


Aux deux chapitres précédents, nous avons introduit les principes de base des classes en C++. Les manipulations d'objets de classes (c.à.d. des instances) sont effectuées en invoquant des fonctions membres de la classe sur l'objet en question. Cette notion d'appels de fonctions membres devient encombrante pour certaines catégories de classes mathématiques, pour lesquelles il serait commode d'utiliser la riche série d'opérateurs prédéfinis, afin de spécifier les manipulations d'objets.



## Objectifs

---


- Redéfinir (surcharger) des opérateurs pour les classes
  - Savoir quand surcharger et comment surcharger
- Créer des classes surchargeant des opérateurs
  - Études de cas

2

---

Marco Lavoie 14728 ORD - Langage C++

Dans ce chapitre, nous étudions comment permettre aux opérateurs du C++ de travailler avec des objets de classes. Ce processus, appelé *surcharge des opérateurs*, permet une extension simple et naturelle du C++ grâce à ces nouvelles possibilités. Toutefois, il faut l'employer avec un soin méticuleux, puisqu'une mauvaise utilisation de la surcharge peut compliquer la compréhension d'un programme.



## Aperçu

---

- Fondements de la surcharge des opérateurs
- Restrictions de la surcharge des opérateurs
- Surcharge d'opérateurs via
  - Fonctions membres
  - Fonctions amies (**friend**)
- Études de cas
  - Classes `Chaine` et `Date`

3

---

Marco Lavoie 14728 ORD - Langage C++

Le C++ permet au programmeur de surcharger la plupart des opérateurs (tels que `<<`, `>>`, `+`, `-`, `*`, `/`, `=`, `+=`, `!=`, `->`, etc.), qui deviennent sensibles au contexte dans lequel on les emploie. Le compilateur génère le code approprié, en fonction de la manière d'utiliser l'opérateur.

Certains opérateurs du C++ peuvent être surchargés via l'implantation de fonctions membres particulières de la classe, alors que d'autres opérateurs requièrent l'implantation de fonctions conventionnelles déclarées amies (**friend**) de la classe. De plus, certains opérateurs peuvent être surchargés des deux façons (c.à.d. soit par une fonction membre ou par une fonction amie, au choix du programmeur).

Enfin, ce ne sont cependant pas tous les opérateurs du langage C++ qui peuvent être surchargés.

## Introduction

- Surcharger un opérateur consiste à définir comment une classe interagit avec un opérateur
- Exemples d'opérateurs surchargeables
  - Flux d'entrées/sorties (>> et <<)
  - Opérateurs arithmétiques (+, -, \*, /, %)
  - Opérateurs relationnels (==, !=, <, <=, etc.)
  - Opérateurs d'affectations (=, +=, /=, etc.)
  - Incrémentation et décrémentation (++ et --)
- Un même opérateur peut être surchargé de multiples fois pour une même classe

Marco Lavoie

14728 ORD - Langage C++

Certains opérateurs sont surchargés fréquemment, comme l'opérateur d'affectation et différents opérateurs arithmétiques tels que + et -. Il est également possible d'effectuer le même travail que les opérateurs surchargés par des appels de fonctions membres, bien que la notation des opérateurs soit souvent plus claire et concise.

## Fondements de la surcharge

- Les types prédéfinis interagissent de façon différents avec les opérateurs prédéfinis
  - Par exemple, l'opérateur d'addition (+) n'effectue pas le même travail avec deux `int` qu'avec deux `float`
- Nos classes peuvent définir comment elles interagissent avec les opérateurs
  - En définissant une fonction membre correspondant à l'opérateur
    - **Exemple** : `operator+()` définit ce que représente l'addition pour notre classe
  - Permet donc l'exploitation des opérateurs surchargés avec notre classe
    - **Exemple** : `objet3 = objet1 + objet2;`

Marco Lavoie

14728 ORD - Langage C++

Les opérateurs du langage C++ sont applicables aux types prédéfinis, tels que `int` ou `float`. Le comportement d'un opérateur varie cependant selon le type de données auquel il est appliqué.

Les programmeurs peuvent également se servir des opérateurs avec des types qu'ils ont définis, tels que des classes. Bien que le C++ ne permette pas la création de nouveaux opérateurs, il permet la surcharge de la plupart des opérateurs existants afin qu'ils puissent avoir une signification appropriée aux nouveaux types lors de leur utilisation avec des instances de classes. Il s'agit d'une des caractéristiques les plus puissantes du C++.

La surcharge d'un opérateur se présente sous la forme d'une définition classique de fonction, dont un en-tête et un corps, avec cette différence que l'on nomme la fonction du mot-clé `operator`, suivi du symbole de l'opérateur surchargé. Par exemple, la fonction `operator<<` surcharge l'opérateur de flux de sortie <<.

## Opérateurs déjà surchargés

- Lorsqu'on définit une classe, deux opérateurs sont automatiquement surchargés pour celle-ci
  - Opérateur d'affectation (=) : effectue une affectation membre par membre d'une instance à l'autre
  - Opérateur d'adresse (&) : retourne l'adresse de l'instance
- Ces deux opérateurs peuvent toutefois être de nouveau surchargés dans la classe

Marco Lavoie

14728 ORD - Langage C++

Pour utiliser un opérateur sur les objets d'une classe, on doit surcharger cet opérateur pour cette classe, mais il existe deux exceptions à cette règle. Premièrement, l'opérateur d'affectation (=) est disponible pour toute classe, sans surcharge explicite. Le comportement par défaut de l'opérateur d'affectation consiste en une *affectation membre à membre* des attributs de la classe. Nous verrons plus loin qu'une telle affectation par défaut des attributs membres d'un objet à l'autre est dangereuse lorsque ces membres sont des pointeurs.

Deuxièmement, l'opérateur d'adresse (&) peut aussi être employé sans surcharge avec les objets de toute classe : il retourne simplement l'adresse de l'objet en mémoire. L'opérateur d'adresse peut cependant être surchargé pour en changer le comportement, au besoin.



## Restrictions de la surcharge

7

- Opérateurs surchargeables

Opérateurs pouvant être surchargés

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

- La surcharge ne permet pas de modifier la priorité ou l'associativité d'un opérateur
- Il n'est pas possible de créer de nouveaux opérateurs via la surcharge
  - Ou de modifier la surcharge d'un opérateur pour un type prédéfini

Marco Lavoie

14728 ORD - Langage C++

La plupart des opérateurs du C++ sont susceptibles d'être surchargés. Ceux-ci sont énumérés dans le tableau ci-contre.

La surcharge d'un opérateur ne modifie pas sa priorité ni son associativité, car ceci pourrait porter à confusion le programmeur exploitant ces opérateurs pour certains objets de classe. De plus, il est impossible de changer le nombre d'opérandes qu'accepte un opérateur. Les opérateurs unaires (tels que ++, -- et !) demeurent des opérateurs unaires, et ceux binaires demeurent binaires quels que soient leurs surcharges. Notez qu'il n'est pas possible de surcharger l'opérateur ternaire ? : .

Il n'est pas possible de créer de nouveaux opérateurs; seuls ceux qui existent peuvent être surchargés. Ce fait interdit au programmeur d'utiliser des notations populaires mais non disponibles en C++, comme l'opérateur d'élévation à une puissance \*\* des langages *FORTAN* et *BASIC*.



## Fonctions de surcharge

8

- Les fonctions de surcharge peuvent être membres et/ou amies
  - Les opérateurs (), [], -> ainsi que ceux d'affectation (=, +=, etc.) doivent être surchargés par des fonctions membres de la classe
  - Les opérateurs >> et << doivent être surchargés par des fonctions ordinaires déclarées amies (**friend**) de la classe
  - Les autres opérateurs (+, -, ==, !=, etc.) peuvent être surchargés par des fonctions membres ou amies, au choix

Marco Lavoie

14728 ORD - Langage C++

Les fonctions d'opérateurs peuvent être des fonctions membres ou non membres de classe; des fonctions non membres sont généralement déclarées amies (**friend**) de la classe pour des raisons d'accès aux attributs membres privés.



## Fonctions de surcharge (suite)

9

- Fonction membre ou amie?
  - Tout dépendant du type de l'opérand gauche
    - Le compilateur convertit l'invocation de l'opérateur en invocation de fonction de surcharge
    - Exemples (où c1 et c2 sont de type Classe)
 

|         |         |                    |    |                     |
|---------|---------|--------------------|----|---------------------|
| c1 + c2 | devient | c1.operator+( c2 ) | ou | operator+( c1, c2 ) |
| c1 + 4  | devient | c1.operator+( 4 )  | ou | operator+( c1, 4 )  |
| 4 + c1  | devient | operator+( 4, c1 ) |    |                     |

Impossible d'invoker **4.operator+( c1 )** car on ne peut pas ajouter une fonction de surcharge au type **int** (qui n'est pas une classe)
    - Si l'opérand gauche est de type **Classe**, une fonction membre de **Classe** peut surcharger l'opérateur; sinon la surcharge doit être fait via une fonction amie

Marco Lavoie

14728 ORD - Langage C++

Les fonctions membres surchargeant un opérateur utilisent implicitement le pointeur **this** pour accéder à l'objet de leur propre classe correspondant à l'argument gauche de l'opérateur (dans le cas des opérateurs binaires). Le seul paramètre de ces fonctions membres reçoit l'argument à droite de l'opérateur.

Les deux arguments de classe doivent être énumérés explicitement dans un appel à un opérateur surchargé par une fonction non membre.

Lors de la surcharge des opérateurs (), [], -> ou de tout autre opérateur d'affectation (p.ex. +=, /=, etc.), la fonction de surcharge de l'opérateur doit être déclarée comme membre de la classe. Pour les autres opérateurs, les fonctions de surcharge d'opérateur peuvent être membre ou non membre, au choix du programmeur.



