Programmation orientée objet

Surcharge des opérateurs et fonctions (Introduction)

Besoin de la surcharge d'opérateurs

- Variables de types simples int, float, double, où l'on peut faire des opérations arithmétiques à l'aide d'opérateur.
- **Opérations arithmétiques sur des objets**.
- **Nouvelles fonctionnalités aux opérateurs**.
- Aucun type Fraction ou Complexe d'où le besoin de créer une classe Fraction ou Complexe.
- Ces classes posséderaient tous les opérateurs définis pour un type tel qu'un entier ou un double (+, -, +=, etc).

La classe Fraction

```
class Fraction
{
   public:
      Fraction();
      Fraction (const double& num, const double& denum);
      long getNumerateur() const;
      long getDenominateur() const;
      double getReel() const;
  private:
      long numerateur_;
      long denominateur_;
      void simplifier();


};
```

Application de la surcharge des opérateurs

```
Fraction f1(2,3), f2(1,5), f3;
f3 = f1 + f2;
f3 = f1 + 10;
cin >> f1;
cout << f1;
```

Surcharges d'opérateurs

- Consiste **à redéfinir la fonctionnalité d'un opérateur** tel que +, -, ou += pour une classe.
- Il faut bien comprendre le fonctionnement de l'opérateur.
- **L'ordre de priorité est conservé.**
- L'objet est toujours l'argument de gauche.
- Les opérateurs ont un nombre déterminé de paramètres et ne peuvent pas avoir de paramètres par défaut (opérateurs binaires et unaires).
- Il n'est **pas possible de créer de nouveaux opérateurs.**

Surcharges d'opérateurs

Pouvant être surchargés

```
+ - * / % ^ & | [] ()
~ = < > += -= *= /=
%= ^= &= ! ++ -- -> ,
>> << && != <= >=
\= || <<= >>= ==
new delete
```

Ne pouvant être surchargés

```
. :: ?: .* sizeof
```

Surcharges d'opérateurs binaires

- Opérateur à deux opérandes.
- Il doit y avoir un opérande à gauche et à droite du symbole de l'opérateur.

Opérateur binaire

- Fraction f1(1,2), f2(4,5), f3;
- f3 = f1 + f2; // f1.operator+(f2);
- operator+() fonction membre de la classe Fraction, car f1 est un objet de la classe Fraction.
- Retourne un objet de Fraction.

fraction.h

```
class Fraction {
   public:
      Fraction();
      Fraction (const double& num, const double& denum);
      long getNumerateur() const;
      long getDenominateur() const;
      double getReel() const;
      Fraction operator+ (const Fraction& fract) const;
  private:
       long numerateur_;
       long denominateur_;
       void simplifier();

};
```

Implémentation d'opérateur + de Fraction

```
Fraction Fraction::operator+(const Fraction& fract) const
{
  Fraction somme ;

  somme.numerateur_=numerateur_*fract.denominateur_+
                    fract.numerateur_*denominateur_;

  somme.denominateur_=denominateur_*fract.denominateur_;

  somme.simplifier();

  return somme;
}
```

Remarque: des fonctions de la même classe peuvent toujours accéder aux attributs privés des autres objets de cette classe!

Implémentation de l'opérateur + de Fraction (suite)

```
Fraction f1(2,3), f2;
f2 = f1 + 4; // f2 = f1.operator+(4);
```

2 possibilités:

- Fonction membre avec comme paramètre un long
- Constructeur qui prend comme paramètre un long (conversion implicite: voir livre)

Implémentation de l'opérateur + de Fraction (suite)

```
Fraction Fraction::operator+(const long& entier) const
{
  Fraction resultat;
  resultat.numerateur_ =
      numerateur_ + entier * denominateur_;
  resultat.denominateur_ = denominateur_;
  resultat.simplifier();
  return resultat;
}
```

Opérateur + n'est PAS commutatif!

```
Fraction f1(1,2), f2(4,5), f3;
long l = 12;
f3 = f1 + l; // f1.operator+(l);
// Fraction operator + (const long&) const;
```

Mais

l.operator+(f1) ?

```
f3 = l + f1;
```

Solution 1: fonctions globales

```
class Fraction{
   public:
     …
     Fraction operator+(const Fraction&) const;
     Fraction operator+(const long&) const;
  private:
     …
};

Fraction operator+(const long&, const Fraction&);
```

.h

pas une fonction membre

```
Fraction operator+(const long& e, const Fraction& f){
    return (f + e);
}
```

.cpp

Solution préférée: fonctions globales « friend »

```
class Fraction{
   public:
     …
     Fraction operator+(const Fraction&) const;
     Fraction operator+(const long&) const;
     friend Fraction operator+(const long&,
        const Fraction&);

   private:
     …
};
```

.h

une fonction globale, MAIS avec accès privilégié aux attributs privés

```
Fraction operator+(const long& e, const Fraction& f){
    return (f + e);
}
```

.cpp

Mais friend détruit l'encapsulation?!

