Séance 3

Surcharge des opérateurs

complexe.h

```
#ifndef DEF_COMPLEXE
#define DEF_COMPLEXE
 class Complexe
{
   public:
            Complexe();
            Complexe(int r);
            Complexe(int r, int i);
            ~Complexe();
            // autres méthodes
   private:
          int reel (0);
          int imaginaire(0);
};
 #endif
```

Utilisation

- Chaque objet de type Complexe stockera une valeur pour le reel et une autre valeur pour l'imaginaire.
- Nous avons utilisé des valeurs par défaut au cas où l'utilisateur oublierait de les préciser.

On pourra donc créer un objet de plusieurs façons différentes :

Complexe z1; // Pour stocker 0 au reel et 0 à l'imaginaire
Complexe z2(5); // Pour stocker 5 au reel et 0 à l'imaginaire
Complexe z3(5, 30); // Pour stocker 5 au reel et 30 à l'imaginaire

Utilisation

L'implémentation du constructeur (complexe.cpp)

```
#include " complexe.h"

Complexe::Complexe(int r, int i) : reel(r), imaginaire(i) {
}
```

L'implémentation du main (main.cpp)

```
#include "complexe.h"
 int main(){
    Complexe z1(0, 10), z2(5);
    return 0;
}
```

Surcharge des opérateurs

- Opérateurs arithmétiques:

  operator+, operator-, operator*, operator/
- Opérateurs de comparaison :

  operator==, operator!=, operator<, operator>, operator<=, operator>=
- Opérateurs raccourcis:

  operator+=, operator-=, operator*=, operator/=
- operator<<, operator>>
- …

Des opérations sur les complexes

- Supposons que vous avez 2 nombres complexes et vous souhaitez les ajouter entre eux

- Complexe z, z1 et z2
- z = additionner(z1,z2);
  - La fonction additionner réalisera la somme de z1et de z2
  - Cela fonctionne mais ce n'est pas très lisible
- z=operator+(z1, z2)
- z=z1.additionner(z2)
- z=z1.operator+(z2)
- L’idéal est de faire z= z1 + z2
- Comment le faire comprendre à l’ordinateur ?

Comparaison entre 2 complexes

Pour être capables d'utiliser le symbole « == » entre deux objets, vous devez créer une fonction ayant précisément pour nom operator == et dotée du prototype :

//function normale en dehors de toute classe
bool operator==(const Objet & a, **const** Objet & b);
bool comparer(const Objet & a, **const** Objet & b);

Une fois cette fonction est implémentée, vous pouvez comparer 2 complexes

```
if(a == b) {
    std::cout << "Les deux objets sont egaux !" << std::endl;
}
```

Le compliteur traduit ce code par :

```
if (operator==(a, b)) {
    std::cout << "Les deux objets sont egaux !" << std::endl;
}
```

L'opérateur se transforme en appel de function (**operator==**) qui renvoit un bool

Implémentation de l'operateur ==

```
bool operator==(const Complexe & a, const Complexe & b)
{
   //Teste si.
   if (a.reel == b.reel && a.imaginaire == b.imaginaire)
      return true;
   else
      return false;
}
```

Problème : on a pas accès aux attributs de la classe complexe car déclarés **private**

Il existe quatre solutions à ce problème:

- Vous créez des accesseurs comme on l'a vu (ces fameuses méthodes getter et setter (ce n'est pas élégant)
- Passer par une autre méthode de la classe qui fait la comparaison.
- Utilisation du concept d'amitié.
- Ou simplement définir la fonction **operator==**() comme méthode de la classe

Passer par les accesseurs

complexe.h

```
#ifndef DEF_COMPLEXE
#define DEF_COMPLEXE
class Complexe
{
 public:
Complexe(int r = 0,  int i = 0);
int getReel();
int getImag();
~Complexe();

private:
   int reel;
   int imaginaire;
};
 #endif
```

complexe.cpp

```
int Complexe::.getReel() {
  return reel;
}
int Complexe::getImag(){
   return imaginaire;
}
```

main.cpp

```
bool operator==(const Complexe & a, const Complexe t& b)
{
   return (a.getReel() == b.getReel() && a.getImag() == b.getImag());
}
void main(){
  Complexe z1, z2;
    if(z1 == z2) {
        std::cout << "Les deux complexes sont egaux !" << std::endl;
     }
}
```

Utilisation d'une méthode de comparaison de la classe : estEgal()

complexe.h

```
#ifndef DEF_COMPLEXE
#define DEF_COMPLEXE
class Complexe
{
 public:
Complexe(int r= 0, int i = 0);
bool estEgal (const Complexe &) const;
~Complexe();
private:
   int reel;
   int imaginaire;
};
 #endif
```

complexe.cpp

```
bool Complexe::estEgal(const Complexe & b) const
{
   if (reel == b.reel && imaginaire == b.imaginaire)
       return true;
    else
       return false;
}
```

Ou méthode plus courte

```
bool Complexe::estEgal(const Complexe & b) const
{
   return (reel == b.reel && imaginaire == b.imaginaire);
}
```

On utilise cette methode dans l'opérateur de comparaison

```
bool operator==(const Complex & a, const Complex & b)
{
   return a.estEgal(b);
}
```