## Fonctions de surcharge (suite)

10

- Complexité de la syntaxe
  - La syntaxe relative à la surcharge des opérateur est complexe car
    - Plusieurs opérateurs peuvent être surchargés
      - Et ceux-ci ont des fonctionnalités divergentes
    - Un même opérateur peut souvent être surchargé de différentes façons (ex: fonction membre ou amie)
  - Conséquemment il est illusoire de tenter de mémorier la syntaxe de surcharge
    - L'important est de comprendre les principes de surcharge et les subtilités de la syntaxe

Marco Lavoie

14728 ORD - Langage C++

Lorsqu'une fonction d'opérateur est mise en place sous la forme d'une fonction membre, l'opérande situé le plus à gauche (ou l'unique opérande dans le cas des opérateurs unaires) doit être un objet de la classe ou une référence à un objet de la classe associée à l'opérateur. Si l'opérande gauche doit représenter un objet d'une classe différente ou d'un type prédéfini, cette fonction d'opérateur doit être mise en place sous forme de fonction non membre de la classe. Elle doit cependant être déclarée amie de la classe si elle doit accéder aux attributs membres privés de cette dernière.

Les opérateurs de flux << et >> sont des exemples d'opérateurs ne pouvant être implantés que via des fonctions non membres car l'opérande à gauche de l'opérateur n'est généralement pas du type de la classe surchargeant l'opérateur.



## Surcharge d'entrées/sorties

11

- Une classe peut surcharger les opérateurs de flux >> et <<
  - Permet de contrôler l'affichage des instances de la classe
  - Exemple d'utilisation

```
Date d;
cout << "Entrez une date: ";
cin >> d;
cout << d;
```

```
C:\>date.exe
Entrez une date: 22/11/2002
22 novembre 2002
C:\>
```

- Puisque le programmeur écrit les fonctions du surcharge, il peut contrôler la façon de lire et d'écrire une instance de `Date`

Marco Lavoie

14728 ORD - Langage C++

Le C++ permet les manipulations d'entrée et de sortie des types prédéfinis en utilisant l'opérateur d'extraction de flux >> et l'opérateur d'insertion de flux <<. Ces opérateurs sont surchargés pour ces types dans <iostream>.

Les opérateurs de flux peuvent aussi être surchargés afin d'effectuer des entrées et des sorties pour les objets de classes définies par le programmeur. L'exemple ci-contre montre la surcharge de ces opérateurs pour la classe `Date`. Ce code présume que les dates sont entrées correctement au clavier; il est cependant possible d'écrire une version de la surcharge qui incorpore la détection d'erreurs.



## Surcharge d'entrées/sorties (suite)

12

- Arguments de l'opérateur
- ```
cin >> d;
```

Instance de  
la classe `istream`

Instance de  
la classe `Date`
- Donc la fonction surchargeant l'opérateur >> doit avoir deux paramètres de type approprié
    - Premier paramètre : `istream&`
    - Deuxième paramètre : `Date&`

Paramètre référence essentiel car la fonction doit modifier l'argument correspondant (i.e. un *alias*)

Notez que les fonctions de surcharge exploitent toujours des paramètres référence (&) afin de manipuler les arguments

Marco Lavoie

14728 ORD - Langage C++

Pour comprendre comment une fonction de surcharge est invoquée, il faut étudier le contexte dans lequel l'opérateur est utilisé. Dans l'exemple ci-contre, l'opérateur d'extraction de flux >> extrait (ou lit) une valeur du flux `cin` afin de la stocker dans la variable `d`, de type `Date`. On note que >> est un opérateur binaire, appliqué à un objet de type `istream` à sa gauche et à un objet de type `Date` à sa droite. Pour que `cin` sache comment afficher un objet de type `Date`, il faut surcharger l'opérateur >> pour la classe `istream` à l'aide d'une fonction nommée `operator>>` recevant les deux arguments en paramètre.

Notez que les deux paramètres de `operator>>` doivent être des références aux arguments car la fonction doit modifier ces arguments (c.à.d. extraire une valeur du paramètre de type `istream` et affecter celle-ci à l'argument de type `Date`).



## Surcharge d'entrées/sorties (suite)

13

- Chaînage de l'opérateur (de gauche à droite)

```
cin >> d1 >> d2;
```

Doit retourner `cin` de sorte que le chaînage (i.e. lire dans `d2`) soit supporté

- Donc la fonction surchargeant l'opérateur `>>` doit retourner le même flux d'entrées que celui obtenu via le premier paramètre  
– Type de valeur de retour : `istream&`

Le type de retour des fonctions de surcharge d'opérateurs enchainables est toujours une référence (&) afin de retourner la même instance que celle obtenue via paramètre

Marco Lavoie

14728 ORD - Langage C++

Une autre caractéristique commune à plusieurs opérateurs du C++, tels que les opérateurs arithmétiques, les opérateurs logiques et les opérateurs de flux, est le *chaînage* de ceux-ci, qui consiste à appliquer l'opérateur successivement à plusieurs opérandes (p.ex. `1 + 2 + 3 + 4`). Pour supporter cette caractéristique, les fonctions de surcharge de ces opérateurs doivent retourner l'objet de classe surchargeant l'opérateur afin que la prochaine invocation implique ce même objet comme premier argument.

Dans l'exemple ci-contre, la surcharge de `>>` pour la classe `istream` doit retourner l'objet de type `istream` (c.à.d. `cin` dans l'exemple) afin que la prochaine invocation de l'opérateur (c.à.d. `>> d2` dans l'exemple) ait ce même objet `istream` (c.à.d. `cin`) comme opérande gauche.



## Surcharge d'entrées/sorties (suite)

14

- Surcharge de l'opérateur `>>` pour lire des dates

```
istream& operator>>( istream& istr, Date& date ) {
    char sep;
    // pour lire les barres obliques
    istr >> date.jour >> sep >> date.mois >> sep >> date.annee;
    return istr;
}
// Retourne le même flux d'entrées
```

- À noter que c'est une fonction ordinaire (i.e. non membre de la classe `Date`)  
– Elle doit donc être déclarée *amie* de `Date` afin d'avoir accès aux attributs membres privés

Marco Lavoie

14728 ORD - Langage C++

La fonction de surcharge de l'opérateur d'extraction de flux `operator>>` prend comme arguments une référence `istream` nommée `istr` et une référence `Date` nommée `date`, et retourne une référence `istream`. La fonction d'opérateur `operator>>` sert ici à entrer des dates de la forme `jj/mm/aaaa` dans des objets de la classe `Date`. Lorsque le compilateur voit l'expression

```
cin >> d;
```

dans le code client, il génère l'invocation de fonction

```
operator>>( cin, d );
```

Lorsque l'appel est effectué, le paramètre de référence `istr` devient un alias de `cin`, tandis que le paramètre de référence `date` devient un alias de `d`. La fonction d'opérateur lit de `istr` (c.à.d. l'argument `cin`) les trois parties de la date, incluant les barres obliques les séparant, dans les membres `jour`, `mois` et `annee` de l'objet `date` (c.à.d. l'argument `d`).

La fonction retourne la référence `istr` pour permettre le chaînage des opérations d'entrées sur les objets `Date` avec d'autres types de données.



## Surcharge d'entrées/sorties (suite)

15

- Mise à jour de la déclaration de `Date`

```
class Date {
    friend istream& operator>>( istream&, Date& );
public:
    Date( int, int, int );
private:
    int jour, mois, annee;
};
```

- L'opérateur de sorties (`<<`) doit être similairement surchargé via une fonctions amie

Marco Lavoie

14728 ORD - Langage C++

Les fonctions `operator>>` et `operator<<` sont des fonctions non membres déclarées *friend* de la classe `Date`, ces opérateurs doivent être non membres puisque l'objet de la classe `Date` apparaît dans chaque cas comme opérande droit de l'opérateur; l'opérande de la classe doit apparaître à la gauche afin de surcharger l'opérateur comme fonction membre de la classe de l'objet en question.

Les opérateurs d'entrée et de sortie surchargés se déclarent comme *friend* s'ils nécessitent l'accès directement à des membres non `public` de la classe.



## Surcharge d'entrées/sorties (suite)

16

- Surcharge de l'opérateur <<

```
ostream& operator<<( ostream& ostr, const Date& date ) {
    ostr << date.jour << ' ';
    switch ( date.mois ) {
        case 1: ostr << "janvier"; break;
        case 2: ostr << "février"; break;
        case 3: ostr << "mars"; break;
        case 4: ostr << "avril"; break;
        case 5: ostr << "mai"; break;
        case 6: ostr << "juin"; break;
        case 7: ostr << "juillet"; break;
        case 8: ostr << "août"; break;
        case 9: ostr << "septembre"; break;
        case 10: ostr << "octobre"; break;
        case 11: ostr << "novembre"; break;
        case 12: ostr << "décembre"; break;
    }
    ostr << ' ' << date.annee;
    return ostr;
}
```

*Notez le paramètre const, car la fonction ne modifie pas le contenu de l'argument*

Marco Lavoie

14728 ORD - Langage C++

Similairement à la surcharge de l'opérateur d'extraction de flux >>, l'opérateur d'insertion de flux << est surchargé par la fonction non membre `operator<<` déclarée amie de la classe de l'objet servant d'opérande gauche de l'opérateur. Dans l'exemple ci-contre, la fonction de surcharge `operator<<` affiche sous forme textuelle des objets de type `Date`; elle doit conséquemment accéder aux attributs membres privés de la classe `Date` pour insérer dans la référence de flux de sortie `ostr` la date représentée par l'objet `date`.

