Opérateurs

1. Principe
2. Opérateurs amis
3. Syntaxe
4. Opérateur d'indexation
5. Opérateurs logiques
6. Opérateur d'insertion et d'extraction
7. Opérateurs ++ et -

Principe

Les opérateurs forment une famille particulière de fonctions, spécifique à C++.

L'utilisation des opérateurs par surcharge illustre un aspect intéressant de la programmation évoluée, indépendant de la programmation objet.

Rappel de quelques principes:

* Chaque fonction a une signature, qui est la liste des types des paramètres formels (et le type du résultat);
* Un même nom de fonction peut désigner plusieurs fonctions de signatures différentes, sans compter le type du résultat;
* Il y a surcharge (ou polymorphisme dynamique) si le choix de la fonction est déterminé par le type des paramètres formels;
* Il y a polymorphisme (dynamique) si le choix de la fonction est déterminé par le type de l'objet appelant.

Les opérateurs

Les opérateurs sont au nombre de 40:

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

Les cinq derniers sont: l'indirection, l'indexation, l'appel de fonction, l'allocation et la désallocation.

Il n'est pas possible de changer la précédence des opérateurs,
de changer la syntaxe des expressions,
ni de définir de nouveaux opérateurs. Pourquoi ?

D'autres opérateurs existent:

. :: ?: sizeof .* &

qui ne peuvent être surchargés. Le & de prise de référence n'est pas surchargeable, le "et" bit à bit l'est.
Les opérateur ++ et – sont surchargeables des deux cotés.

Opérateurs amis

Une fonction amie d'une classe a les mêmes droits qu'une fonction membre, et en particulier peut accéder aux membres privés de la classe.

```
class Rat {
 int num, den; // prives
public:
 int getNum() {return num;}
 int getDen() {return den;}
 Rat(int n = 0, int d = 1) : num(n),  den(d) {}
};
```

L'addition

```
Rat operator+(const Rat& a, const Rat& b) {
  int n = a.getNum()*b.getDen() + a.getDen()*b.getNum();
  return Rat(n, a.getDen()*b.getDen());
}
```

s'écrit plus efficacement et plus lisiblement:

```
Rat operator+(const Rat& a, const Rat& b) {
  return Rat(a.num*b.den + a.den* b.num, a.den*b.den);
}
```

Or `num` et `den` sont privés. Pour les rendre accessibles, l'opérateur est déclaré ami, dans la déclaration de la classe Rat, par

```
friend Rat operator+(Rat, Rat);
```

Une fonction amie a les mêmes droits qu'une fonction membre.
Bien noter la différence entre fonction membre et fonction amie (le zéroième argument des fonctions membre).

Syntaxe

Un opérateur est unaire, binaire.
Un opérateur peut être défini
* soit comme fonction globale à un ou deux arguments
* soit comme fonction membre avec un argument de moins.

La syntaxe est
* opérateur binaire au niveau global:

```
type operatorop(type,type);
Rat operator+(Rat,Rat);
```

L'expression `u+v` équivaut à `operator+(u,v)`.
* opérateur binaire membre de classe:

```
type operatorop(type);
Rat& operator+=(Rat);
```

L'expression `u += v` équivaut à `u.operator+=(v)`.
* opérateur unaire au niveau global:

```
type operatorop(type);
Rat operator-(Rat);
```

L'expression `-u` équivaut à `operator-(u)`.
* opérateur unaire membre de classe:

```
type operatorop();
Rat operator-();
```

L'expression `-u` équivaut à `u.operator-()`.

Un Conseil

* Choisir un opérateur global et ami lorsque l'opération est symétrique: + - * / ==
* Choisir un opérateur membre lorsque l'opération est asymétrique ou nécessite une référence : +=

```
class Rat {
  int num, den;
public:
  ...
  Rat operator-(Rat);                 //membre : mauvais choix
  Rat& operator+=(const Rat&);        //membre :bon choix
  friend Rat operator/(Rat , Rat ); //amie : bon choix
};
```

Avec

```
Rat Rat::operator-(Rat b) {return Rat(num*b.den - den* b.num, den*b.den);}
Rat& Rat::operator+=(const Rat& d) {
  num = num*d.den + d.num*den;
  den = den*d.den;
  return *this;
}
Rat operator/(Rat a, Rat b) {
  return Rat(a.num*b.den, a.den*b.num);
}
```

Exemple

```
cout << a << b << a-b << a.operator-(b)
     << a-1 << a+=2 << a;
```

Résultat

```
1/2  2/3  -1/6  -1/6  -1/2 3/2 3/2
```

Vérifier la complétude de votre syntaxe

Attention

```
cout << 1-a ; // erreur syntaxique
```

L'expression

```
1.operator-(a)
```

et ceci n'est pas défini : il manque la fonction

```
friend Rat operator-(int, Rat);
```

Opérateur d'indexation

L'opérateur d'indexation [] est surchargé dans les vecteurs et matrices.

```
class Vecteur {
protected:
  int n;
  int *a;
public:
  int taille() const { return n;}
  Vecteur(int t = 10) : n(t), a(new int[n]) {}
  Vecteur(int t, int val) : n(t), a(new int[n]) {
    for (int i=0; i<n;i++) a[i] = val;
  }
  int& operator[](int i) { return a[i];} // surcharge
};
```

Il manque le constructeur de copie et l'opérateur d'affectation. Ainsi

```
Vecteur x;
```

crée un vecteur de taille 10.

