TP 10 : Classes Complexes

notions : Constructeurs, accesseurs, mutateurs, membres et amies

---

Un complexe c a deux représentations :

1. une représentation cartésienne avec 2 réels représentant la partie réelle re et la partie imaginaire im. Il s'écrit alors $c = re + i * im$
2. une représentation polaire avec 2 réels représentant le module et l'angle ρ ou argument θ. Il s'écrit alors $c = \rho.exp^{(i*\theta)}$

Les relations qui existent entre les 2 représentations sont les suivantes :

$$\rho = \sqrt{re^2 + im^2} \qquad \theta = atan2(im, re) \qquad re = \rho.cos(\theta) \qquad im = \rho.sin(\theta) \tag{1}$$

1 La classe MesComplexes et sa représentation algébrique

Dans un fichier header mescomplexes.h et un fichier source mescomplexes.cc, écrire une classe `MesComplexes` possédant les deux attributs **privés** réels `re` et `im` représentant les parties réelles et imaginaires.

Vous implanterez ensuite progressivement les méthodes et fonctions suivantes, en les testant au fur et à mesure de leur implantation. Un programme de test est fourni sur le site, que vous pouvez modifier au fur et à mesure.

Attention : dans les prototypes de ces méthodes et fonctions, ajoutez le mot clé `const` à tous les endroits appropriés : méthodes `const`, paramètres `const`...

- Les constructeurs et destructeurs (pour bien suivre ce qui se passe durant les tests, affichez quelque chose dans le terminal dans chacun des constructeurs et dans le destructeur) :
  - `MesComplexes()`, constructeur par défaut,
  - `MesComplexes(double re, double im = 0)`, constructeur qui permet d'initialiser un complexe soit à partir de deux réels passés en paramètres pour créer $re + im * i$, soit avec uniquement la partie réelle pour créer $re + 0 * i$.
  - `MesComplexes(const MesComplexes& other)`, constructeur de copie initialisant un complexe à partir d'un autre complexe
  - `~MesComplexes()`, le destructeur,
- la méthode d'affichage
  - `void afficher()` qui affiche l'état du complexe dans le Terminal
- les accesseurs
  - `double getReelle()` qui renvoie la partie réelle d'un complexe
  - `double getImaginaire()` qui renvoie la partie imaginaire d'un complexe
  - `double getRho()` qui renvoie le module d'un complexe
  - `double getTheta()` qui renvoie l'argument d'un complexe
- les mutateurs privés :
  - `void setReelle(double re)` qui modifie l'objet sur lequel cette méthode est appelée pour que sa partie réelle prenne la valeur du double `x` passé en paramètre.
  - `void setImaginaire(double im)` qui modifie l'objet sur lequel cette méthode est appelée pour que sa partie imaginaire prenne la valeur du double `x` passé en paramètre.

  - `void setRho(double r)` qui modifie l'objet sur lequel cette méthode est appelée pour que son module prenne la valeur du double `x` passé en paramètre. Faites attention à l'aspect séquentiel des instructions utilisées. Pensez que si vous modifiez la partie réelle, la valeur de l'argument est elle aussi modifiée. Si vous avez besoin de l'argument du complexe avant modification, il n'est plus accessible, sauf si vous l'avez conservé par ailleurs.
  - `void setTheta(double t)` qui modifie l'objet sur lequel cette méthode est appelée pour que son argument prenne la valeur du double `x` passé en paramètre. Faites attention à l'aspect séquentiel des instructions utilisées.
- les méthodes publiques :
  - `double costa()` qui retourne le produit de la partie réelle par la partie imaginaire (qui est la fonction d'erreur d'un comparateur de phase d'une PLL pour la récupération de porteuse d'un signal modulé à 2 états de phase)
  - `void add( MesComplexes & other)` qui ajoute au complexe `this` le complexe `other`. Cette méthode modifie l'objet sur lequel elle est appelée.
- l'opérateur membre :
  - `complexe & operator+= (const complexe& other)` qui réalise "l'addition en place" $this = this + other$
  - *bien sûr, dans la vraie vie, on complèterait par des opérateurs en place* `-=`, `*=`, *etc.*
- les fonctions amies :
  - `friend ostream& operator<<(ostream& out, const MesComplexes& a)` qui ajoute une représentation textuelle de `a` dans le flux `out` et retourne ce flux.
  - `friend MesComplexes operator+(const MesComplexes& a, const MesComplexes& b)` qui réalise la somme des complexes a et b. Notez qu'il est plus facile de faire la somme de 2 nombres complexes en utilisant la représentation rectangulaire.
  - `friend MesComplexes operator*(const MesComplexes& a, const MesComplexes& b)` qui réalise le produit des complexes a et b. Notez qu'il est plus facile de faire le produit de 2 nombres complexes en utilisant la représentation polaire.
  - `friend MesComplexes pow (const MesComplexes& a, int n)` qui calcule l'élévation à la puissance du complexe. Notez qu'il n'existe pas d'opérateur puissance en C. On pourrait utiliser l'opérateur ^, mais on changerait la sémantique habituelle de celui-ci.

Nous fournissons un programme de test :

```
main() {
    MesComplexes x1(1,10), x2(4,2), x3(x2),x4;
    x1.afficher();       x2.afficher();       x3.afficher();
    cout << "Partie reelle de x1 " << x1.getReelle()<<endl;
    x1.setImaginaire(1);
    cout <<"Affichage de x1: ";        x1.afficher();
    x3.setRho(1.0);
    cout <<"Affichage de x3 : ";       x3.afficher();
    cout <<"Costa:"<<x2.costa()<<endl;
    x3=x1+x2;
    x4=x3;
    cout <<"x3="<<x3;
    cout <<"x4="<<x4;
    MesComplexes x5(x3);
    x4 = x1 * x2;
    cout <<"x4="<<x4;
    x5 = x5 * x1;
    cout <<"x5="<<x5;
    x5 = x1 * x1 * x1 * x1;
    cout <<"x5="<<x5;
    x5 = pow(x1,6);
    cout <<"x5="<<x5; }
```

Il vous appartient de compléter si besoin ce programme de test pour tester toutes vos méthodes et fonctions.

En l'état ce programme devrait afficher :

```
1 + 10*i
4 + 2*i
4 + 2*i
Partie reelle de x1 1
Affichage de x1: 1 + 1*i
Affichage de x3 : 0.894427 + 0.447214*i
Costa:8
x3=5 + i* 3
x4=5 + i* 3
x4=2 + i* 6
x5=2 + i* 8
x5=-4 + i* 0
x5=0 + i* -8
```