Notez également que la référence `Date` dans la liste des paramètres d'`operator<<` est `const`, puisque l'objet `date` n'est pas modifié par la fonction de surcharge. Remarquez par contre que la référence `Date` dans la liste de paramètres d'`operator>>` est non `const`, puisque l'objet `date` doit être modifié pour stocker la date entrée dans l'objet.

## Exercice 8.1

17

- Récupérez le code source distribué par l'instructeur (solution du Devoir #5)
  - Surchargez les opérateurs d'entrées/sorties (>> et <<) afin de lire et afficher en format fractionnel (ex: 3/4)
  - Supprimez les anciennes fonctions d'affichage définies dans la classe
- N'oubliez pas les conventions d'écriture
- Soumettez votre projet selon les indications de l'instructeur

Marco Lavoie

14728 ORD - Langage C++

Le code source permettant de lire et d'afficher un objet de type `Rationnel` vous est déjà fourni. Votre tâche consiste donc à déplacer ce code respectivement dans les nouvelles fonctions non membres `operator>>` et `operator<<`, déclarées amies (`friend`) de la classe `Rationnel`.

## Opérateurs arithmétiques

18

- Considérons l'exemple suivant consistant à additionner deux heures

```
Temps t1( 12, 23, 34 ), t2( 1, 17, 8 );
cout << t1 + t2;
```

- c.à.d. ajouter 1h 17m 8s à 12h 23m, 34s
- L'opérateur + peut être surchargé de deux façons
  - Fonction amie (comme les opérateurs >> et <<)
  - Fonction membre de la classe `Temps`

Marco Lavoie

14728 ORD - Langage C++

La plupart des opérateurs binaires, tels que ceux d'arithmétique, peuvent être surchargés par une fonction membre avec un paramètre ou par une fonction non membre avec deux paramètres (tels que `operator>>` et `operator<<`).

Pour présenter la surcharge d'opérateurs binaires, nous présentons un exemple de surcharge de l'opérateur arithmétique d'addition pour la classe `Temps`. L'addition de deux objets de type `Temps` consiste à produire un nouvel objet `Temps` dont les attributs membres sont initialisés à la somme des deux opérandes fournis à l'opérateur. Dans l'exemple ci-contre, l'instruction

```
cout << t1 + t2;
```

est transformée en

```
operator<<( cout, operator+( t1, t2 ) );
```

Le nouvel objet résultant de l'addition des objets `t1` et `t2` sera initialisé à 13h 40m 42s.



## Opérateurs arithmétiques (suite)

19

- Surcharge via fonction amie
  - Convertir les temps en secondes, en faire la somme puis reconvertir celle-ci en heures, minutes et secondes

```
Temps operator+( const Temps& gauche, const Temps& droite ) {
    Temps resultat;
    int  secG = gauche.heure * 3600 + gauche.minute * 60 + gauche.seconde;
    int  secD = droite.heure * 3600 + droite.minute * 60 + droite.seconde;
    int  secRes = secG + secD;

    resultat.heure = secRes / 3600; // extraire les heures
    secRes = resultat.heure % 3600; // secondes restantes

    resultat.minutes = secRes / 60; // extraire les minutes restantes
    secRes = resultat.minutes % 60; // secondes restantes

    resultat.seconde = secRes;

    return resultat; // retourner l'heure résultante
}
```

Marco Lavoie

14728 ORD - Langage C++

La fonction non membre `operator+` ci-contre additionne ses deux paramètres en les transformant en secondes, puis retransforme la somme des secondes en heures, minutes et secondes pour un nouvel objet `Temps` nommé `resultat`.

Notez les deux paramètres références de la fonction de surcharge qui sont désignés `const` puisque ceux-ci ne sont pas modifiés par la fonction. L'avantage de désigner ces paramètres constants est de permettre l'application de l'opérateur d'addition à des opérandes de type `const Temps`, c'est-à-dire des objets constants.



## Opérateurs arithmétiques (suite)

20

- Notez l'en-tête de la fonction de surcharge

```
Temps t1( 12, 23, 34 ), t2( 1, 17, 8 );
cout << t1 + t2;
```

```
Temps operator+( const Temps& gauche, const Temps& droite ) {
    Temps resultat;
    ...
    return resultat;
}
```

*Retourne une nouvelle instance de Temps que cout peut afficher via une surcharge de l'opérateur <<*

- Particularité de l'opérateur : l'argument de gauche (`gauche`) est de même classe que la valeur de retour (i.e. type `Temps`)
  - L'opérateur peut dans ces cas être surchargé via une fonction membre de la classe

Marco Lavoie

14728 ORD - Langage C++

Similairement à l'exemple ci-contre, pour les objets `t1`, `t2` et `t3` de type `Temps`, l'instruction

```
t3 = t1 + t2;
```

est transformée en

```
operator=( t3, operator+( t1, t2 ) );
```

par le compilateur. L'invocation de la fonction de surcharge `operator+` retourne un nouvel objet de type `Temps` (c.à.d. l'objet correspondant à la variable locale `resultat` dans la fonction). Ce nouvel objet est ensuite affecté à l'objet `t3` via l'invocation de l'opérateur d'affectation `=` qui effectue une affectation membre à membre des attributs d'un objet à l'autre.

Nous verrons plus loin comment surcharger explicitement l'opérateur d'affectation pour une classe afin de modifier le comportement de l'opérateur `=`.



## Opérateurs arithmétiques (suite)

21

- Surcharge via fonction membre de la classe
  - Le pointeur `this` remplace l'argument de gauche
    - En d'autre mots, `operator+( )` s'exécute pour l'opérand gauche, et l'opérand droite est obtenu via le paramètre

```
Temps t1( 12, 23, 34 ), t2( 1, 17, 8 );
cout << t1 + t2;
```

```
Temps Temps::operator+( const Temps& droite ) const {
    Temps resultat;
    int  secG = this->heure * 3600 + this->minute * 60 + this->seconde;
    int  secD = droite.heure * 3600 + droite.minute * 60 + droite.seconde;
    int  secRes = secG + secD;

    ...

    return resultat; // retourner l'heure résultante
}
```

Marco Lavoie

14728 ORD - Langage C++

Lorsque la valeur retourne par une fonction de surcharge d'un opérateur binaire est de même classe que l'opérande gauche de l'opérateur (en d'autres mots, lorsque le type de retour d'une fonction de surcharge non membre implique la même classe que celle du type de son premier paramètre), l'opérateur peut être aussi surchargé par une fonction membre de la classe :

```
Temps operator+(const Temps&, const Temps&);
```

Dans l'exemple ci-contre, l'instruction `t1 + t2` est transformée par le compilateur en une des deux formes d'invocation suivantes : `operator( t1, t2 )` ou `t1.operator( t2 )`, selon la disponibilité d'une ou l'autre de ces surcharges. Dans le cas de la seconde surcharge (c.à.d. via une fonction membre de la classe `Temps`), l'opérande gauche de l'opérateur (c.à.d. `t1`) devient l'objet pour lequel la fonction membre `operator+` est invoquée et dont l'adresse est accessible via le pointeur `this` dans la fonction membre.

## Opérateurs arithmétiques (suite)

22

- En résumé, les opérateurs arithmétiques sont surchargés sous deux formes alternatives

```
class Classe { // Surchargées via fonctions amies
    friend Classe operator+( const Classe&, const Classe& );
    friend Classe operator-( const Classe&, const Classe& );
    friend Classe operator*( const Classe&, const Classe& );
    friend Classe operator/( const Classe&, const Classe& );
    friend Classe operator%( const Classe&, const Classe& );
    ...
};

class Classe { // Surchargées via fonctions membres
public:
    Classe operator+( const Classe& ) const;
    Classe operator-( const Classe& ) const;
    Classe operator*( const Classe& ) const;
    Classe operator/( const Classe& ) const;
    Classe operator%( const Classe& ) const;
    ...
};
```

Marco Lavoie

14728 ORD - Langage C++

Le choix de la façon de surcharger ces opérateurs binaires est laissé au programmeur. Cependant une bonne pratique de programmation en C++ consiste à privilégier la surcharge d'opérateurs sous forme de fonctions membres, soit la deuxième alternative dans l'exemple ci-contre. Il n'y a cependant aucun avantage de privilégier une forme à l'autre du point de vue performances.

Même si un opérateur binaire peut être surchargé sous forme de fonction membre et de fonction non membre, une seule surcharge de l'opérateur doit être fournie à une classe. En effet, si les deux formes de surcharge de `operator+` sont fournies à la classe `Temps`, le compilateur ne saura pas laquelle invoquer lorsque l'opérateur `+` est appliqué à deux objets de type `Temps`.

## Opérateurs arithmétiques (suite)

23

- Un opérateur arithmétique peut être surchargé de multiples fois

- En autant que chaque fonction de surcharge ait une signature distincte

- Exemple : ajouter des secondes : `Temps t1( 12, 23, 34 );`  
`cout << t1 + 5;`

```
Temps Temps::operator+( const int droite ) const {
    Temps resultat;
    int secG = this->heure * 3600 + this->minute * 60 + this->seconde;
    int secRes = secG + droite;

    ...

    return resultat; // retourner l'heure résultante
}
```

Marco Lavoie

14728 ORD - Langage C++

Un opérateur peut être surchargé plus d'une fois pour une même classe, en autant que ces surcharges aient des signatures distinctes. Dans l'exemple ci-contre, la classe `Temps` dispose d'une seconde surcharge de l'opérateur `+` afin de permettre d'ajouter un entier à un objet de type `Temps`. Cette surcharge aurait tout aussi bien pu être effectuée sous forme de fonction non membre :

```
Temps operator+(const Temps&, const int droite);
```

Une autre raison de choisir une fonction non membre afin de surcharger un opérateur est de permettre à cet opérateur de devenir commutatif, c'est-à-dire que l'opérande gauche de l'opérateur soit d'un type autre que celui de la classe. Ainsi, la surcharge

```
Temps operator+(const int droite, const Temps&);
```

permet d'invoquer l'opérateur d'addition en inversant ses opérandes (p.ex. `5 + t1`). Notez que cette surcharge peut seulement être fait via une fonction non membre car le premier paramètre n'est pas de type `Temps`.

