorte le symétrique du constructeur. Le destructeur possède le nom de la classe précédé par le symbole `~`. Il n'a pas de paramètre et ne renvoie rien. Par exemple, pour la classe `complexe`, le destructeur associé est :

```
~complexe()
```

Si vous ne définissez pas de destructeur dans votre classe, le système en définit un pour vous automatiquement, mais qui ne fait *rien*.

Le rôle du destructeur est de libérer les ressources utilisées par l'objet. Le destructeur est automatiquement appelé en sortie de fonction lors de la destruction des variables locales, ou en fin de programme lors de la destruction des variables globales.

Nous verrons dans les prochains TD, l'importance des destructeurs, en particulier dans le cas de l'allocation dynamique. Mais, c'est une autre histoire. Dans le cas présent de notre classe `complexe`, il est sans importance.

exercice 8) Toutefois, afin de mettre en évidence cette notion, ajoutez à votre classe `complexe` son destructeur qui écrit sur la sortie standard *je suis le destructeur de complexes !*.

exercice 9) Recompilez et testez votre programme.

4 Méthodes

On va ajouter à la classe `complexe` 4 méthodes publiques `getPreelle`, `getPimg`, `setPimg` et `setPreelle`, pour accéder et modifier les données membres. Par exemple, pour la partie réelle, on écrira dans le fichier `complexe.hpp` les prototypes des méthodes :

```
class complexe {

private:
  double preelle;
  double pimg;

public:
  double getPreelle() const;
  void setPreelle(const double x);
};
```

et le code de ces méthodes dans le fichier `complexe.cpp` :


```
/*
 * Rôle : renvoie la partie réelle du complexe courant
 */
double complexe::getPreelle() const {
  return this->preelle;
}
/*
 * Rôle : affecte r à la partie réelle du complexe courant
 */
void complexe::setPreelle(const double r) {
  this->preelle = r;
}
```

Notez que les méthodes pourrait être définies dans la classe `complexe`, mais en général elle le sont à l'extérieur, et l'opérateur `::` permet d'indiquer à quelle classe appartient la méthode. Le mot-clé **const** indique que la méthode ne modifie par l'objet.

exercice 10) Ajoutez à votre classe `complexe` les deux méthodes précédentes et écrivez, selon le même modèle, les méthodes `getPimg` `setPimg`.

exercice 11) Dans le `main`, affichez les parties réelles et imaginaires de vos complexes `c1`, `c2` et `c3`.

---

exercice 12) Ajoutez la méthode `ecrireComplexe`, qui écrit sur la sortie standard le complexe courant sous la forme *(r,i)*.

exercice 13) Testez votre méthode pour afficher les valeurs de `c1`, `c2` et `c3`.

On veut pouvoir réaliser les opérations mathématiques standard sur les complexes.

Commençons par programmer les méthodes de conversion de la représentation polaire vers la représentation cartésienne.

Rappel : Tout complexe c admet une représentation cartésienne $x + iy$ et polaire ρe^{θ} où ρ est la norme et θ la phase de c. Le passage d'un système de coordonnées à l'autre se fait à l'aide des formules de conversion :

coordonnées polaires	coordonnées cartésiennes
$\rho = \sqrt{x^2 + y^2}$	$x = \rho\cos(\theta)$
$\theta = arctan(y/x)$	$y = \rho\sin(\theta)$

exercice 14) Ajoutez la fonction `rho` qui renvoie la norme de l'objet courant à la classe `complexe`.

exercice 15) Écrivez la fonction `theta` qui renvoie l'argument d'un complexe. Attention, la fonction `atan` est définie de $\mathbb{R}$ vers $]-\frac{\pi}{2}, \frac{\pi}{2}[$. Utilisez La fonction `atan2` qui règle ce problème (inclure `cmath`).

exercice 16) Testez les deux fonctions `rho` et `theta`.

exercice 17) Écrivez la fonction `polComplexe` qui possède deux paramètres de type réel représentant le module et l'argument d'un complexe polaire, et qui renvoie un objet de type complexe en coordonnées cartésiennes. Remarquez que cette fonction ne dépend pas de l'objet courant (*i.e.* elle ne se sert pas de `preelle` et `pimg`). Cette méthode sera alors être déclarée **static** dans la classe `complexe`. Son en-tête est le suivant :

```
static complexe polComplexe(const double rho, const double theta)
```

Notez que le mot-clé **static** ne devra pas apparaître dans l'en-tête de la méthode dans le fichier `complexe.cpp` (uniquement dans `complexe.hpp`).

Puisque cette fonction ne dépend pas d'un objet particulier, on pourra l'appeler en la préfixant par le nom de la classe : `complexe::polComplexe(r,t)`.

exercice 18) Ajoutez les fonctions publiques qui effectuent la somme, la soustraction et le produit de deux nombres complexes. Ces fonctions ont les en-têtes suivants :

```
complexe plus(const complexe &c) const;
complexe moins(const complexe &c) const;
complexe mult(const complexe &c) const;
complexe div(const complexe &c) const;
```

Notez que le produit de deux complexes est plus simple à écrire en utilisant les coordonnées polaires :

$$\begin{aligned} \rho(z1 \times z2) &= \rho(z1) \times \rho(z2) \\ \theta(z1 \times z2) &= \theta(z1) + \theta(z2) \end{aligned}$$

De même, pour la division :

$$\begin{aligned} \rho(z1/z2) &= \rho(z1)/\rho(z2) \\ \theta(z1/z2) &= \theta(z1) - \theta(z2) \end{aligned}$$

Pensez à utiliser la méthode `polComplexe` pour le produit et la division.

exercice 19) Définissez les deux méthodes :

- **bool** `egal(const complexe &)` **const**
- **bool** `different(const complexe &)` **const**

qui testent l'égalité et la différence entre l'objet courant et le complexe transmis en paramètre. On prendra soin de traiter le problème posé par l'opérateur `==` dû à l'inexactitude des calculs sur les réels.

exercice 20) Déclarez la constante complexe `I` qui représente le complexe $(0, 1)$.

---

5 Surcharge d'opérateur

Dans la première feuille de TD, nous avons vu que C++ offrait la possibilité de surcharger des fonctions. Il permet aussi de surcharger des opérateurs. Il est clair qu'il sera plus agréable d'écrire `c1 + c2` pour additionner les deux complexes `c1` et `c2`, plutôt que d'écrire `c1.plus(c2)`.

Un opérateur surchargé suivra la même syntaxe que l'opérateur qu'il surcharge, et possédera le même nombre d'opérandes et conserve ses règles de priorité et d'associativité.

La syntaxe pour surcharger un opérateur (ici `+`) d'une classe `C` est la suivante :

```
C C::operator+(C){ /*...*/} const;
```

Pour la somme de deux complexes, l'en-tête de la surcharge de l'opérateur `+` dans la classe `complexe` s'écrira :

```
complexe complexe::operator+(const complexe &c) const;
```

exercice 21) Écrivez les surcharges des opérateurs `+`, `-`, `*`, `/`, `==` et `!=`.