Modélisation objet& programmation objet en C++

TELLEZ Bruno, IUT Lyon1.
Département Informatique, Site de Bourg-en-Bresse
2023-2024

Séance 2

Copie or not Copie ?

- Station::addCapteur(Mesure);
- Mesure m1(m2);
- Mesure m1 = m3;
- Mesure m4; m4=m5;

Mesure m4; m4=m5;

Si ce n'est pas un constructeur par copie, c'est quoi ?

C'est une surcharge (redéfinition) de l'opérateur =

m4 = m5 en fait, c'est m4.operator =(m5)

Mesure& Mesure::operator = (const Mesure&)

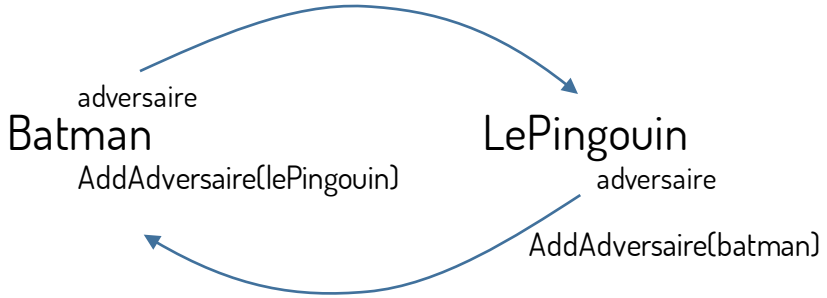
Mais avant de surcharger l'operateur =, parlons de this

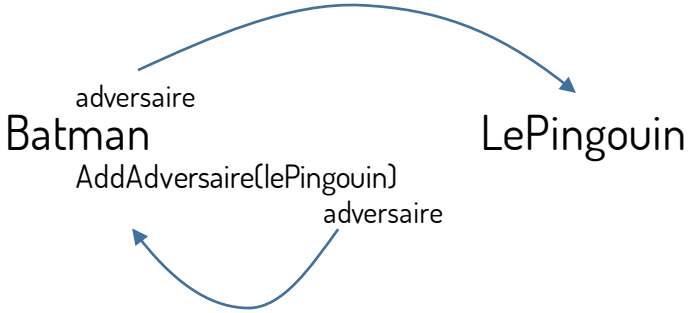
this, le pointeur qui dit à l'objet qui il est, où il est

this est un *pointeur constant* qui est ajouté à chaque objet et qui porte l'adresse de cet objet.

Un exemple pour des relations reflexives (type *association*)

```
class SuperHeros {
        SuperHeros * adversaire;
        void AddAdversaire(SuperHeros *)
};
```






```
void SuperHeros:: AddAdversaire (SuperHeros* a){
    if (adversaire == NULL)
        adversaire = a;
        a. AddAdversaire(this);
    }
};
```

Pourquoi l'opérateur d'affectation (=) c'est plus compliqué ?

Reprenons le constructeur par copie

```
Mesure(const Mesure& m) {
    name = m.name; nbElements = m.nbElements;
    data = new int[nbElements];
    for (int i = 0; i < nbElements; i++) {
        data[i] = m.data[i];
    }
}
```

```
Mesure& Mesure::operator =(const Mesure& m) {
    if (this != &p) {   // Pour que B2=B2; fonctionne
        name = m.name; nbElements = m.nbElements;
        delete [] data;
        data = new int[m.nbElements];
        for (int i = 0; i < m.nbElements; i++) {
                data[i] = m.data[i];
        }
    }
    return (*this);    // Pour pouvoir faire B2=B3=B4;
}
```

Les surcharges d'opérateurs

On peut surcharger l'opérateur = mais les autres ?
Oui on peut quasiment tous (sauf ./?:/::/sizeof/typeid)

```
mystring1+mystring2;
madate+3;
montableau(8,2);
```

Un exemple : la classe *Rationnel*

```
class Rationnel  {
private:
    int Num;
    int Denom;
public:
    Rationnel(int n, int d) : Num(n) {
        if (d != 0)
            Denom = d;
    }
};
```

Une première surcharge
l'opérateur +

```
Rationnel operator+(const Rationnel& r) {
    Rationnel result(Num * r.Denom + Denom * r.Num, Denom * r.Denom);
    //je créé un nouveau Rationnel pour mon résultat
    return result;
    // je le retourne par copie pour ne pas la 'perdre'
}
```

Quelques améliorations

```
Rationnel operator*(const Rationnel& r) {
    Rationnel result(Num * r.Num, Denom * r.Denom);
    result.simplifie();
    return result;
}
```

```
void simplifie() {
    int m = abs(Num) < abs(Denom) ? abs(Num) : abs(Denom);
    for (int i = m; i > 1; i--) {
        if (Num % i == 0 && Denom % i == 0) {
                Num /= i; Denom /= i;
        }
    }
}
```

Le cas de l'affichage *l'operateur <<*

```
void Rationnel::affiche() {
        cout << Num << "/" << Denom << endl;
}

Rationnel r1(5,12);
r1.affiche();
```

Et pourquoi pas : `cout << r1` ?

La surcharge de l'opérateur <<

```
ostream& operator<<(ostream& o, const Rationnel& r)
{
    o << p.getNum() << ‘’/‘’ << p.getDenom() << endl;
    return o;
}
```

Attention aux accesseurs qui doivent avoir un statut de fonction constante
int getNum() const
pour s'appliquer sur des objets constants

Une autre surcharge moins évidente

```
float evaluate() {
       return (float)Num / Denom;
 }

Rationnel r2(-12,8);
float fe = r2.evaluate();
cout << fe;
```

Et pourquoi pas : float fe = r2 directement ?

Quelques remarques sur les surcharges d'operateurs

Opérateur de cast

```
Rationnel::operator double() const
{
    return (double) num / denom;
}
```

Mais aussi les Operateurs d'incrémentation ++

Incrémentation préfixe ++a A& operator++ ();

Incrémentation postixe a++ A operator++ (int);

Attention à bien respecter les règles arithmétiques sur les opérateurs comme ==, < ou > etc...

A l'intérieur ou à l'extérieur ?

Une surcharge peut se définir à l'intérieur de la classe ou à l'extérieur
Soit la classe A A::operator +(const A& a) ou operator +(const A& a1, constA& a2)

On privilégiera la forme à l'extérieur quand :

- Celle à l'intérieur n'est pas possible !
  ex : la surcharge de l'opérateur <<
- Des conversions offriront plus de flexibilité
  ex : Rationnel + double peut être faite à l'intérieur mais double + Rationnel non
  Soit on écrit les deux surcharges à l'extérieur
  Soit on imagine une conversion (cast) ici de double vers Rationnel via un constructeur.

Sur cet exemple d'écriture à l'extérieur, que remarquez-vous ?

```
Rationnel operator +(Rationnel r1, Rationnel r2) {
       return Rationnel(r1.getNum()*r2.getDenom()+r2.getNum()*r1.getDenom(),
                        r1. getDenom()*r2.getDenom()); }
```

Les fonctions amies

Sur l'exemple précédent, des accesseurs (et des mutateurs) qui alourdissent le code

```
class Rationnel {
public :
  friend Rationnel operator+(Rationnel,Rationnel)
};
```

La fonction est rendue amie

```
Rationnel operator +(Rationnel r1, Rationnel r2) {
        return Rationnel(r1.Num*r2.Denom+r2.Num*r1.Denom, r1. Denom*r2.Denom);
}
```