## Opérateurs arithmétiques (suite)

24

- Une fonction de surcharge peut même en invoquer une autre au besoin

```
Temps Temps::operator+( const Temps& droite ) const {
    int secD = droite.heure * 3600 + droite.minute * 60 + droite.seconde;
    return *this + secD; // invoque operator+( const int )
}
```

- Il est fréquent que des fonctions de surcharge d'une classe fassent appel à d'autres fonctions de surcharge de cette même classe

- Ça évite de dupliquer du code source, facilitant ainsi la maintenance

Marco Lavoie

14728 ORD - Langage C++

Pour des raisons de facilité de maintenance, il est fréquent qu'une fonction de surcharge d'un opérateur en invoque une autre pour effectuer le travail. Dans l'exemple ci-contre, la surcharge d'addition de deux objets de type `Temps` transforme l'objet en opérande droite de l'opérateur en secondes, puis invoque l'autre surcharge de l'addition permettant d'ajouter des secondes à un objet de type `Temps`.

Les bénéfices de cette stratégie d'implantation sont multiples, le plus important étant que le code source effectuant l'addition est principalement retrouvé dans une seule surcharge de l'opérateur `+`, facilitant ainsi le débogage et la maintenance du code source de la classe, ce code n'étant pas dupliqué dans plusieurs fonctions.





## Exercice 8.2

25

- Poursuivez l'exercice 8.1 (classe **Rationnel**)
  - Surchargez l'opérateur arithmétique d'addition ( + ) sous deux formes afin de permettre l'addition d'une fraction à un entier (ex: `f1 + 5`) et vice-versa (ex: `5 + f1`)
- N'oubliez pas les conventions d'écriture
- Soumettez votre projet selon les indications de l'instructeur

Marco Lavoie

14728 ORD - Langage C++

Référez-vous à l'exemple des surcharges de l'opérateur d'addition de la classe **Temps** des pages précédentes pour solutionner cet exercice. En particulier, vous aurez trois formes de surcharges de l'opérateur dans la classe **Rationnel** :

1. l'addition de deux objets de type **Rationnel**,
2. l'addition d'un entier à un objet de type **Rationnel**, et
3. l'addition d'un objet de type **Rationnel** à un entier.

Deux de ses trois surcharges peuvent être implantées sous forme de fonction membre de **Rationnel**, ou de fonction non membres amies de **Rationnel**; l'autre surcharge pouvant seulement être implantée sous forme de fonction non membre.



## Opérateurs d'affectation

26

- Par défaut, une classe dispose d'un opérateur d'affectation (=) membre à membre
  - La valeur de chaque attribut membre est copiée

```
Temps t1( 12, 23, 34 ), t2;
t2 = t1;
cout << t2; // affiche 12:23:34
```
- On peut cependant surcharger explicitement l'opérateur =
- Les autres opérateurs d'affectation (+=, -=, etc.) ne sont pas automatiquement surchargés, mais on peut les surcharger

Marco Lavoie

14728 ORD - Langage C++

Comme mentionné au chapitre 7, toute classe dispose implicitement d'une surcharge de l'opérateur d'affectation = qui effectue une affectation membre à membre des attributs d'un objet à l'autre. Ainsi, l'invocation `t2 = t1` dans l'exemple ci-contre est traduite par le compilateur à `operator=( t1, t2 )`, qui correspond à la surcharge implicite fournie par le compilateur à la classe **Temps**.

Même si le compilateur fournit une surcharge implicite de l'opérateur d'affectation à la classe **Temps**, il est tout de même possible de « remplacer » cette surcharge par une surcharge explicite. Notez que seule l'opérateur d'affectation = est implicitement fournie aux classes; les autres opérateurs d'affectation doivent, eux, être explicitement surchargés.



## Opérateurs d'affectation (suite)

27

- Les opérateurs d'affectation se surchargent uniquement sous forme de fonctions membres
 

```
const Temps& Temps::operator=( const Temps& droite ) {
    // Éviter l'auto-affectation
    if ( &droite == this ) } On s'assure de ne pas affecter
                           } une instance à elle-même
    return *this;

    // Copier attribut à attribut
    this->heure = droite.heure;
    this->minute = droite.minute;
    this->seconde = droite.seconde;

    return *this;
}
```
- Toujours éviter l'auto-affectation lors de la surcharge de l'opérateur =
  - Essentiel pour les attributs dynamiques

Marco Lavoie

14728 ORD - Langage C++

Toute surcharge d'opérateurs d'affectation doit obligatoirement être faite sous forme de fonction membre. Il est en effet interdit de surcharger un opérateur d'affectation avec une fonction non membre amie, et ce même si l'opérande gauche est du même type que la classe visée.

La fonction `operator=` ci-contre teste l'*auto-affectation*. Si cette fonction détecte une tentative d'affecter un objet à lui-même, alors l'affectation est omise. L'*auto-affectation* est dangereuse et doit généralement être évitée dans les surcharges d'opérateurs d'affectation.

**Rappel** : l'opérateur d'affectation doit généralement permettre le chaînage :

```
int i, j, k;
i = j = k = 0;
```

Conséquemment, la surcharge de l'opérateur d'affectation doit retourner une référence à l'opérande gauche (c.à.d. `*this`) pour permettre ce chaînage.

## Opérateurs d'affectation (suite)

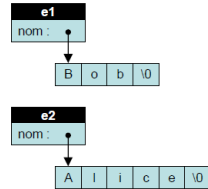
28

### Importance d'éviter l'auto-affectation

```
class Exemple {
public:
    Exemple( const char * n = NULL );
    const Exemple & operator=( const Exemple &);
private:
    char *nom;
};

Exemple::Exemple( const char *n ) {
    if ( n != NULL ) {
        nom = new char[ strlen( n ) + 1 ];
        strcpy( nom, n );
    }
    else
        nom = NULL;
}
```

```
int main {
    Exemple e1( "Bob" ), e2( "Alice" );
    return 0;
}
```



Marco Lavoie

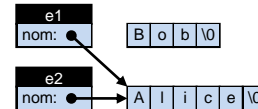
14728 ORD - Langage C++

La classe **Exemple** ci-contre exploite l'allocation dynamique de mémoire pour stocker la chaîne de caractères fournie en argument du constructeur (paramètre **n**) dans le bloc mémoire dont l'adresse est stockée dans le pointeur attribut membre **nom**.

Puisque la classe **Exemple** ne surcharge pas l'opérateur d'affectation **operator=**, celui par défaut fourni par le compilateur effectue une affectation attribut par attribut d'un objet à l'autre. Si l'affectation

```
e1 = e2;
```

est exécutée dans le code client (c.à.d. dans **main**), les deux pointeurs partageront le même bloc mémoire puisque l'adresse de **e2.nom** sera copiée dans **e1.nom**. L'état résultant des deux objets est corrompu, et le bloc mémoire stockant « Bob » est perdu à jamais car son adresse n'est plus accessible.



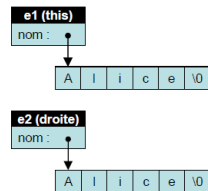
## Opérateurs d'affectation (suite)

29

### Importance d'éviter l'auto-affectation (suite)

```
const Exemple & Exemple::operator=( const Exemple & droite) {
    // Éviter l'auto-affectation
    if ( &droite == this )
        return *this;

    // Copier le nom de droite dans this
    delete [] this->nom;
    if ( droite.nom != NULL ) {
        nom = new char[ strlen( droite.nom ) + 1 ];
        strcpy( nom, droite.nom );
    }
    else
        nom = NULL;
    return *this;
}
```



Marco Lavoie

14728 ORD - Langage C++

Pour éviter l'anomalie décrite à la page précédente lors de l'affectation d'un objet de type **Exemple** à un autre objet du même type, nous surchargeons l'opérateur d'affectation = afin de gérer adéquatement les blocs mémoires lors de l'affectation.

La surcharge ci-contre libère le bloc mémoire alloué préalablement à **this** (correspondant à l'objet **e1** de l'invocation **e1.operator=(e2)** dans **main**). Un nouveau bloc mémoire est ensuite alloué à **e1**, puis le contenu du bloc mémoire alloué à **e2** y est copié (avec **strcpy**).

Notez que les premières instructions de la fonction de surcharge évitent l'auto-affectation en s'assurant que l'objet pointé par **this** et celui référencé par le paramètre **droite** ne sont pas le même objet (c.à.d. qu'ils n'ont pas la même adresse mémoire).

