Centre Informatique pour les Lettres et les Sciences Humaines

C++ : Leçon 15
Fonctions opérateur

La syntaxe normale pour invoquer une fonction est de mentionner son nom et de faire suivre celui-ci d'un couple de parenthèses entourant les variables ou valeurs que la fonction doit utiliser pour initialiser ses paramètres. Cette syntaxe n'est pas toujours la plus agréable que l'on puisse imaginer, notamment lorsque la fonction réalise une opération qui est habituellement désignée par un symbole bien connu. Imaginons par exemple que nous avons défini une classe `CTruc` pour laquelle la notion d'addition a un sens[1]. Il est, bien entendu, possible de définir une fonction membre nommée `ajoute()`, qui permettra d'écrire

```
//premierTruc, secondTruc et trucSomme sont des instances de CTruc
trucSomme = premierTruc.ajoute(secondTruc);
```

Toutefois, notre code serait sans doute plus lisible si nous pouvions écrire

```
trucSomme = premierTruc + secondTruc;
```

C'est justement cette possibilité que nous offre le langage C++ en nous autorisant à définir des fonctions qui peuvent être appelées à l'aide de certains opérateurs.

1 - Quelles fonctions opérateur pouvons-nous définir ?

En dépit de sa grande générosité, C++ ne permet pas de créer de *nouveaux opérateurs*, mais seulement de définir la façon dont certains de ses opérateurs peuvent s'appliquer à de nouveaux types d'opérandes[2]. Nous ne pourrons donc jamais écrire des choses comme :

```
1  if (a <> b)      //il n'existe pas d'opérateur <> en C++
2    c = a # b;     //il n'existe pas d'opérateur # en C++
3  else
4    c = a : b;     //il n'existe pas d'opérateur : en C++
```

Qui plus est, parmi les opérateurs C++, cinq ne peuvent pas faire l'objet de fonctions opérateur :

Symbole	Nature	Commentaire
`sizeof`	Calcul de taille	Cet opérateur donne déjà le seul résultat raisonnable (la taille en octet de son opérande), même lorsqu'il est appliqué à une classe ou à une instance. Le redéfinir ne donnerait certainement rien de bon.
`?:`	if arithmétique	C'est le seul opérateur ternaire du langage. Il aurait donc fallu introduire une syntaxe spéciale pour qu'une fonction le définissant puisse accéder à ses trois paramètres, et l'on a sans doute estimé que cela ne se justifiait pas.
`::`	Résolution de portée	Ces opérateurs touchent à des mécanismes fondamentaux. Il est difficile d'envisager de les redéfinir sans compromettre gravement la cohérence du langage.
`.`	Sélection d'un membre	
`.*`	Déréférencement d'un pointeur sur membre[3]	

Bien entendu, puisqu'il s'agit de conférer de nouvelles significations à des symboles que le langage utilise déjà, il faut que le contexte permette de lever l'ambiguïté. Deux cas peuvent se présenter :
- Si la fonction opérateur est membre d'une classe, le type de l'objet au titre duquel elle est invoquée la désigne de façon parfaitement univoque.
- Si la fonction opérateur est globale (c'est-à-dire n'est pas membre d'une classe), il s'agit d'un cas de surcharge[4], et le type de ses arguments doit permettre de la distinguer de ses homonymes.

---

[1] Peut-être la classe `CTruc` comporte-t-elle quelques variables membre numériques, dont l'addition des valeurs qu'elles présentent dans deux instances donnerait la représentation d'un `CTruc` qui pourrait légitimement être considéré comme la somme des deux premiers ?

[2] Il n'est, en général, pas possible de redéfinir la façon dont les opérateurs s'appliquent aux types prédéfinis. La surcharge globale des opérateurs `new`, `new[]`, `delete` et `delete[]` constitue à cet égard une exception.

[3] Les pointeurs sur membre et leur usage sont présentés dans l'Annexe 9.

[4] En effet, la définition du comportement des opérateurs lorsqu'ils sont appliqués à des objets relevant des types prédéfinis peut être considérée comme donnant (implicitement) naissance à des fonctions opérateurs globales. Celles-ci se trouvent donc surchargées lorsque de nouvelles fonctions globales entreprennent de définir l'application des opérateurs à de nouveaux types.