Dans la fonction main()

```
#include  " complexe.h "
using namespace std;
int main()
{
   Complexe z1(5), z2(5, 0);
   if (z1 == z2)
      cout << "Les complexes sont identiques";
   else
      cout << "Les complexes sont differentes";
   return 0;
}
```

Resultat:

Les complexes sont identiques

Operator== : fonction amie

complexe.h

```
#ifndef DEF_COMPLEXE
#define DEF_COMPLEXE

class Complexe
{
   public:
          Complexe(int r = 0, int i = 0);
          ~Complexe();
          friend bool operator== (const Complexe & , const Complexe & );
          // autres méthodes
   private:
         int reel;
         int imaginaire;
};
 #endif
```

Operator== : fonction amie

Implémentation de la fonction ami en dehors de la classe Complexe, par exemple dans le fichier main.cpp

```
bool operator== (const Complexe & z1, const Complexe & z2) {
      return (z1.reel==z2.reel && z1.imaginaire==z2.imaginaire);
}
```

Utilisation de la fonction ami

```
#include  " complexe.h "
using namespace std;
int main()
{
   Complexe z1(5), z2(5, 0);
   if (z1 == z2)
      cout << "Les complexes sont identiques";
   else
      cout << "Les complexes sont differentes";
   return 0;
}
```

"*Whenever you can avoid friend functions, you should, because, much as in real life, friends are often more trouble than they're worth." Scott Meyers :-)*

Methode de la classe

complexe.h

```
#ifndef DEF_COMPLEXE
#define DEF_COMPLEXE
class Complexe
{
 public:
Complexe(int m_reel = 0,
                int m_imaginaire = 0);
bool operator== (const Complexe &);

~Complexe();

private:
   int reel;
   int imaginaire;
};
 #endif
```

complexe.cpp

```
#include  " complexe.h "
bool Complexe ::operator== (const Complexe& a) {
      return (reel==a.reel && imaginaire==a.imaginaire);
}
```

main.cpp

```
#include <iostream>
#include  " complexe.h "
using namespace std;

void main()[
   Complexe z1, z2;
    if(z1 == z2) {
        cout << "Les deux complexes sont egaux !" << endl;
     }
}
```

fraction.h

```
class Fraction {
    private :
        int numerateur (0);
        int denominateur (1);
    public :
        Fraction();
        Fraction(int n);
        Fraction(int n, int d);
        ~Fraction();
        void operator*=(const Fraction & autre) ;
        // autres méthodes
};
```

operator*= : fonction membre

```
// fichier fraction.cpp
void Fraction::operator*=(const Fraction & autre) {
    numerateur *= autre.numerateur;
    denominateur *= autre.denominateur;
}

// fichier main.cpp
Fraction f1(4,5);
Fraction f2(3,11);
f1 *= f2; //f1.operator*=(f2) , // fraction f1 =>12/55
f1 *= 2;    // f1. operator*=(2/1);  f1 => 8/5
```

operator* : fonction membre

```
Fraction operator*(const Fraction & autre) const;

Fraction Fraction::operator*(const Fraction & autre) const {
    return Fraction(numerateur*autre.numerateur,
                    denominateur *autre.denominateur);
}

Fraction f1(1,8);
Fraction f2(1,2);
Fraction res = f1 * f2;              // OK plus elegant
Fraction res = f1.operator*(f2);
Fraction res.operator=(f1.operator*(f2));
```

operator* : fonction membre

```
Fraction operator*(const Fraction & autre) const;

Fraction Fraction::operator*(const Fraction & autre) const {
    return Fraction(numerateur*autre.numerateur,
                    denominateur *autre.denominateur);
}
```

```
res = f1 * 2;          // OK  res= f1.operator*(2/1);
res = 2 * f1;          // not OK, why ?
```

operator* : fonction membre

```
Fraction operator*(const Fraction & autre) const;

Fraction Fraction::operator*(const Fraction & autre) const {
    return Fraction(numerateur*autre.numerateur,
                    denominateur *autre.denominateur);
}
```

```
res = f1 * 2;        // OK          res = f1.operator*(2)
res = 2 * f1;        // not OK,     res = 2.operator*(f1)
```

operator* : fonction **non membre** pour "mixed-mode arithmetic"

```
Fraction operator*(const Fraction & lho, const Fraction & rho)
   {
   return Fraction(lho.getNumerateur()*rho.getNumerateur(),
                   lho.getDenominateur() *rho.getDenominateur());
}
```

```
res = f1 * 2;          // OK
res = 2 * f1;          // OK
```

Operator << : fonction amie ?

A ajouter dans la classe Fraction.h

```
friend ostream& operator<< (ostream& out, const Fraction f);
```

Implémentation en dehors de la classe Fraction pourquoi pas le fichier main.cpp

```
ostream& operator<< (ostream& out, const Fraction f) {
      out << "Fraction : " << f.numerateur  << "/" << f.denominateur  << endl;
      return out;
}
```

Utilisation dans le main

```
void main(){
   Fraction f1(5,7);
   Fraction f2();
   cout << f1 << f2;
}
```