## Opérateurs d'affectation (suite)

30

### Importance d'éviter l'auto-affectation (suite)

```
const Exemple & Exemple::operator=( const Exemple & droite) {
    // Éviter l'auto-affectation
    if ( &droite == this )
        return *this;

    // Copier le nom de droite dans this
    delete [] this->nom;
    if ( droite.nom != NULL ) {
        nom = new char[ strlen( droite.nom ) + 1 ];
        strcpy( nom, droite.nom );
    }
    else
        nom = NULL;
    return *this;
}
```



L'appel à **strlen()** va échouer car le bloc mémoire à l'adresse pointée est libéré

Marco Lavoie

14728 ORD - Langage C++

Que se passerait-il si aucune détection d'auto-affectation n'était faite dans la fonction de surcharge? Si **\*this** et **droite** sont un même objet (tel que c'est le cas lors de l'invocation **e2.operator=(e2)** dans **main**), l'instruction

```
delete [] this->nom;
```

supprime le bloc mémoire associé à l'objet **e2**. La condition de la structure conditionnelle qui suit, **droite.nom != NULL**, étant vraie (car **e2.nom** contient toujours l'adresse du bloc mémoire supprimé, l'instruction suivante tente de créer un nouveau bloc mémoire en invoquant **strlen** sur le bloc mémoire supprimé, ce qui occasionne habituellement une défaillance du programme à l'exécution.

La détection de l'auto-affectation évite ce comportement erroné du programme en court-circuitant l'affectation si **\*this** et **droite** sont un même objet.

## Opérateurs d'affectation (suite)

31

- La surcharge des autres opérateurs d'affectation (`+=`, `+=`, `*=`, `/=` et `%=`) se fait de façon similaire

```
const Exemple & Exemple::operator+=( const Exemple & droite) {
    // Calculer la longueur du nom concaténé
    int totLen = ( this->nom != NULL ? strlen( this->nom ) : 0 ) +
                ( droite.nom != NULL ? strlen( droite->nom ) : 0 );

    // Concaténer les deux noms dans une nouvelle chaîne
    char *nouv = new char[ totLen + 1 ];
    nouv[ totLen ] = '\0';
    if ( this->nom != NULL ) strcpy( nouv, this->nom );
    if ( droite.nom != NULL ) strcat( nouv, droite->nom );

    // Remplacer le nom de this
    delete [] this->nom;
    nom = nouv;

    return *this;
}
```

Marco Lavoie

14728 ORD - Langage C++

Les autres opérateurs d'affectation peuvent être surchargés via une fonction membre de classe telle que `operator+=`. Dans l'exemple ci-contre surchargeant l'opérateur `+=` pour la classe `Chaine`, un nouveau bloc mémoire est alloué pour l'attribut `nom` de `this` (i.e. l'objet de type `Chaine` servant d'opérande gauche de `+=`), ce bloc ayant une taille correspondant aux tailles additionnées des blocs mémoires des deux opérandes de l'opérateur (c.à.d. la taille du bloc mémoire de `this->nom` additionnée à la taille de celui de `droite.nom`). Dans ce nouveau bloc mémoire est copié bout à bout le contenu de `this->nom` suivi de celui de `droite.nom`. En conclusion, ce nouveau bloc remplace l'ancien bloc de `this->nom`, qui est supprimé au préalable.

Notez que l'opérateur `+=` ci-contre autorise l'auto-affectation car celle-ci n'occasionne pas le comportement erroné rencontré par `operator=`, l'objet droite n'étant pas altéré dans `operator+=`.

## Opérateurs d'affectation (suite)

32

- Les opérateurs d'affectation peuvent être surchargés plusieurs fois
  - En autant que chaque version de surcharge ait une signature distincte
  - Exemple :

```
// Ajouter une chaîne à la fin de this (ex: c1 += c2);
const Chaine & Chaine::operator+=( const Chaine & );

// Ajouter un caractère à la fin de this (ex: c1 += 'x');
const Chaine & Chaine::operator+=( const char & );

// Ajouter un tableau de caractères à la fin de this (ex: c1 += "xyz");
const Chaine & Chaine::operator+=( const char [] );
```

Marco Lavoie

14728 ORD - Langage C++

Comme les autres opérateurs binaires tels que ceux arithmétiques vus précédemment, les opérateurs d'affectation peuvent être surchargés à de multiples reprises dans une même classe, en autant que chaque surcharge d'un même opérateur ait une signature distincte.

Notez que les surcharges d'opérateurs d'affectation retournent une référence de type `const` car en fait elles retournent l'objet pointé par le pointeur constant `this` (c.à.d. `return *this;`). De plus, puisque ces fonctions de surcharge doivent retourner l'objet pointé par `this`, et non une copie de cet objet, la valeur de retour doit être une référence (&) à l'objet. C'est ainsi que, dans l'exemple ci-contre, `this` étant de type `const Chaine *`, la valeur de retour des fonctions de surcharge (soit `*this`) doivent être `const Chaine &`.

## Opérateurs relationnels

33

- Les opérateurs relationnels peuvent être surchargés

```
Chaine s1( "table" ), s2( "chaise" );
if ( s1 < s2 ) cout << s1;
```

- Via des fonctions membres

```
bool Classe::operator==( const Classe & ) const;
bool Classe::operator!=( const Classe & ) const;
bool Classe::operator<( const Classe & ) const;
bool Classe::operator<=( const Classe & ) const;
bool Classe::operator>( const Classe & ) const;
bool Classe::operator>=( const Classe & ) const;
```

- Ou via des fonctions amies

```
friend bool operator==( const Classe &, const Classe & );
friend bool operator!=( const Classe &, const Classe & );
friend bool operator<( const Classe &, const Classe & );
friend bool operator<=( const Classe &, const Classe & );
friend bool operator>( const Classe &, const Classe & );
friend bool operator>=( const Classe &, const Classe & );
```

Marco Lavoie

14728 ORD - Langage C++

Les opérateurs relationnels (`<`, `<=`, `>`, `>=`, `==` et `!=`) peuvent être surchargés soient comme fonctions membres de classe ou comme fonction non membre amie de la classe. Ces fonctions doivent retourner une valeur booléenne indiquant le résultat de la comparaison des deux objets, ces derniers n'étant généralement pas altérés par l'opérateur, d'où le type des paramètres, des références constantes aux objets (p.ex. `const Classe &`).

Généralement, plusieurs de ces surcharges d'opérateurs relationnels invoquent d'autres opérateurs relationnels. Par exemple, la fonction de surcharge `operator!=` peut simplement retourner la négation du résultat de `operator==`. Il en est de même pour `operator<` qui peut retourner la négation du résultat de `operator>=`.

Notez qu'une classe n'a pas à surcharger tous les opérateurs relationnels. Il arrive fréquemment qu'un classe ne surcharge que les opérateurs `==` et `!=`.

## Opérateurs logiques

34

- Les opérateurs logiques peuvent aussi être surchargés

- L'opérateur `!` (négation) peut seulement être surchargé via une fonction membre

```
bool Classe::operator!() const;
```

- Les opérateurs `&&` et `||` peuvent être surchargés via une fonction membre ou une fonction amie

```
bool Classe::operator&&( const Classe & ) const;
bool Classe::operator|| ( const Classe & ) const;

friend bool operator&&( const Classe &, const Classe & );
friend bool operator|| ( const Classe &, const Classe & );
```

Marco Lavoie

14728 ORD - Langage C++

Comme les opérateurs relationnels, les opérateurs logiques (`&&`, `||` et `!`) peuvent aussi être surchargés pour une classe. Notez cependant que l'opérateur unaire de négation (`!`) peut seulement être surchargé par une fonction membre de classe, comme les autres opérateurs unaires surchargeable du C++.

## Opérateur d'indexage []

35

- L'opérateur d'indexage (`[]`) peut être surchargé

```
class Exemple {
public:
    Exemple( const char * = NULL );
    char & operator[]( int );
private:
    char *nom;
};

char & Exemple::operator[]( int indice ) {
    // Valider l'indice
    assert( indice >= 0 && indice < strlen( nom ) );
    // Retourne une référence
    return nom[ indice ];
}
```

- Le retour d'une référence permet l'affectation de caractères via l'opérateur

```
Exemple e("Gustave");
cout << e[ 4 ]; // Affiche o
e[ 4 ] = 'a'; // Gustave
```

Marco Lavoie

14728 ORD - Langage C++

L'opérateur d'indexation `[]` peut aussi être surchargés pour une classe sous forme de fonction membre de celle-ci. La fonction de surcharge reçoit en paramètre l'indice fourni en argument lors de l'invocation de l'opérateur.

Dans l'exemple ci-contre, l'instruction

```
cout << e[ 4 ];
```

est convertie par le compilateur en invocation suivante :

```
operator<<( cout, e.operator[]( 4 ) );
```

## Exemple : classe Chaîne

36

- Classe de manipulation de chaînes de caractères

```
class Chaîne {
    friend ostream &operator<<( ostream &, const Chaîne & );
    friend istream &operator>>( istream &, Chaîne & );
public:
    Chaîne( const char * = "" ); // constr. de conversion et par défaut
    Chaîne( const Chaîne & ); // constructeur de copie
    ~Chaîne(); // destructeur

    const Chaîne &operator=( const Chaîne & ); // affectation
    const Chaîne &operator+=( const Chaîne & ); // concaténation
    bool operator!() const; // Chaîne est-elle vide?
    bool operator==( const Chaîne & ) const; // teste s1 == s2
    bool operator<( const Chaîne & ) const; // teste s1 < s2

    // Teste s1 != s2
    bool operator!=( const Chaîne & droite ) const {
        return !( *this == droite ); }
};
```

Marco Lavoie

14728 ORD - Langage C++

À titre d'exemple de synthèse pour notre étude de la surcharge, nous construisons une classe supportant la création et la manipulation de chaînes de caractères, semblable à la classe `string` en Java (notez que la bibliothèque du langage C++ dispose aussi d'une classe `string`, que nous verrons plus tard).

La classe Chaîne dispose de deux constructeurs, soient

```
Chaîne( const char * = "" );
Chaîne( const Chaîne & );
```

Le premier est un *constructeur de conversion* et sert aussi de constructeur par défaut (puisque une valeur par défaut est fournie pour son paramètre). Ce constructeur convertit la chaîne fournie, de type `char *`, en un objet `Chaîne`.

