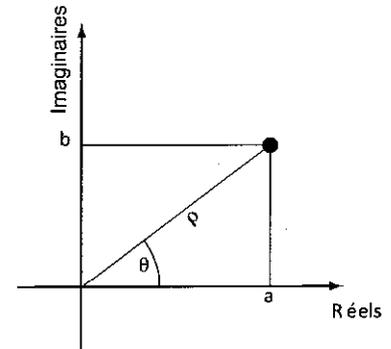


Nombres Complexes

Le but de ce TP est de vous faire implémenter en C++ un patron (template) de nombres complexe sous deux formes différentes : les nombres complexes cartésiens $(a + ib)$ et les nombres complexes d'Euler (ou polaires) : $\rho e^{i\theta}$.

Il s'agit donc ici de définir les membres de ces deux formes de nombres complexes, mais surtout les différentes méthodes et opérateurs communs à toute forme de nombres complexes.



Opérations sur les nombres complexes cartésiens

Addition : $(a + ib) + (c + id) = (a + c) + i(b + d)$

Multiplication : $(a + ib)(c + id) = (ac - bd) + i(bc + ad)$

Conjugué : $\overline{a + ib} = a - ib$ & $(a + ib)(a - ib) = (a^2 + b^2)$

Inverse : $\frac{1}{a + ib} = \frac{(a - ib)}{(a + ib)(a - ib)} = \frac{a - ib}{a^2 + b^2} = \left(\frac{a}{a^2 + b^2}\right) + i\left(\frac{-b}{a^2 + b^2}\right)$

Division : $\frac{(a + ib)}{(c + id)} = \frac{(a + ib)(c - id)}{(c + id)(c - id)} = \left(\frac{ac + bd}{c^2 + d^2}\right) + i\left(\frac{bc - ad}{c^2 + d^2}\right)$

Module : $|a + ib| = \sqrt{a^2 + b^2}$

$$\text{Argument : } \theta = \begin{cases} \arctan\left(\frac{b}{a}\right) & \text{si } a > 0 \\ \arctan\left(\frac{b}{a}\right) \pm \pi & \text{si } a < 0 \\ +\frac{\pi}{2} & \text{si } a = 0 \text{ \& } b > 0 \\ -\frac{\pi}{2} & \text{si } a = 0 \text{ \& } b < 0 \\ \text{indéfini} & \text{si } a = 0 \text{ \& } b = 0 \end{cases}$$

Opérations sur les nombres complexes polaires

$\rho e^{i\theta} = \rho(\cos(\theta) + i \sin(\theta))$

Partie réelle : $\rho \cos(\theta)$

Partie imaginaire : $\rho \sin(\theta)$

Multiplication : $(\rho_1 e^{i\theta_1})(\rho_2 e^{i\theta_2}) = (\rho_1 \rho_2) e^{i(\theta_1 + \theta_2)}$

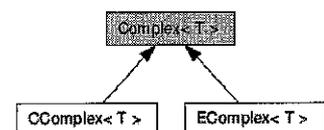
Conjugué : $\overline{\rho e^{i\theta}} = \rho e^{-i\theta}$

Inverse : $(\rho e^{i\theta})^{-1} = \frac{1}{\rho} e^{-i\theta}$

Travail à réaliser

A partir de la classe abstraite `Complex<T>` qui vous est fournie, développez par héritage les classes concrètes `CComplex<T>` et `EComplex<T>` qui implémentent respectivement les nombres complexes cartésiens et les nombres complexes polaires (Complexes d'Euler).

Un squelette du code vous est fournit dans `~[david.]roussel/pub/ILO/TP3.zip`



Classe abstraite Complex<T>

Le patron Complex<T> est abstrait (la classe contient des méthodes pures virtuelles = 0) et définit l'interface que doivent respecter toutes les implémentations des nombres complexes (cartésiens et d'Euler).

Attributs

Le patron Complex<T> contient les constantes de classes ESPILON utilisée pour comparer deux valeurs à epsilon près ainsi que ZERO qui définit l'élément nul. Ces deux constantes seront spécialisées en fonction du type T utilisé pour instancier le template Complex<T>.

Le patron Complex<T> contient aussi un compteur d'instances « NbInstances » qui sera utilisé pour donner un numéro d'instance (instanceNumber) à chaque variable de type Complex<XXX>, où XXX est un type concret. Vous remarquerez que chaque type de complexe <double>, <float> ou <int> aura son propre compteur d'instances contrairement à Java où les variables de classes d'une classe générique sont partagées par toutes les instanciations concrètes (à cause du Type Erasure).