Nous pouvons donc conclure qu'une fonction opérateur est toujours liée à un type défini par le programmeur : soit parce qu'elle est membre d'une classe, soit parce qu'au moins un de ses paramètres relève d'un type énuméré ou d'une classe.

Si la définition d'une fonction opérateur permet de spécifier comment cet opérateur s'applique à de nouveaux types, il reste des caractéristiques de l'opérateur que cette définition ne saurait remettre en cause. C'est en particulier le cas de la priorité et du nombre d'arguments. Ainsi, quel que soit le type des objets concernés,

```
x + y * z; //l'opérateur * est prioritaire par rapport à l'opérateur +
```

équivaut toujours à x + (y * z) et non à (x + y) * z, et il ne sera jamais possible d'écrire

```
j = k &&; //&& est un opérateur nécessitant 2 opérandes !
```

Par rapport à une fonction "ordinaire", une fonction opérateur ne présente guère que trois particularités :
- La première est assez anecdotique : le nom attribué à la fonction lors de sa déclaration est composé du mot `operator`, suivi du symbole qui sera utilisé pour appeler la fonction.
- La deuxième particularité des fonctions membre découle de la syntaxe particulière utilisée pour les appeler : leurs paramètres ne peuvent pas être dotés de valeurs par défaut.
- La dernière particularité mérite plus d'attention, car il s'agit de la façon dont la fonction accède aux opérandes soumis à l'opérateur, et cette façon varie selon le nombre d'objets impliqués et selon si la fonction opérateur est globale ou membre d'une classe.

2 - Dans quels cas est-il judicieux de surcharger un opérateur ?

Dans la plupart des cas, le recours à une fonction opérateur ne relève que du souci d'augmenter le confort des programmeurs et d'assurer une lisibilité maximale au texte source. Ces objectifs ne seront atteints que dans la mesure où le symbole de l'opérateur surchargé révèle sans ambiguïté la nature de l'opération qu'il effectue.

S'il peut y avoir le moindre doute sur le sens qu'aurait l'application d'un opérateur à l'un de vos types, n'utilisez pas une fonction opérateur mais une fonction ordinaire portant un nom bien choisi.

Il existe toutefois des cas particuliers : l'opérateur d'affectation doit parfois impérativement être surchargé[5]. En effet, en l'absence d'une fonction opérateur le prenant explicitement en charge, cet opérateur est défini par défaut comme effectuant une copie membre à membre de l'opérande de droite vers l'opérande de gauche. Il s'agit là d'un traitement tout à fait semblable à celui opéré par le constructeur par copie fourni automatiquement par le langage (cf. Leçon 13). Si ce traitement convient parfaitement dans un très grand nombre de cas, il s'avère parfois tout à fait inacceptable, ce qui entraîne à la fois la nécessité de définir explicitement un constructeur par copie et celle de surcharger l'opérateur d'affectation.

3 - Surcharger les opérateurs monadiques

Le langage C++ comporte huit opérateurs qui s'appliquent à un opérande unique. Dans la plupart des cas, la syntaxe exige que le symbole de l'opérateur précède la désignation de son opérande :

Opérateur	Exemple
!	`bool n = ! true;`
-	`int a = - 2;`
+	`a = + 2;`
&	`int * ptr = & a;`
*	`* ptr = 4;`
~	`char c = ~ 'x';`
++	`++ a;`
--	`-- a;`

[5] Si cette obligation ne se présentait jamais, cette Leçon n'existerait sans doute pas (ou porterait le numéro 43...).

Les opérateurs ++ et -- se singularisent toutefois par le fait qu'ils admettent également une syntaxe suffixée[6], qui doit être prise en charge par une fonction spécifique.

Les opérateurs préfixés

La surcharge d'un opérateur ayant un seul argument peut être réalisée de deux façons : soit par une fonction membre de la classe visée, soit par une fonction globale.

