ISEN
ALL IS DIGITAL!
LILLE

yncréa

CIR²

Le Langage C++

La surcharge des opérateurs

Pascal MOSBAH & Nacim IHADDADENE

Exemple

La concaténation de chaînes de caractères

The C way

```
char str[80];
strcpy (str,"Chat ");
strcat (str,"et chien ");
strcat (str,"sont ");
strcat (str,"des quadrupèdes.");
printf("%s\n",str);
```

La taille “statique” (80) du tableau “char” est arbitraire et doit être suffisante pour contenir l’ensemble de la chaîne de caractères

str est de type “char *”, il faut également spécifier le spécificateur de format à la fonction printf.

The C++ way

String est un objet C++ qui peut accepter des opérations mathématiques.

```
string cpp_str = "Chat ";
cpp_str += "et chien ";
cpp_str += "sont ";
cpp_str += "des quadrupèdes.";
cout << cpp_str << endl;;
```

cpp_str est de type string, cout affiche un type string

Principe

- Comment écrire simplement une opération d'addition entre deux fractions (Fraction f3 = f1+f2; et non Fraction f3 = somme(f1,f2); voir la classe Fraction définie précédemment) voire deux objets de type nombre complexe, ou deux objets de type matrice ...

- Comment faire pour que cette opération soit associative et supporte plusieurs éléments (Somme de 3, 4 ou 5 fractions) ?

- Comment se comportent ces nouveaux types (ou classes) avec les autres types ou les types de base ? (Somme d'un entier avec une fraction ou un nombre complexe)

Principe

- Le langage C++ permet de (re)définir le comportement des opérateurs pour les classes développées par l'utilisateur.

- À l'exception des types des bases (int, float, char, double, bool, …), les opérateurs sont des méthodes.

- Syntaxe

```
class C
{
    ...
    public:
    ...
    returnType operatoroperatorName(type1 arg1, ...)
    ...
}
```

Les opérateurs en C++

- Opérateur d'affectation : `=`
- Opérateurs arithmétiques : `+ - * / % ++ -- += -= ...`
- Opérateurs logiques : `&& || !`
- Opérateurs binaires : `& | ^ ~ << >>`
- Opérateurs de comparaison : `< <= > >= == !=`
- Opérateurs d'accès : `[] & * . ->`
- Autres…

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

- Le comportement de ces opérateurs est défini pour les types de base (int, double …)
- Quelques comportements par défaut pour les classes (ex. affectation)

Notations alternatives/équivalentes

- Les opérateurs permettent une écriture plus naturelles pour des fonctions communes:
- Notation préfixée:

  a ⇔ a.operator()

  &a ⇔ a.operator&()

- Notation infixée:

  a->b ⇔ a.operator->(b)

  a + b ⇔ a.operator+(b)

- Notation postfixée

  a[i] ⇔ a.operator[](i)

  i++ ⇔ i.operator++()

Exemple

Surcharge de l’opérateur +

Reprenons la classe Fraction

```
class Fraction {
        private:
        int num = 0;
        int den  = 1;

        public:
        Fraction();
        Fraction(int num, int den);
        void print();
        Fraction plus(const Fraction &other);
        void setNum(int num){
            this->num = num;
        }
        void setDen(int den){
            this->den = den;
        }
};
```

Reprenons la classe Fraction

```
void Fraction::print() {
        cout << num << "/" << den << endl;
}
Fraction::Fraction() {
        cout << "Fraction() builds 0/1" << endl;
    /*
     * this->num = 0;
     * this->den = 1;
    */
}
Fraction::Fraction(int num, int den) {
        cout    << "Fraction(int, int) builds " << num << "/" << den << endl;
        setNum(num);
        setDen(den);
}
Fraction Fraction::plus(const Fraction &other){
      return Fraction(this->num * other.den + this->den * other.num, this->den * other.den);
}
```

Test 1

```
... // all required lines
int main() {
    Fraction f1(1, 2);
    Fraction f2(1, 3);
    Fraction f3 = f1.plus(f2);  // Fraction f3 = f1 + f2;
    Fraction f4 = f1.plus(f2).plus(f3);  // Fraction f4 = f1 + f2 + f3;

    cout << "f1: "; f1.print();
    cout << "f2: "; f2.print();
    cout << "f3: "; f3.print();
    cout << "f4: "; f4.print();

    return 0;
}
```

```
Fraction(int, int) builds 1/2
Fraction(int, int) builds 1/3
Fraction(int, int) builds 5/6
Fraction(int, int) builds 5/6
Fraction(int, int) builds 60/36
f1: 1/2
f2: 1/3
f3: 5/6
f4: 60/36
```

Développer l'opérateur + pour la classe Fraction

Fraction.h : v2

```
class Fraction
{
    private:
    int num;
    int den;

    public:
    Fraction();
    Fraction(int, int);
    void print();
    Fraction operator+(const Fraction&);
    ...
};
```

Développer l'opérateur + pour la classe Fraction

Fraction.cpp : v2

```
...

Fraction Fraction::operator+(const Fraction& other)
{
        cout << "Fraction + Fraction" << endl;

        Fraction result;

        result.num = this->num * other.den + this->den * other.num;
        result.den = this->den * other.den;

        return result;  // peut être simplifié
};
```