Méthodes

Constructeurs et destructeur

Les constructeurs du patron Complex<T> sont protégés pour éviter que l'utilisateur ne puisse les appeler dans la mesure où la classe est abstraite. Ils seront utilisés dans les classes filles. Ils ne sont là que pour incrémenter NbInstances et donner un numéro d'instance à l'instance courante. De même le destructeur ne fait que décrémenter NbInstances.

Propriétés

Le patron contient les déclarations abstraites des propriétés de tous les nombres complexes telles que

- la partie réelle (real()),
- la partie imaginaire (imag()),
- la norme (abs())
- et l'argument (arg()).

Self-opérations

Le patron contient les déclarations abstraites des opérateurs affectant l'instance courante et renvoyant une référence à l'instance courante telles que

- l'affectation : opérateur de copie Complex<T> & operator = (const Complex<T> &)
- la self-addition : Complex<T> & operator += (const Complex<T> &)
- la self-soustraction : Complex<T> & operator -= (const Complex<T> &)
- la self-multiplication : Complex<T> & operator *= (const Complex<T> &)
- la self-division : Complex<T> & operator /= (const Complex<T> &)

Toutes les self-opérations renvoient une référence vers le complexe courant afin de pouvoir être cumulées.

Opérations

Le patron contient les déclarations abstraites des opérateurs arithmétiques sur les complexes avec la particularité que toutes les opérations arithmétiques renvoient un pointeur vers une nouvelle instance. Ces opérateurs ne peuvent pas renvoyer une instance directement puisque la classe Complex<T> étant abstraite ne peut pas avoir d'instances mais on peut s'y référer par pointeurs ou par références. Les classes filles pourront en revanche posséder leurs propres opérateurs arithmétiques renvoyant des instances dans la mesure où elles seront concrètes.

- Addition : Complex<T> * operator +(const Complex<T> &) const
- Soustraction : Complex<T> * operator -(const Complex<T> &) const
- Multiplication : Complex<T> * operator *(const Complex<T> &) const
- Division : Complex<T> * operator /(const Complex<T> &) const

Comparaison

Le patron Complex<T> définit et implémente les opérateurs de comparaison entre deux complexes :

- Comparaison : bool operator ==(const Complex<T> &) const
- Différence : bool operator !=(const Complex<T> &) const

Affichage

Le patron `Complex<T>` définit de manière abstraite une méthode `toString` permettant de générer une chaîne de caractères représentant le nombre complexe. Cette méthode abstraite est utilisée dans les opérateurs de sortie standard à partir d'une référence et d'un pointeur :

- `string toString()`
- opérateur de sortie standard d'après une référence :
`ostream & operator << (ostream &, const Complex<T> &)`
- opérateur de sortie standard d'après un pointeur :
`ostream & operator << (ostream &, const Complex<T> *)`

Instanciation et spécialisation

Un patron de classe n'est pas une classe tant qu'il n'a pas été instancié avec un type concret. C'est pourquoi, à la fin de l'implémentation du patron `Complex<T>` il faut « proto-instancier » celui-ci avec des types concrets : par exemple `Complex<double>` pour forcer le compilateur à générer le code objet correspondant à cette instanciation particulière du patron `Complex<T>`.

Par ailleurs, on peut aussi spécialiser des membres (attributs et méthodes) d'un patron en fonction d'un type concret. C'est le cas pour les constantes de classe `EPSILON` et `ZERO`.

Classe des complexes cartésiens `CComplex<T>`

Attributs

Les Complexes cartésiens contiennent des attributs stockant la valeur réelle et imaginaire (`_real` et `_imag`).

Méthodes

Constructeurs

`CComplex<T>` contient constructeur :

- Un constructeur valué avec des valeurs par défaut à `ZERO` ce qui permet de l'utiliser comme constructeur par défaut : `CComplex(const T r = Complex<T>::ZERO, const T i = Complex<T>::ZERO)`
- Un constructeur de copie à partir d'un `Complex<T>` de même type :
`CComplex(const Complex<T> & c)`
- Un constructeur de copie/conversion à partir d'un `Complex<U>` d'un type différent. Ce constructeur étant lui-même un patron, il faudra le spécialiser en fonctions des conversions que l'on souhaite réaliser :
`template <class U> CComplex<T> ::CComplex(const Complex<U> & c)`

Propriétés

`CComplex<T>` implémente les propriétés

- `real()` et `imag()` en lecture seule (const) et en lecture/écriture (non const)
- `abs()` et `arg()` en lecture seule (const)

Self-Opérations

`CComplex<T>` implémente les self-opérations et spécialise l'opérateur d'affectation (`operator=(...)`) pour un faire un opérateur de copie/conversion permettant de convertir une nature de complexe (U) vers une autre (T) comme avec le constructeur de copie/conversion :