S'il s'agit d'une fonction membre, elle doit être dépourvue d'argument, car elle sera invoquée au titre de l'instance sur laquelle l'opérateur est appliqué. En d'autres termes, si une classe `CTruc` possède, par exemple, une fonction membre surchargeant l'opérateur `!`, alors l'instruction

```
! monTruc; //monTruc est une instance de la classe CTruc
```

est équivalente à

```
monTruc.operator !(); //appel explicite de la fonction opérateur !
```

La définition d'une fonction membre surchargeant un opérateur ne présente qu'une seule particularité : son nom complet comporte non seulement celui de la classe, mais aussi le mot `operator` précédant le symbole de l'opérateur concerné. Le type de la valeur éventuellement renvoyée par la fonction, tout comme la logique des opérations qu'elle effectue, dépend bien entendu totalement de la sémantique associée à la classe en général, et à l'opérateur concerné en particulier.

```
1  CTruc CTruc::operator !() const //cet opérateur est sans effet
2  {
3  CTruc resultat = *this;
   //modification de resultat pour calculer le résultat de l'application de !
   //…
   return resultat;
   }
```

> L'opérateur `!` étant habituellement dénué d'effet, il est vraisemblable que l'objet sur lequel il est appliqué ne doit pas être modifié (d'où une déclaration `const` de la fonction membre et l'usage d'une variable temporaire pour calculer le `resultat`). Cet opérateur doit, en revanche, produire comme `resultat` une valeur du même type que son opérande, ce qui explique que la fonction renvoie une valeur de type `CTruc`.

Si la fonction surchargeant l'opérateur est une fonction globale, elle doit être munie d'un paramètre unique, qui correspond à l'opérande sur lequel l'opérateur est appliqué. Si l'opérateur doit avoir un effet, ce paramètre sera de type "référence à une instance de la classe visée", ce qui permettra à la fonction de modifier l'opérande. Si l'opérateur ne fait que produire un résultat, la fonction peut se contenter de recevoir une valeur ayant pour type la classe concernée, et les manipulations qu'elle effectuera sur son paramètre resteront sans effet sur l'objet utilisé comme opérande.

Un exemple d'une telle fonction globale pourrait être :

```
1  CTruc operator !(CTruc parametre) //cet opérateur est sans effet
2  {
   //modifications de parametre pour calculer le résultat de l'application de !
   //…
   return parametre; //renvoie la valeur obtenue
   }
```

> Dans la plupart des cas, cette fonction globale va devoir accéder à des membres de l'objet concerné qui ne sont pas publics. Il faut alors faire de cette fonction une amie de la classe dont cet objet est une instance.

Les opérateurs ++ et –- suffixés

Les opérateurs d'incrémentation et de décrémentation doivent correspondre à des fonctions opérateurs différentes selon s'ils sont placés avant ou après leur opérande. L'effet qu'ont ces

[6] Lorsqu'un opérateur est suffixé (on dit aussi parfois *postfixé*), il apparaît après (i.e. à droite de) son opérande.

opérateurs sur leur opérande est a priori le même dans les deux cas, mais la valeur renvoyée doit être différente : il s'agit de la valeur de l'opérande avant que le traitement n'ait été effectué lorsque l'opérateur est suffixé, et de la valeur de l'opérande après application du traitement lorsque l'opérateur est préfixé.

> Il serait très fâcheux qu'une fonction surchargeant l'opérateur ++ ou l'opérateur -- n'adhère pas à ces conventions, qui sont profondément ancrées dans la culture de tous les programmeurs C++.

Si la création des fonctions opérateurs correspondant à l'usage préfixé ne diffère en rien du cas des autres opérateurs à un seul argument, un problème se pose pour définir les fonctions correspondant à l'usage suffixé. Le nom de la fonction (`operator ++` ou `operator --`) est en effet nécessairement le même que pour la fonction correspondant à l'usage préfixé. Pour que les fonctions prenant en charge les usages préfixé et suffixé d'un même opérateur puissent être toutes deux définies, il faut donc que leurs signatures diffèrent. La liste des arguments nécessaires (aucun argument dans le cas d'une fonction membre, un seul argument, du type concerné, dans le cas d'une fonction globale) étant la même dans les deux cas, et la constance de la fonction étant exclue par la nature de l'opérateur, les concepteurs du langage C++ ont décidé d'introduire un argument inutile (de type `int`) pour singulariser la signature de la fonction opérateur devant être invoquée lorsque l'opérateur est utilisé en position suffixée.

