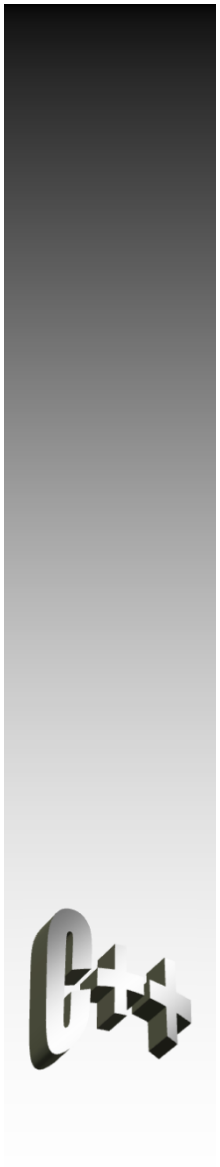


Chapitre 8
Surcharge des opérateurs

Auteur : Marco Lavoie

Adaptation : Sébastien Bois

Langage C++

14278 ORD

Objectifs

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

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`**

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

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;`

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

Restrictions de la surcharge

- 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

Fonctions de surcharge

- 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 surchargées 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

Fonctions de surcharge (suite)

- 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'invoquer **`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

Fonctions de surcharge (suite)

- 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

Surcharge d'entrées/sorties

- 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`

Surcharge d'entrées/sorties (suite)

- 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

Surcharge d'entrées/sorties (suite)

- 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 enchaînables est toujours une référence (**`&`**) afin de retourner la même instance que celle obtenu via paramètre

Surcharge d'entrées/sorties (suite)

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

Flux d'entrées à exploiter — *Instance de `Date` à lire*

```
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

Surcharge d'entrées/sorties (suite)

- 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

Surcharge d'entrées/sorties (suite)

- 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*

Exercice 8.1

- 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

Opérateurs arithmétiques

- 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`

Opérateurs arithmétiques (suite)

- 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.minute % 60;      // secondes restantes

  resultat.seconde = secRes;

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

Opérateurs arithmétiques (suite)

- 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

Opérateurs arithmétiques (suite)

- 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
}
```

Opérateurs arithmétiques (suite)

- 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;
  ...
};
```

Opérateurs arithmétiques (suite)

- 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
}
```

Opérateurs arithmétiques (suite)

- 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'autre fonctions de surcharge de cette même classe
    - Ça évite de dupliquer du code source, facilitant ainsi la maintenance

Exercice 8.2

- 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

Opérateurs d'affectation

- 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

Opérateurs d'affectation (suite)

- 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 )
    return *this;

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

  return *this;
}
```

On s'assure de ne pas affecter une instance à elle-même

- Toujours éviter l'auto-affectation lors de la surchagre de l'opérateur =
  - Essentiel pour les attributs dynamiques

Opérateurs d'affectation (suite)

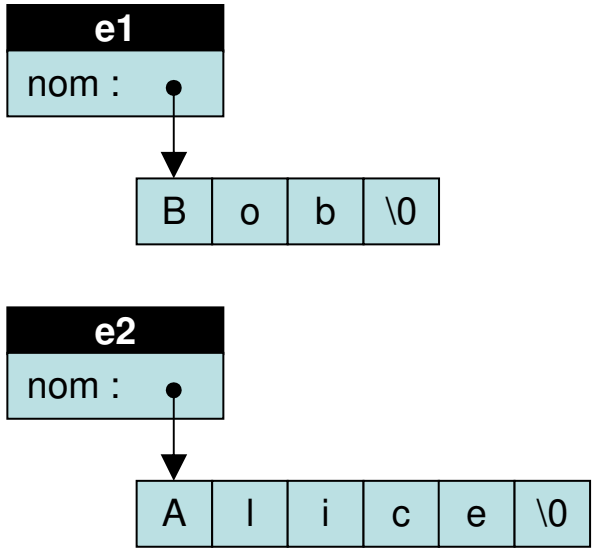
- Importance d'éviter l'auto-affectation