Le second constructeur est un *constructeur de copie*, qui initialise un objet `Chaîne` en fabriquant une copie d'un objet `Chaîne` existant.



## Exemple : classe Chaine (suite)

```

// Teste s1 > s2
bool operator>( const Chaine &droite ) const { return droite < *this; }

// Teste s1 <= s2
bool operator<=( const Chaine &droite ) const { return !( *this > droite ); }

// Teste s1 >= s2
bool operator>=( const Chaine &droite ) const { return !( *this < droite ); }

char &operator[]( int );           // opérateur d'indice
const char &operator[]( int ) const; // opérateur d'indice
Chaine operator()( int, int );     // renvoie une sous-chaîne
int lectureLongueur() const;       // renvoie la longueur de la chaîne

private:
int longueur;           // longueur de la chaîne
char *sPtr;             // pointeur vers le début de la chaîne

void ajusterChaine( const char * ); // fonction utilitaire
};

```

Marco Lavoie

14728 ORD - Langage C++

Notre implantation d'un objet **Chaine** surcharge plusieurs opérateurs utiles aux codes clients, dont quelques opérateurs d'affectation ainsi que tous les opérateurs relationnels. De plus, elle surcharge deux versions de l'opérateur d'indexation, **operator[]**, ainsi que l'opérateur d'appel de fonction, **operator()**. Cette dernière surcharge, acceptant deux arguments entiers, permettent de spécifier l'emplacement de départ ainsi que la longueur d'une sous-chaîne à sélectionner à même un objet **Chaine** existant.

Notre classe **Chaine** possède un attribut membre privé **longueur** représentant le nombre de caractères dans la chaîne, à l'exclusion du caractère nul de fin de chaîne. La classe possède aussi un pointeur privé **sPtr** vers son espace de stockage alloué dynamiquement qui identifie la chaîne de caractères.

Finalement, la classe **Chaine** fournit un accesseur public à son attribut **longueur**, soit **lectureLongueur**.



## Exemple : classe Chaine (suite)

```

// Constructeur de conversion: convertit char * en Chaine
Chaine::Chaine( const char *s ): longueur( strlen( s ) ) {
    ajusterChaine( s ); // appelle la fonction utilitaire
}

// Constructeur de copie
Chaine::Chaine( const Chaine &copie ): longueur( copie.longueur ) {
    ajusterChaine( copie.sPtr ); // appelle la fonction utilitaire
}

// Destructeur
Chaine::~Chaine() {
    delete [] sPtr; // récupère la chaîne
}

// Cette Chaine est-elle inférieure à Chaine droite?
bool Chaine::operator<( const Chaine &droite ) const {
    return strcmp( sPtr, droite.sPtr ) < 0;
}

```

Marco Lavoie

14728 ORD - Langage C++

Les constructeurs de la classe **Chaine** exploitent un initialiseur pour initialiser l'attribut privé **longueur** à la longueur de la chaîne reçue en paramètre. L'attribut **sPtr** se voit allouer un bloc mémoire dynamiquement via l'invocation de la fonction membre privée **ajusterChaine**, qui crée le bloc mémoire et copie la chaîne fournie dans ce bloc.

Le destructeur de la classe, **~Chaine()**, utilise **delete** pour récupérer l'espace mémoire dynamiquement obtenue par **new** (dans **ajusterChaine**) afin de libérer l'espace occupé par la chaîne de caractères.

Tous les opérateurs relationnels sont surchargés dans **Chaine** de façons sensiblement semblables, en invoquant les fonctions de la bibliothèque **<cstring>** pour comparer le contenu des blocs mémoires pointés par l'attribut **sPtr** des deux objets comparés. Dans l'exemple ci-contre, **operator<** invoque **strcmp** pour comparer **this->sPtr** à **droite.sPtr**.



## Exemple : classe Chaine (suite)

```

// Surcharge operator=: évite l'auto-affectation
const Chaine &Chaine::operator=( const Chaine &droite ) {
    if ( &droite != this ) { // évite l'auto-affectation
        delete [] sPtr; // évite une fuite de mémoire
        longueur = droite.longueur; // nouvelle longueur de Chaine
        ajusterChaine( droite.sPtr ); // appelle la fonction utilitaire
    }
    else
        cerr << "Tentative d'affectation d'une chaîne ... elle-même\n";

    return *this; // permet des affectations en cascade
}

// Cette Chaine est-elle vide?
bool Chaine::operator!() const { return longueur == 0; }

// Cette Chaine est-elle égale à Chaine droite?
bool Chaine::operator==( const Chaine &droite ) const {
    return strcmp( sPtr, droite.sPtr ) == 0;
}

```

Marco Lavoie

14728 ORD - Langage C++

La fonction de surcharge de l'opérateur d'affectation, **operator=**, teste la possibilité d'une auto-affectation. S'il s'agit bien d'une auto-affectation, la fonction retourne simplement le contrôle puisque l'objet se représente déjà lui-même. Si cette vérification était omise, la fonction supprimerait immédiatement l'espace dans l'objet cible, ce qui provoquerait la perte de la chaîne, soit un exemple classique de *fuite de mémoire*.

S'il ne s'agit pas d'un cas d'auto-affectation, la fonction supprime l'espace, copie le champ **longueur** de l'objet source (**droite**) dans l'objet cible (**\*this**) tout en invoquant **ajusterChaine** afin de créer un nouvel espace mémoire pour l'objet cible. Finalement, **\*this** est retourné pour permettre des affectations en cascade.

L'opérateur de négation (!) est surchargé pour simplement permettre au code client de vérifier si l'objet **Chaine** contient une chaîne de caractères ou non.





## Exemple : classe Chaine (suite)

```

40
// Concaténation de l'opérande de droite à l'objet this
// et remisage dans l'objet this
const Chaine &Chaine::operator+=( const Chaine &droite ) {
    char *tempPtr = sPtr; // maintient pour supprimer
    longueur += droite.longueur; // nouvelle longueur de Chaine
    sPtr = new char[ longueur + 1 ]; // crée de l'espace
    assert ( sPtr != 0 ); // termine si mémoire non allouée
    strcpy ( sPtr, tempPtr ); // partie gauche de la nouvelle Chaine
    strcat ( sPtr, droite.sPtr ); // partie droite de la nouvelle Chaine
    delete [] tempPtr; // récupère l'ancien espace

    return *this; // permet des appels en cascade
}

// Renvoie une référence à un caractère dans une Chaine
// comme valeur gauche
char &Chaine::operator[]( int indice ) {
    // Premier test pour indice hors de portée
    assert ( indice >= 0 && indice < longueur );

    return sPtr[ indice ]; // crée une valeur gauche
}

```

Marco Lavoie 14728 ORD - Langage C++

La fonction membre `operator+=` surcharge l'opérateur d'affectation sous forme d'*opérateur de concaténation de chaînes*. Cette fonction crée d'abord un pointeur temporaire pour contenir la chaîne de caractères de l'objet courant jusqu'à ce que la mémoire associée puisse être supprimée. Elle calcule ensuite la longueur combinée de la chaîne concaténée, emploie `new` afin de réserver de l'espace mémoire pour la chaîne, vérifie si `new` a bien fonctionné via `assert` et copie la chaîne d'origine dans l'espace nouvellement alloué grâce à `strcpy`.

Finalement, la fonction utilise `strcat` afin de concaténer la chaîne de caractères de l'objet source (`droite`) dans le nouvel emplacement mémoire, se sert de `delete` pour récupérer l'espace occupé par la chaîne d'origine et retourne finalement `*this` comme élément `const Chaine` & pour permettre le chaînage d'opérateurs `+=`.



## Exemple : classe Chaine (suite)

```

41
// Renvoie une référence à un caractère dans une Chaine
// comme valeur droite
const char &Chaine::operator[]( int indice ) const {
    // Premier test pour indice hors de portée.
    assert ( indice >= 0 && indice < longueur );

    return sPtr[ indice ]; // crée une valeur droite
}

// Renvoie une sous-chaîne commençant à index et d'une
// longueur sousLongueur comme référence à un objet Chaine
Chaine Chaine::operator()( int index, int sousLongueur ) {
    // Assure qu'index est dans la plage
    // et que la longueur de sous-chaîne >= 0
    assert ( index >= 0 && index < longueur && sousLongueur >= 0 );

    // Déterminer la longueur de la sous-chaîne.
    int lng;
    if ( ( sousLongueur == 0 ) || ( index + sousLongueur > longueur ) )
        lng = longueur - index;
    else
        lng = sousLongueur;

    return Chaine( sPtr + index, lng );
}

```

Marco Lavoie 14728 ORD - Langage C++

La classe `Chaine` surcharge l'opérateur d'indexation `[]` de deux façons. Lorsque le compilateur lit une expression telle que `chaîne[0]`, il génère l'invocation `chaîne.operator[]( 0 )`, en utilisant la version appropriée d'`operator[]`, selon que la chaîne `chaîne` est `const` ou non. Ces fonctions retournent un `char &` pouvant agir à titre d'opérande gauche d'une affectation pour modifier le caractère désigné de l'objet `Chaine`.

L'opérateur d'appel de fonction surchargé, `operator()`, permet de sélectionner une sous-chaîne à partir d'un objet `Chaine` existant. Les deux paramètres entiers spécifient l'emplacement de départ ainsi que la longueur de la sous-chaîne sélectionnée à même l'objet `*this`. Si l'emplacement de départ se trouve hors de portée ou si la longueur de la sous-chaîne est négative, un message d'erreur est généré par `assert`. Par convention, pour une longueur de 0, la sous-chaîne est sélectionnée jusqu'à la fin de la chaîne d'origine.