Si les fonctions opérateur sont membre de la classe, leur définition sera donc quelque chose comme :

```
1  CTruc & CTruc::operator ++()//EXECUTEE LORSQUE ++ EST PREFIXE
2  {
   //modification des variables membre pour réaliser l'effet associé à ++
   //…
   return *this;              //renvoie l'opérande modifié
   }
```

```
1  CTruc CTruc::operator ++(int) //EXECUTEE LORSQUE ++ EST POSTFIXE
2  {
3  CTruc resultat(*this);     //met la valeur initiale de côté
4  ++ (*this) ;               //modifie la valeur à l'aide de l'opérateur préfixé
5  return resultat;           //renvoie la valeur initiale de l'opérande
6  }
```

Si les fonctions opérateur sont globales[7], elles ressembleront à :

```
1  CTruc & operator ++(CTruc &operande)//EXECUTEE LORSQUE ++ EST PREFIXE
2  {
   //modification de operande pour réaliser l'effet associé à ++
   //…
   return operande;           //renvoie l'opérande modifié
   }
```

```
1  CTruc operator ++(CTruc &operande, int) //EXECUTEE LORSQUE ++ EST POSTFIXE
2  {
3  CTruc resultat(operande); //met la valeur initiale de côté
4  ++ operande ;              //modifie la valeur à l'aide de l'opérateur préfixé
5  return resultat;           //renvoie la valeur initiale de l'opérande
6  }
```

> Il est préférable que les fonctions opérateurs prenant en charge l'usage préfixé d'un opérateur ++ ou -- ne se contentent pas de renvoyer la nouvelle valeur de l'opérande, mais renvoient la référence à cet opérande qui leur a été transmise comme paramètre. De cette façon, les expressions utilisant l'opérateur préfixé sont équivalentes à leur opérande, ce qui en fait des lvalue. Même si cette propriété ne présente pas un intérêt majeur, nous avons vu dans la Leçon 11 que les opérateurs ++ et -- la possèdent lorsqu'ils sont appliqués aux types prédéfinis. Il est donc souhaitable que l'extension de ces opérateurs à un type que nous définissons la possède aussi.

[7] D'un strict point de vue syntaxique, rien n'empêche de panacher (la fonction opérateur correspondant à l'usage préfixé peut être membre de la classe alors que celle correspondant à l'usage suffixé est globale, par exemple), mais je ne vois pas l'intérêt que cette hétérogénéité pourrait bien présenter...

Bien que les opérateurs d'incrémentation et de décrémentation n'admettent qu'un seul argument, les fonctions opérateur prenant en charge leur usage suffixé impliquent donc deux objets (soit l'instance au titre de laquelle la fonction est exécutée et un paramètre entier, soit deux paramètres). Cette implication de plusieurs objets va, bien entendu, se retrouver dans les fonctions prenant en charge les opérateurs qui admettent deux arguments.

4 - Surcharger les opérateurs dyadiques infixés

Le langage C++ ne comporte pas moins de trente et un opérateurs qui produisent un résultat à partir de deux opérandes placés de part et d'autre du symbole de l'opérateur et qui peuvent être surchargés. Trente d'entre eux peuvent faire l'objet soit d'une fonction opérateur globale, soit d'une fonction opérateur membre d'une classe, alors que l'opérateur d'affectation ne peut être pris en charge que par une fonction membre.

> Lorsqu'un opérateur admet deux opérandes de types différents mais jouant des rôles symétriques, il est généralement préférable de définir deux fonctions opérateur le surchargeant : l'une prendra en charge les expressions adoptant l'ordre `typeA # typeB`, alors que l'autre traitera les cas `typeB # typeA`. Si cela n'est pas fait, les utilisateurs devront se faire à l'idée qu'ils peuvent par exemple écrire
>
> ```
> monTruc + 4;
> ```
>
> mais pas
>
> ```
> 4 + monTruc;
> ```
>
> Ce genre de limitations est très désagréable, surtout dans le cas d'opérateurs qui sont, comme l'addition, perçus comme étant vraisemblablement commutatifs.