```
class Exemple {
public:
  Exemple( const char * = 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;
}
```



Opérateurs d'affectation (suite)

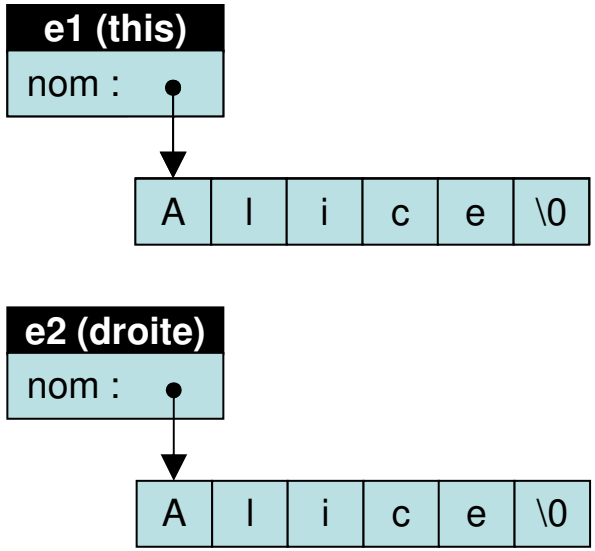
- Importance d'éviter l'auto-affectation (suite)

```
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;
}

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



Opérateurs d'affectation (suite)

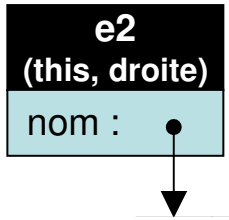
- Importance d'éviter l'auto-affectation (suite)

```
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;
}

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



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

Opérateurs d'affectation (suite)

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

```
const Exemple & Exemple::operator+=( const Exemple & droite) {
  // Calculer la longeur 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;
}
```

Opérateurs d'affectation (suite)

- 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 une 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 [] );
```

Opérateurs relationnels

- 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 & );
```

Opérateurs logiques

- 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 fonctionmembre 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 & );
```

Opérateur d'indiçage `[]`

- L'opérateur d'indiçage (`[]`) 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("Gustove");
cout << e[ 4 ];  // Affiche o
e[ 4 ] = 'a';    // Gustave
```

Exemple : classe **Chaine**

- Classe de manipulation de chaînes de caractères

```
class Chaine {
  friend ostream &operator<< ( ostream &, const Chaine & );
  friend istream &operator>> ( istream &, Chaine & );

public:
  Chaine( const char * = "" ); // constr. de conversion et par défaut
  Chaine( const Chaine & );    // constructeur de copie
  ~Chaine();                   // destructeur

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

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

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
};
```

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;
}
```

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;                // empêche 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;
}
```

Exemple : classe **`Chaine`** (suite)

```
// 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
}
```

Exemple : classe **Chaine** (suite)

```
// 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;
```

Exemple : classe **Chaine** (suite)

```
  // Allouer un tableau temporaire pour la sous-chaine 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
  strncpy( 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; }
```

Exemple : classe **Chaine** (suite)

```
// 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 entree

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

Exercice 8.3

- 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

Opérateurs ++ et --

- 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

Opérateurs ++ et -- (suite)

- Opérateurs en version préfixe (ex: `++d`, `--d`)
  - Fonctions membres
  - Fonctions amies

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

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 signature

Exemple : classe **Date**

```
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
};
```

Exemple : classe **`Date`** (suite)

```
// 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;
}
```

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, remisé 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
}
```

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;
}
```

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
}
```

Erreurs de programmation

- 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

Bonnes pratiques de programmation

- 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 {
  // Invoquer l'opérateur ==
  return !( *this == droite );
}
```

Devoir #7

- 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'incrémentation (**++**) 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, s*inon la note EC sera attribuée à ceux-ci*

Pour la semaine prochaine

- 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