## Exemple : classe Chaine (suite)

```

42
// Allouer un tableau temporaire pour la sous-chaîne et le caractère nul
// de terminaison
char *tempPtr = new char[ lng + 1 ];
assert( tempPtr != 0 ); // vérifier que l'espace est Bien alloué

// Copier sous-chaîne dans le tableau de caractères et compléter chaîne
strcpy( tempPtr, sPtr[ index ], lng );
tempPtr[ lng ] = '\0';

// Créer un objet de Chaine temporaire contenant la sous-chaîne
Chaine tempChaine( tempPtr );
delete tempPtr; // supprimer le tableau temporaire

return tempChaine; // renvoie une copie de la Chaine temporaire
}

// Renvoie la longueur de la chaîne
int Chaine::lectureLongueur() const { return longueur; }

```

Marco Lavoie 14728 ORD - Langage C++

La surcharge de l'opérateur d'appel de fonction `()` est efficace puisque les fonctions peuvent alors prendre des listes de paramètres arbitraires à la fois longues et complexes. Nous pouvons donc employer cette caractéristique pour nombre d'applications intéressantes.

La fonction membre `lectureLongueur` est un accesseur à l'attribut membre privé `longueur`, et ne fait donc que retourner la valeur stockée dans cet attribut.



## Exemple : classe Chaine (suite)

43

```
// Fonction utilitaire appelée par les constructeurs et
// par l'opérateur d'affectation
void Chaine::ajusterChaine( const char *chaine2 ) {
    sPtr = new char[ longueur + 1 ]; // alloue la mémoire, incluant le \0
    assert ( sPtr != 0 );           // termine si mémoire non allouée
    strcpy ( sPtr, chaine2 );       // copie le littéral dans l'objet
}

// Opérateur de sortie surchargé
ostream &operator<<( ostream &sortie, const Chaine &s ) {
    sortie << s.sPtr;
    return sortie; // permet la mise en cascade
}

// Opérateur d'entrée surchargé.
istream &operator>>( istream &entree, Chaine &s ) {
    char temp[ 100 ]; // tampon pour remiser entrée

    entree >> setw ( 100 ) >> temp;
    s = temp; // utilise l'opérateur d'affectation de classe Chaine
    return entree; // permet la mise en cascade
}
```

Marco Lavoie

14728 ORD - Langage C++

La fonction `ajusterChaine`, d'accès `private`, est une fonction utilitaire de la classe `Chaine` utilisée par les constructeurs pour initialiser l'attribut `sPtr` à une chaîne de caractères fournie. La fonction invoque `new` pour obtenir dynamiquement un espace mémoire juste assez grand pour stocker la chaîne de caractères fournie en paramètre, s'assure qu'un tel bloc mémoire fut bien obtenu (via un `assert`), puis copie la chaîne fournie dans ce bloc mémoire.

Les opérateurs de flux de la classe `Chaine` exploitent les capacités des objets `istream` et `ostream` de manipuler des chaînes de caractères données sous forme de tableaux de caractères, ce qu'est en réalité l'attribut membre `sPtr`.



## Exercice 8.3

44

- Poursuivez l'exercice 8.2 (classe `Rationnel`)
  - Surchargez l'opérateur d'affectation `+=` pour fractions et entiers (ex: `f2 += f1` et `f2 += 4`)
  - Surchargez les opérateurs relationnels `==` et `!=` pour fractions et entiers (ex: `f2 == f1` && `f3 != 7` || `6 == f4`)
- N'oubliez pas les conventions d'écriture
- Soumettez votre projet selon les indications de l'instructeur

Marco Lavoie

14728 ORD - Langage C++

Pour solutionner cet exercice vous devez implanter dans la classe `Rationnel` des surcharges multiples des opérateurs demandés. En effet, ces opérateurs doivent permettre de manipuler des objets de type `Rationnel` entre eux ainsi qu'avec des entiers (l'entier 4 est équivalent à la rationnelle 4/1).



## Opérateurs ++ et --

45

- Ces opérateurs sont disponibles en deux versions : préfixe et suffixe

```
Date d( 31, 3, 2010 );
cout << d++ << " " << d << endl;
cout << --d << " " << d << endl;
```

```
C:\>date.exe
31/03/2010 01/04/2010
31/03/2010 31/03/2010
C:\>
```

- Les opérateurs d'incrémentation et décrémentation peuvent être surchargés
  - Via des fonction membres, ou
  - Via des fonctions amies

Marco Lavoie

14728 ORD - Langage C++

On peut surcharger tous les opérateurs d'incrémentation et de décrémentation, que ce soit l'opérateur de pré-incrémentation, de post-incrémentation, de pré-décrémentation ou de post-décrémentation. Prenons l'exemple des opérateurs d'incrémentation (l'approche est identique pour ceux de décrémentation).

Pour surcharger un opérateur d'incrémentation de façon à permettre l'usage simultané de la pré-incrémentation et de la post-incrémentation, chaque fonction d'opérateur surchargé doit avoir une signature distincte, afin que le compilateur détermine la version d'opérateur `++` proposée. Les versions à préfixe sont surchargées exactement de la même façon que les autres opérateurs unaires à préfixe. Par exemple, l'instruction suivante incrémentant un objet `Date` d'une journée,

```
++d
```

est transformée par le compilateur en invocation

```
d.operator++()
```

## Opérateurs ++ et -- (suite)

46

- Opérateurs en version préfixe (ex: ++d, --d)

- Fonctions membres

```
Classe & Classe::operator++();
Classe & Classe::operator--();
```

- Fonctions amies

```
friend Classe & Classe::operator++( Classe & );
friend Classe & Classe::operator--( Classe & );
```

- La surcharge des opérateurs en version suffixe (ex: d++, d--) exploite une signature spéciale

- Le type `int` est spécifié en argument, sans nommé le paramètre

```
Classe Classe::operator++( int );
Classe Classe::operator--( int );

friend Classe Classe::operator++( Classe &, int );
friend Classe Classe::operator--( Classe &, int );
```

*Notez les distinctions dans les signatures*

Marco Lavoie

14728 ORD - Langage C++

Si la pré-incrémentation est implantée comme une fonction non membre amie, le compilateur génère alors l'invocation

```
operator++( d )
```

La surcharge de l'opérateur de post-incrémentation présente un certain défi, car le compilateur doit être en mesure de distinguer entre les signatures des fonctions de l'opérateur de pré-incrémentation surchargé et de l'opérateur de post-incrémentation surchargé. La convention adoptée en C++ stipule que, lorsque le compilateur lit une expression de post-incrémentation

```
d++
```

ce dernier génère une invocation de la fonction membre

```
d.operator++( 0 )
```

Le 0 ne constitue qu'une « valeur fictive », employée pour marquer la distinction entre la liste d'arguments d'`operator++` de post-incrémentation et la liste d'arguments d'`operator++` de pré-incrémentation.

## Exemple : classe Date

47

```
class Date {
    friend ostream & operator<< ( ostream &, const Date & );

public:
    Date( int j = 1, int m = 1, int a = 1900 ); // constructeur
    void ajusterDate( int, int, int ); // ajuste la date
    Date & operator++(); // opérateur pré-incrémentation
    Date operator++( int ); // opérateur post-incrémentation
    const Date & operator+=( int ); // additionne les jours, modifie l'objet
    bool anneeBissextile( int ); // est-ce une année bissextile?
    bool finDeMois( int ); // est-ce la fin de mois?

private:
    int jour;
    int mois;
    int annee;

    static const int jours[]; // tableau des jours par mois
    void aideIncrementation(); // fonction utilitaire
};
```

Marco Lavoie

14728 ORD - Langage C++

Les prochaines pages illustrent la classe `Date`, utilisant des opérateurs surchargés de pré-incrémentation et post-incrémentation afin d'additionner 1 au jour dans un objet `Date` et d'effectuer, au besoin, des incréments au mois et à l'année.

L'interface `public` de `Date` inclut un opérateur d'insertion de flux surchargé, un constructeur par défaut, une fonction `ajusterDate`, un opérateur surchargé de pré-incrémentation, un opérateur surchargé de post-incrémentation, un opérateur surchargé d'affectation d'addition (`+=`), une fonction de test des années bissextiles ainsi qu'une fonction déterminant s'il s'agit du dernier jour du mois.

Les membres `private` de `Date` incluent les trois attributs permettant de stocker le jour, le mois et l'année, un tableau constant `static` stockant le nombre de jours dans chaque mois (sans égard aux années bissextiles) et une fonction `aideIncrementation` que gère le passage au prochain mois ou à la prochaine année lors de l'incrément du jour.

## Exemple : classe Date (suite)

48

```
// Initialise un membre static à portée de fichier,
// une copie à portée de classe
const int Date::jours[] = { 0, 31, 28, 31, 30, 31, 30,
                           31, 31, 30, 31, 30, 31 };

// Constructeur de Date
Date::Date( int j, int m, int a ) { ajusterDate( j, m, a ); }

// Ajuste la date
void Date::ajusterDate( int jj, int mm, int aa ) {
    mois = ( mm >= 1 && mm <= 12 ) ? mm : 1;
    annee = ( aa >= 1900 && aa <= 2100 ) ? aa : 1900;

    // Test pour une année bissextile
    if ( mois == 2 && anneeBissextile( annee ) )
        jour = ( jj >= 1 && jj <= 29 ) ? jj : 1;
    else
        jour = ( jj >= 1 && jj <= jours[ mois ] ) ? jj : 1;
}
```

Marco Lavoie

14728 ORD - Langage C++

Le constructeur par défaut de la classe `Date` invoque la fonction `ajusterDate` afin d'assurer la conformité des valeurs fournies pour les attributs de l'objet. Cette fonction négocie avec les « renouements » ou « reports » qui se produisent lors de l'incrément du dernier jour du mois, puisque ceux-ci nécessitent l'incrément du mois. Si le mois était le dernier de l'année, l'année doit également être incrémentée. La fonction `aideIncrementation` utilise les fonctions `anneeBissextile` et `finDeMois` afin d'incrémenter le jour correctement.