- OUI, des fonctions non-membres ont accès aux attributs privés

- NON, une **classe contrôle** quelle fonctions deviennent friend
- NON, les opérateurs font partie de l'interface et alors doivent être inclus dans la définition de la classe
- NON, c'est plus fiable qu'ajouter des accesseurs publiques juste pour l'opérateur

Surcharge de flux d'entrée et de sortie

- Étant donné que le paramètre de gauche de l'opérateur << ou >> doit être un stream, il doit être surchargé par une fonction globale ou friend.

- Cet opérateur doit retourner une référence à un stream afin de permettre les cascades.

```
cout << f1 << ": " << f2
```

Surcharge de flux de sortie (suite)

.h

```
class Fraction{
 public:
    …
  friend ostream& operator<<(ostream& o,const Fraction& f);
    …
 private:
    …
};
```

référence à stream

accès aux attributs privés!

.cpp

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

Le pointeur this

- Le pointeur **`this`** est un pointeur sur **l'objet courant**
- Toutes les méthodes ont ce pointeur
- Ainsi, dans une méthode d'un objet ayant un attribut dénommé x, les deux instructions suivantes sont équivalentes:

```
x_ = 2;
this->x_ = 2;
```

Une application du pointeur this: appels en cascade

- Supposons que nous voulions appeler plusieurs fois consécutives une méthode d'un objet
- Imaginons par exemple une méthode incrementer(), que l'on veut appliquer plusieurs fois:

```
objet.incrementer();
objet.incrementer();
objet.incrementer();
```

- Si la méthode nous retournait une référence au même objet, le même effet pourrait être obtenu en faisant des appels en cascade:

```
objet.incrementer().incrementer().incrementer();
```

Pointeur this et appels en cascade

```
class ClasseA
{
public:
  ClasseA();
   ...
  ClasseA& incrementer();
private:
   int att_;
};
```

```
ClasseA::ClasseA():att_(0){}

ClasseA& ClasseA::incrementer()
{
   ++att_;
   return *this;
}
```

Surcharges d'opérateurs unaires

```
class Fraction{
 public:
    …
  Fraction& operator++ ();     // ++f; = prefix ++
  Fraction operator++ (int);   // f++; = postfix ++
    …
};
```

aucun paramètre

.h

dummy paramètre (PAS utilisé, même pas un nom)

```
Fraction& Fraction::operator++(){
  *this=*this+1;//mets à jour utilisant operator+, et …
  return *this; //retourne l'objet mis à jour
}
Fraction Fraction::operator++(int){
  Fraction res=*this;
  ++(*this); //mets à jour l'objet, mais …
  return res;//retourne une copie de l'objet original
}
```

.cpp

Pourquoi la surcharge fonctionne?

```
void f(int a);

void f(int a,double b);

void f(int a,double b,int c);

void f(int a,double b,string s);

void f(int a,double b,Fraction f);

void f(int a,double b,Point p);
```

nombre différent de paramètres

ou différents types de paramètres

MAIS: fais attention aux **conversion automatique** des types (« casts »)

La classe Company

```
class Company
{
public:
    Company(string name, string presidentName);
    bool operator==(const Company &company) const;
    Company& operator+=(Employee *employee);
    ...
    Company operator+(Employee &employee) const;
    Company operator+(const Company &company) const;
    ...
    friend ostream& operator<<(ostream& os, const Company&);

private:
    ...
    vector<Employee*> employees_;
};
```

Opérateur == pour comparer deux compagnies

Opérateur de flux de sortie, affichant chaque employé

La classe Company (suite)

```
ostream& operator<<(ostream &os, const Company &company)
{
   os << "This is the company " << company.getName()
    << " presided by " << company.getPresident()->getName() << endl;
   os << "Employees: " << endl;
   for (int i = 0; i < company.getNumberEmployees(); i++)
      os << company.getEmployee(i)->getName() << " "
         << company.getEmployee(i)->getSalary() << endl;

   return os;
}

Company& Company::operator+=(Employee *employee)
{
        addEmployee(employee);
        return *this;
}
```