Test 2

```
... // all required lines
int main()
{
    Fraction f1(1, 2);
    Fraction f2(1, 3);

    Fraction f3 = f1 + f2;

    cout << "f1: "; f1.print();
    cout << "f2: "; f2.print();
    cout << "f3: "; f3.print();

    return 0;
}
```

On appelle operator+ sur l'instance f1 en passant f2 en paramètre.
Cette méthode renvoie un résultat qui sera affecté à f3

Ecriture équivalente
f3 = f1.operator+(f2);

Output

```
Fraction(int, int) builds 1/2
Fraction(int, int) builds 1/3
Fraction + Fraction
Fraction() builds 0/1
f1: 1/2
f2: 1/3
f3: 5/6
```

Mixer avec les autres types (de la droite)

```
... // all required lines
int main()
{
    Fraction f1(1, 2);
    int f2 = 1;

Fraction f3 = f1 + f2;

    cout << "f1: "; f1.print();
    cout << "f2: "; f2.print();
    cout << "f3: "; f3.print();

    return 0;
}
```

On essaye d'appeler operator+ sur l'instance f1 en passant un entier f2 en paramètre.

Ecriture équivalente
f3 = f1.operator+(1);

Compiler output
Error

```
fracPlusInt.cpp: In function 'int main()':
fracPlusInt.cpp:11:21: error: no match for 'operator+' in 'f1 + f2'
fracPlusInt.cpp:11:21: note: candidates are:
Fraction.cpp:29:10: note: Fraction Fraction::operator+(const Fraction&)
Fraction.cpp:29:10: note:   no known conversion for argument 1 from 'int'
to 'const Fraction&'
```

Mixer avec les autres types (de la droite)

- Revenons sur les messages d'erreurs qui nous mettent sur deux pistes possibles:

```
fracPlusInt.cpp: In function 'int main()':
fracPlusInt.cpp:11:21: error: no match for 'operator+' in 'f1 + f2'
fracPlusInt.cpp:11:21: note: candidates are:
Fraction.cpp:29:10: note: Fraction Fraction::operator+(const Fraction&)
Fraction.cpp:29:10: note:   no known conversion for argument 1 from 'int'
to 'const Fraction&'
```

- **Soit :** Créer une deuxième version de l'opérateur + (le surcharger) pour prendre en paramètre un entier au lieu d'une fraction.

- **Ou :** Trouver un moyen pour convertir un entier en fraction

Mixer avec les autres types (de la droite)

Solution 1: Surcharger l'opérateur « + » pour accepter « Fraction + int »

```
class Fraction
{
    private:
    int num;
    int den;

    public:
    Fraction();
    Fraction(int, int);
    void print();
    Fraction operator+(const Fraction&);
    Fraction operator+(const int&);
    ...
};
```

Mixer avec les autres types (de la droite)

Solution 1: Surcharger l'opérateur « + » pour accepter « Fraction + int »

```
...

Fraction Fraction::operator+(const int& i)
{
        cout << "Fraction + int" << endl;

        Fraction result;

        result.num = this->num + this->den * i;
        result.den = this->den;

        return result;
};
```

Mixer avec les autres types (de la droite)

```
... // all required lines
int main()
{
    Fraction f1(1, 2);
    int f2 = 1;

Fraction f3 = f1 + f2;

    cout << "f1: "; f1.print();
    cout << "f2: "; f2.print();
    cout << "f3: "; f3.print();

    return 0;
}
```

Output

```
Fraction(int, int) builds 1/2
Fraction + int
Fraction() builds 0/1
f1: 1/2
f2: 1
f3: 3/2
```

Mixer avec les autres types (de la droite)

Solution 2: Définir un constructeur pour la conversion de int vers Fraction

```
class Fraction
{
    private:
    int num;
    int den;

    public:
    Fraction(int num=0, int den=1);
    void print();
    Fraction operator+(const Fraction&);
    ...
};
```

Les arguments par défauts permettent de combiner trois constructeurs en un :
Fraction ()
Fraction (int)
Fraction (int, int)
Nous nous interressons en particulier au second, appelé aussi constructeur de conversion, pour sa capacité à convertir un int en Fraction

Mixer avec les autres types (de la droite)

```
... // all required lines
int main()
{
    Fraction f1(1, 2);
    int f2 = 1;

Fraction f3 = f1 + f2;

    cout << "f1: "; f1.print();
    cout << "f2: "; f2.print();
    cout << "f3: "; f3.print();

    return 0;
}
```

Output

```
Fraction(int, int) builds 1/2
Fraction + int
Fraction() builds 0/1
f1: 1/2
f2: 1
f3: 3/2
```

Mixer avec les autres types (de la gauche)

```
... // all required lines
int main()
{
    Fraction f1(1, 2);
    int f2 = 1;

Fraction f3 = f2 + f1;

    cout << "f1: "; f1.print();
    cout << "f2: "; f2.print();
    cout << "f3: "; f3.print();

    return 0;
}
```

On essaye d'appeler operator+ sur un entier f2 en passant une Fraction f1 en paramètre.