* *Sans surcharge*, on accède aux éléments par `x.a[i]`.
* *Avec surcharge*, l'expression `x[i]` est équivalente à `x.operator[](i)`, donc à `x.a[i]`.
* La valeur de retour est une référence pour pouvoir servir en **lvalue**. On peut alors écrire:

```
main() {
  const int N = 4;
  Vecteur u(N, 7);
  Vecteur v(N, 5);
  for (int i = 0; i < N; i++)
    v[i] += u[i];
}
```

Vecteur suite

On en profite pour surcharger les entrées-sorties:

```
ostream& operator<<(ostream& out , Vecteur& u) {
  out << "(";
  for (int i=0; i<u.taille();i++) out << u[i] << " ";
  return out << ")" << endl;
};
istream& operator>>(istream& in , Vecteur& u) {
  for (int i=0; i<u.taille();i++) in >> u[i];
  return in;
};
```

Remarquer qu'ici, les opérateurs d'insertion et d'extraction n'ont pas besoin d'être déclarées amis, grace à cet opérateur d'accès.

Pour

```
main() {
  const int N = 4;
  Vecteur u(N, 7); cout << u;
  Vecteur v(N, 5); cout << v;
  for (int i = 0; i < N; i++)
    v[i] += u[i];
cout << v ;
}
```

On obtient alors les sorties

(7 7 7 7)
(5 5 5 5)
(12 12 12 12)

L'opérateur d'indexation [] est surchargé dans les vecteurs et matrices.

Pour l'indexation des matrices, voici deux méthodes. Elles reposent sur le fait qu'une matrice est un vecteur de lignes.

Pour des matrices de taille fixe, on définit

```
class Mat {
  float a[4][4];
 public:
  ...
  float* operator[](int i) { return a[i]; }
};
```

Ainsi `x[i]` est l'adresse de la i-ième ligne de a. Le sens de `x[i][j]` résulte de l'interprétation usuelle de l'indexation.

Le résultat `x[i]` est une r-valeur, et ne peut être une référence car c'est un tableau au sens de C.
En revanche, `x[i][j]` est une l-valeur comme en C.

Matrices

Les matrices dynamiques se définissent comme suit:

```
class Mat {
  float **a;
 public:
  int nlignes, ncol;
  Mat(int l, int c) {
    nlignes = l;
    ncol = c;
    a = new float*[nlignes];
    for (int i=0; i< nlignes; i++)
      a[i] = new float[ncol];
  }
  float*& operator[](int i) { return a[i]; }
};
```

Ici aussi, x[i] est l'adresse de la i-ième ligne de a.
Elle peut cette fois être une référence.
On peut donc écrire:

```
Mat  x(4,4);
float b[4] = {55,56,57,58};
delete x[3];
x[3] = b; // car [] retourne une reference
```

(on peut l'écrire mais moi non)

Opérateurs logiques

Tester l'égalité de deux objets par surcharge de l'opérateur ==.

```
int operator==(const Rat& a, const Rat& b) {
  return (a.num == b.num) && (a.den == b.den) ;
}
```

Cet opérateur s'écrit bien entendu aussi pour les chaînes de caractères, vecteurs, etc.

On peut nier par

```
int operator!=(const Rat& a, const Rat& b) {
  return !(a==b) ;
}
```

mais le mieux est de nier par

```
template <class T>
int operator!=(const T&a, const T&b)
{
return  !(a==b);
}
```

ce qui est plus global :).
de même

```
template <class T>
int operator>(const T&a, const T&b) { return  (b<a);}
```

et Hop.

Opérateurs d'insertion et d'extraction

Les opérateurs « et » sont souvent surchargés pour les entrées et sorties.

On va privilégier ce sens la par rapport a un décalage de bit dans la plus part des cas :).

Le problème des entrée sortie vient du fait qu'il existe souvent deux formats, un format « lisible » pour les humains et un format « machine/fichier » binaire , une des techniques pour ne pas se mélanger entre ces deux format et de créer un type de iostream spécifique pour les entrées/sortie binaire. Ainsi il n'y pas de confusion entre les deux types d'entrées sortie; En effet les entrées sorties pour humain sont loin d'être facile a manipuler en C/C++ et on laissera ce type de travaille a des couches logiciel distinctes (Interface Graphique, parseur, analyseur syntaxiques etc.).

Privilégier des opérateurs non amis, sauf si l'encapsulation ne le permet pas.

Une autre surcharge courante est pour l'insertion ou la suppression dans un conteneur.

Mais est-ce bien raisonnable ? La surcharge doit conserver le sens original des opérateurs.

[ZZZ: grand classique de Debutant StartUp incorp. Les opérateurs qui tuent :
String s; s++; // passe en majucule
++s; // consome le premier caractère comme une pile
+s; // test si c'est un palindrome ...

Opérateurs ++ et --

Il y a deux formes, préfixe et suffixe. Pour l'opérateur préfixe, le prototype est au niveau global:
```
type operator++ (type);
```
et comme membre de classe:
```
type operator++ ();
```

Pour l'opérateur suffixe, le prototype est au niveau global:
```
type operator++ (type,int);
```
et comme membre de classe:
```
type operator++ (int);
```
Il n'y a pas d'ambiguïté car `++` ne peut être redéfini sur `int`.

La valeur de l'argument passée est 0; ainsi, l'expression `x++` équivaut à `operator++(x,0)` respectivement à `x.operator(0)`.
Par exemple:
```
class Rat {
  int num, den;
public:
  ...
  Rat operator-(Rat);
  Rat operator++() { return Rat(den+num,den);}
  Rat operator++(int z) { Rat r = *this; num -= den; return r;}
};
```
permet d'écrire:
```
  cout <<"a et ++a"<< a << ++a << endl;
  cout <<"a et a++"<< a << a++ << endl;
```