## Exemple : classe Date (suite)

```

// Opérateur de pré-incrémentation surchargé comme fonction membre
Date & Date::operator++() {
    aideIncrementation();
    return *this; // renvoi d'une référence pour créer une valeur gauche
}

// Opérateur de post-incrémentation surchargé comme fonction membre
// Notez que le paramètre d'entier fictif ne possède pas de nom de paramètre
Date Date::operator++( int ) {
    Date temp = *this;
    aideIncrementation();

    // Renvoie un objet non incrémenté, remis et temporaire
    return temp; // renvoi de valeur et non de référence
}

// Additionne un nombre de jours spécifique à une date
const Date & Date::operator+=( int joursAdditionnels ) {
    for ( int i = 0; i < joursAdditionnels; i++ ) aideIncrementation();

    return *this; // permet la mise en cascade
}

```

Marco Lavoie

14728 ORD - Langage C++

L'opérateur surchargé de pré-incrémentation retourne une référence vers l'objet **Date** courant, c'est-à-dire celui venant tout juste d'être incrémenté, puisque l'objet courant **\*this** est retourné comme **Date &**.

La surcharge de l'opérateur de post-incrémentation est un peu plus délicate. Afin d'émuler l'effet de la post-incrémentation, nous devons retourner une copie de l'objet **Date**. Nous sauvons l'objet courant (**\*this**) dans **temp** à l'entrée de **operator++**, pour invoquer ensuite **aideIncrementation** afin d'incrémenter l'objet **\*this**. Par la suite, nous retournons la copie non incrémentée de l'objet, stockée précédemment dans **temp**. Notez que cette fonction ne peut retourner une référence vers l'objet local **temp** puisque les variables locales sont détruites lors de la sortie de la fonction où elles sont déclarées. Par conséquent, une déclaration de retour du type **Date &** retournerait pour cette fonction une référence à un objet qui n'existe plus. Le renvoi d'une référence vers une variable locale constitue une erreur courante pour laquelle certains compilateurs émettront un message d'avertissement.

## Exemple : classe Date (suite)

```

// Si l'année est bissextile, renvoie true, sinon false
bool Date::anneeBissextile( int a ) {
    if ( a % 400 == 0 || ( a % 100 != 0 && a % 4 == 0 ) )
        return true; // année bissextile
    else
        return false; // année non bissextile
}

// Fonction d'aide pour incrémenter la date
void Date::aideIncrementation() {
    if ( finDeMois( jour ) && mois == 12 ) { // fin d'année
        jour = 1;
        mois = 1;
        ++annee;
    }
    else if ( finDeMois( jour ) ) { // fin de mois
        jour = 1;
        ++mois;
    }
    else // pas une fin de mois ni d'année; incrémente le jour
        ++jour;
}

```

Marco Lavoie

14728 ORD - Langage C++

La fonction **anneeBissextile** détermine si l'année courante est bissextile ou non. Une année est bissextile si elle est divisible par 400, ou si elle est divisible par 4 mais non par 100.

Finalement, la fonction **aideIncrementation** incrémente l'objet **Date** d'une journée tout en gérant les reports potentiels.

## Exemple : classe Date (suite)

```

// Détermine si le jour représente la fin du mois
bool Date::finDeMois( int j ) {
    if ( mois == 2 && anneeBissextile( annee ) )
        return j == 29; // dernier jour de février pour une année bissextile
    else
        return j == jours[ mois ];
}

// Opérateur de sortie surchargé
ostream & operator<<( ostream &sortie, const Date &j ) {
    static char *nomMois[ 13 ] = { "", "janvier", "février", "mars", "avril", "mai", "juin", "juillet", "août", "septembre", "octobre", "novembre", "décembre" };

    sortie << nomMois[ j.mois ] << " "
        << j.jour << " " << j.annee;

    return sortie; // permet la mise en cascade
}

```

Marco Lavoie

14728 ORD - Langage C++

La fonction **finDeMois** retourne le dernier jour du mois courant, qu'elle obtient du tableau **static** **jours**. Si le mois courant est février et l'année courante est bissextile, elle corrige le dernier jour en conséquence.

Finalement, l'opérateur d'insertion de flux de sortie surchargé (**operator<<**) affiche l'objet **Date** fourni sous forme textuelle.



## Erreurs de programmation

52

- Tenter de surcharger un opérateur non surchargeable
- Oublier de spécifier le référence (&) dans la surcharge lorsque requise
  - Ou spécifier la référence lorsque non requise
- Ne pas surcharger l'opérateur d'affectation (=) lorsque la classe dispose d'attributs dynamiques

Marco Lavoie

14728 ORD - Langage C++

Voici d'autres observations pouvant mener à des erreurs de programmation :

- Un constructeur de copie doit utiliser l'appel par référence et non l'appel par valeur. S'il en était autrement, l'appel du constructeur de copie résulterait en une récursion infinie, c'est-à-dire une erreur de logique fatale, puisque l'appel par valeur requiert le passage d'une copie de l'objet vers le constructeur de copie, provoquant les appels récursifs vers ce dernier.
- Si le constructeur de copie se limite à copier le pointeur de l'objet source dans le pointeur de l'objet cible, les deux objets pointent alors vers le même espace mémoire alloué dynamiquement à l'objet source antérieurement. Le premier des deux destructeurs qui s'exécuterait supprimerait alors cet espace mémoire commun, laissant l'autre objet avec un pointeur non défini; cette situation, qualifiée de « pointeur mal placé », est susceptible de provoquer une erreur grave à l'exécution.



## Bonnes pratiques de programmation

53

- Surcharger les opérateurs les plus enclins à être exploités avec votre classe
- Favoriser les surcharges via fonctions membres à celles via fonctions amies
- Réutiliser le code afin de minimiser les risques d'erreurs :

```
bool Chaine::operator==( const Chaine &droite ) const {
    return strcmp( sPtr, droite.sPtr ) == 0;
}

bool Chaine::operator!=( const Chaine &droite ) const {
    // Invoker l'opérateur ==
    return !( *this == droite );
}
```

Marco Lavoie

14728 ORD - Langage C++

Voici d'autres conseils portant sur la surcharge des opérateurs en C++ :

- Évitez l'utilisation abusive ou contradictoire de la surcharge des opérateurs qui pourrait rendre la lecture d'un programme obscure et difficile.
- Surchargez les opérateurs pour effectuer une fonction identique ou similaire sur des objets de classes, comme celles effectués par les opérateurs sur des variables de types prédéfinis. Rendez intuitive l'utilisation des opérateurs pour vos classes.
- Pour assurer la cohérence entre les opérateurs apparentés, utilisez-en un pour implanter les autres (tel que dans l'exemple ci-contre). En d'autres termes, utilisez un opérateur + surchargé pour implanter un opérateur += surchargé.
- Lors de la surcharge d'opérateurs, privilégiez l'utilisation de fonctions membres à celles non membres amies de la classe, si possible.



## Devoir #7

54

- Rehaussez la solution de l'exercice 8.3 en surchargeant les opérateurs suivants dans la classe **Rationnel**
  - Les opérateurs arithmétiques (+, -, \* et /) pour fractions et entiers (ex: f3 = f1 + 6 + f2)
  - Les opérateurs relationnels (==, !=, <, <=, > et >=) pour fractions et entiers (ex: f2 < f1 && f3 >= 7 || 8 < f4)
  - Les opérateurs d'affectation (=, +=, -=, \*= et /=) pour fractions et entiers (ex: f2 /= f1)
  - Les opérateurs d'incrémentement (++) et décrémentation (--) en format préfixe (ex: --f1) et suffixe (ex: f2++)
- Respectez l'échéance imposée par l'instructeur
- Soumettez votre projet selon les indications de l'instructeur
  - **Attention** : respectez à la lettre les instructions de l'instructeur sur la façon de soumettre vos travaux, sinon la note EC sera attribuée à ceux-ci

Marco Lavoie

14728 ORD - Langage C++

Ce devoir est relativement facile à réaliser. Inspirez-vous des exemples présentés dans les notes de cours pour surcharger les opérateurs indiqués pour la classe **Rationnel**.

Ce que les programmeurs débutants trouvent généralement le plus difficile dans la surcharge des opérateurs en C++ est la lourdeur de la syntaxe des signatures de fonctions. Il n'est pas nécessaire de connaître par cœur cette syntaxe; même les programmeurs expérimentés effectuent généralement du copier-coller pour intégrer la surcharge d'opérateurs dans une classe C++ : ils s'inspirent donc de projets antérieurs où les opérateurs visés furent surchargés.





## Pour la semaine prochaine

55

- Vous devez relire le contenu de la présentation du chapitre 8
  - Il y aura un **quiz** sur ce contenu au prochain cours
    - À livres et ordinateurs fermés
  - Profitez-en pour réviser le contenu des chapitres précédents

Marco Lavoie

14728 ORD - Langage C++

En début de classe la semaine prochaine vous aurez à répondre à des questions sur le C++ sans consultation du matériel pédagogique. Vous êtes donc fortement encouragé à relire les notes de cours du chapitre 8.

Profitez-en pour réviser le contenu des chapitres précédents.