Est-ce que le type de base « int » accepte cette opération ?

Compiler output
Error

```
intPlusFrac1.cpp: In function 'int main()':
intPlusFrac1.cpp:11:21: error: no match for 'operator+' in 'f2 + f1'
intPlusFrac1.cpp:11:21: note: candidates are:
...
```

Mixer avec les autres types (de la gauche)

Solution 1: Définir int + Fraction en respectant l'encapsulation

```
Fraction operator+(int i, Fraction f)
{
    cout << "int + Fraction" << endl;

    Fraction result;

    result.setNum(f.getNum() + f.getDen() * i);
    result.setDen(f.getDen());

    return result;
}
```

Nous ne sommes pas dans le code (ou la portée) de la classe Fraction.

Nous ne pouvons pas accéder aux membres privées.

Mixer avec les autres types (de la gauche)

```
... // all required lines
int main()
{
    Fraction f1(1, 2);
    int f2 = 1;

Fraction f3 = f2 + f1;

    cout << "f1: "; f1.print();
    cout << "f2: "; f2.print();
    cout << "f3: "; f3.print();

    return 0;
}
```

Output

```
Fraction(int, int) builds 1/2
int + Fraction
Fraction() builds 0/1
f1: 1/2
f2: 1
f3: 3/2
```

Les classe et les fonctions amies

- Les opérateurs des autres classes (qui manipuleront la classe Fraction) ne pourront pas accéder au membres privés de la classe Fraction (il est obligatoire de respecter l'encapsulation).

- Comment faire pour faciliter l'écriture des ces méthodes spéciales et donner un accès aux membres privés pour ces méthodes « amies »?

- C'est le rôle du mot clé C++ : **`friend`**

Les classe et les fonctions amies

- En C++, il est possible d'autoriser une fonction (pas nécessairement un opérateur) ou toute une classe à accéder aux membres privés d'une classe C,
  en les déclarant **friend**

```
class C
{
    ...
    public:
    ...
friend class classeAmie;        // une classe amie
                                // Toutes les méthodes de classeAmie sont amies

    friend return_type function_name(...); // une function amie
    ...
};
```

Mixer avec les autres types (de la gauche)

Solution 2: Définir int + Fraction en tant qu'opérateur ami

```
class Fraction
{
    private:
    int num;
    int den;

    public:
    Fraction(int num=0, int den=1);
    void print();
    Fraction operator+(const Fraction&);
    friend Fraction operator+(int i, const Fraction& f);
};
```

Mixer avec les autres types (de la gauche)

Solution 2: Définir int + Fraction en tant qu'opérateur ami

```
...

Fraction operator+(int i, const Fraction& f)
{
    cout << "int + Fraction" << endl;

    Fraction result;

    result.num = f.num + f.den * i;
    result.den = f.den;

    return result;
}
```

Nous ne sommes toujours à l'extérieur de la classe Fraction.

Avec la possibilité d'accéder aux membres privés

Les opérateurs de conversion

Test

```
... // all required lines

int main()
{
    Fraction f(3, 2);
    int i = f;

    cout << "f: "; f.print();
    cout << "i: " << i << endl;

    return 0;
}
```

Pour intégrer encore plus notre classe Fraction, nous souhaitons pourvoir la convertir en int.

Exemple : convertir 6/3 en entier (2)
ou convertir 3/2 en entier (1, avec perte)

Compiler output
Error

```
fracToInt.cpp: In function 'int main()':
fracToInt.cpp:8:11: error: cannot convert 'Fraction' to 'int' in
initialization
```

Principe

Le constructeur de conversion permet de convertir tout `type` en Fraction.

Syntaxe :

```
Fraction(type)
```

⚠ Ne pas confondre ≠

L'opérateur de conversion permet de convertir Fraction en tout type "possible".

Syntaxe :

```
operator type()
```

Implémentation

Fraction.h updated

```
class Fraction{
    private:
        int num;
        int den;

    public:
        Fraction(int num=0, int den=1);
        void print();
        Fraction operator+(const Fraction&);
        friend Fraction operator+(float, const Fraction&);
        operator int();
};
```

Permet de convertir int → Fraction

Permet de convertir Fraction → int

Implémentation

Mise à jour de Fraction.cpp

```
...

Fraction::operator int()
{
        cout << "Fraction -> int" << endl;

        return num/den;  // renvoie la division entière
}
```

Résultat

```
... // all required lines

int main()
{
    Fraction f(3, 2);
    int i = f;

    cout << "f: "; f.print();
    cout << "i: " << i << endl;

    return 0;
}
```

Output

```
Fraction(int, int) builds 3/2
Fraction -> int
f: 3/2
i: 1
```

A voir dans les TP:

- Surcharge de l'opérateur d'affectation (=)
- Surcharge de l'opérateur accès à un tableau ([])
- Surcharge des opérateurs de comparaison (==, <, >, …)
- Surcharge des opérateurs de gestion de flux (<<, >>)