`template <class U> CComplex<T> ::operator =(const Complex<U> & c).`

Opérations

`CComplex<T>` implémente les opérations définies dans `Complex<T>` (+, -, * et /) ainsi que des implémentations spécifiques à `CComplex<T>` qui renvoient une instance plutôt qu'un pointeur (plus faciles à utiliser) :

- Addition : `CComplex<T> operator +(const CComplex<T> &) const`
- Soustraction : `CComplex<T> operator -(const CComplex<T> &) const`
- Multiplication : `CComplex<T> operator *(const CComplex<T> &) const`

- Division : `CComplex<T> operator /(const CComplex<T> &) const`
- Inverse : `CComplex<T> inverse() const`
- Conjugué : `CComplex<T> conjugate() const`

Les deux types d'implémentation pourront utiliser la combinaison d'un constructeur de copie et d'une self-opération afin de n'écrire les algorithmes arithmétiques qu'une seule fois.

Exemple avec un opérateur hypothétique \odot :

```
template<class T>
CComplex<T> CComplex<T>::operator  $\odot$ (const CComplex<T> & c) const
{
    CComplex<T> nouveau(*this);
    nouveau.operator  $\odot$ =(c);
    return nouveau;
}

template<class T>
Complex<T> * CComplex<T>::operator  $\odot$ (const Complex<T> & c) const
{
    CComplex<T> * nouveau = new CComplex<T>(*this);
    nouveau->operator  $\odot$ =(c);
    return nouveau;
}
```

Comparaison

Les opérateurs de comparaison « == » et « != » sont déjà implémentés dans le patron `Complex<T>`

Entrées / Sorties

- La méthode `toString()` reste à implémenter.
- Il faut aussi implémenter un opérateur d'entrée standard afin de pouvoir lire un complexe cartésien sur la console ou depuis un fichier en fournissant la partie réelle et imaginaire :
`template <class T> istream & operator >>(istream & in, CComplex<T> & c)`

Instanciation et Spécialisation

Toutes les proto-instanciations réalisées dans `Complex<T>` devront avoir leur pendant dans `CComplex<T>`

Classe des complexes d'Euler `EComplex<T>`

Attributs

Le Complexes d'Euler contiennent des attributs stockant la norme et l'argument (`_abs` et `_arg`).

Méthodes

Constructeurs

- Un constructeur valué avec des valeurs par défaut à ZERO ce qui permet de l'utiliser comme constructeur par défaut : `EComplex(const T norm = Complex<T>::ZERO, const T argument = Complex<T>::ZERO)`
- Un constructeur de copie à partir d'un `Complex<T>` de même
- Un constructeur de copie/conversion à partir d'un `Complex<U>` d'un type différent.

Le reste des méthodes du patron `EComplex<T>` à implémenter sont les mêmes que dans `CComplex<T>`

Programme de test

Le programme de test contenu dans `TestComplex.cpp` contient 3 blocs que vous pourrez décommenter en fonctions de votre avancement dans l'implémentation.

- Un premier bloc spécifique aux complexes cartésiens

- Un second bloc spécifique aux complexes d'Euler
- Un troisième bloc mélangeant les deux types

Annexes

Structure des fichiers

Les fichiers contenant du code C++ se distinguent de ceux contenant du C par leur extension. Le compilateur considère des fichiers C++ si ceux ci se terminent par l'extension « .C » (majuscule), ou bien « .cc », ou encore « .cpp ». D'autres compilateurs attendent l'extension « .cxx ». Les objets C++ doivent être définis par des couples de fichiers « MonObjet.h / MonObjet.cpp ». « MonObjet.h » contiendra la définition d'un objet, et « MonObjet.cpp » contiendra son implémentation. On rajoute un programme principal « TestMesObjets.cpp » pour tester les différents objets.

Makefile

Vous disposez d'un Makefile complet qui vous permet de compiler l'ensemble de vos sources avec « make » ou « make all ». Ainsi que d'autres targets utiles telles que la génération des listings en PDF « make pdf » ou bien la génération de la documentation des classes avec doxygen (« make doc »).

Compilation

« g++ » : Le compilateur C++ de GNU.

g++ agit de manière similaire à gcc mais est spécialisé dans la compilation de sources C++.

Ses options sont donc sensiblement les mêmes que gcc.

Documentation

« doxygen » : Utilitaire de documentation. Similaire à javadoc.

Pour documenter les classes nous ne documenterons que leur définitions, c'est à dire les fichiers « *.h » de chaque objet. Les commentaires de documentation de doxygen sont exactement les mêmes que ceux de Javadoc.

On obtiendra aisément un résumé des options de doxygen avec la commande :

```
> doxygen --help
```