Fonction opérateur globale

Les opérateurs dyadiques susceptibles d'être surchargés par une fonction globale[8] sont :

+	-	/	*	%	==	>	<	&&	,	^	&	|	<<	>>
+=	-=	/=	*=	%=	!=	>=	<=	||	->*	^=	&=	|=	<<=	>>=

> Remarquez que les symboles +, -, * et & correspondent chacun à deux opérateurs : un opérateur monadique et un opérateur dyadique. C'est donc le contexte (c'est à dire, ici, la présence d'un ou de deux opérandes) qui détermine la signification de chacune des occurrences de l'un de ces symboles. Les caractères * et & sont par ailleurs utilisés pour déclarer respectivement des pointeurs et des références. Ils ne désignent aucun opérateur lorsqu'ils apparaissent dans ce contexte.

Tous ces opérateurs utilisent une notation infixée : l'un des opérandes est à gauche de l'opérateur, alors que l'autre est à droite. Cette régularité permet d'exprimer simplement la correspondance entre les opérandes et les arguments d'une fonction globale surchargeant un opérateur dyadique : si # représente l'un des 30 symboles mentionnés ci-dessus

```
operandeGauche # operandeDroit ; //notation infixée
```

est équivalent à l'appel explicite de la fonction réalisé par

```
operator #(operandeGauche, operandeDroit) ;
```

Comme nous l'avons signalé dans la Leçon 8, il est souvent intéressant de rendre les classes capables de se décrire elles-mêmes sous formes textuelles. Il devient alors possible d'envoyer dans un fichier le contenu de toutes les variables membres d'un objet, en une seule opération :

```
1 ofstream fichier("essai.txt");
2 fichier << unTruc;        //unTruc est une instance de la classe CTruc
```

Il faut pour cela définir une fonction prenant en charge l'opérateur d'insertion lorsque son opérande de droite est de type `CTruc` :

[8] D'après mes calculs, un seul de ces opérateurs devrait vous être encore totalement inconnu : `->*`. Son rôle est présenté dans l'Annexe 9.

```
ostream & operator << (ostream &out, CTruc leTruc)
{
//il suffit ici d'envoyer chacun des membres dans le flux out à l'aide
//d'instructions du type :
out << leTruc.leMembre << endl;
//…

return out;
}
```

Bien entendu, cette fonction globale ne sera autorisée à accéder aux membres de son second paramètre que si ces membres sont `public:` (ce qui n'est pas conseillé) ou si elle bénéficie d'une relation d'amitié avec la classe `CTruc` (ce qui est la bonne solution).

Remarquez que la fonction `operator <<` ne se contente pas d'insérer toutes les valeurs des variables membre de son second paramètre dans le flux qui lui est désigné par son premier paramètre, mais qu'elle prend aussi la peine de renvoyer la référence à ce flux[9] qui lui a été transmise. En d'autres termes, la fonction ne se contente pas de réaliser l'effet souhaité (les insertions de données dans le flux), elle confère de plus une valeur de type "référence à un flux" à l'expression qui l'a appelée.

Pourquoi est-il intéressant qu'une expression telle que

```
fichier << unTruc;
```

soit de type "référence à un flux" ? Parce que, de cette façon, elle désigne un flux, ce qui permet d'utiliser l'expression en question comme opérande de gauche d'un autre opérateur d'insertion :

```
fichier << unTruc << "Fin des valeurs contenues dans unTruc\n"
```

Fonction opérateur membre d'une classe

Lorsque la fonction opérateur est membre d'une classe, la notation infixée usuelle

```
operandeGauche # operandeDroit ; //notation infixée
```

est équivalente à l'appel explicite de la fonction réalisé par

```
operandeGauche.operator #(operandeDroit);
```

Cette équivalence syntaxique révèle immédiatement une contrainte importante : l'opérande gauche doit nécessairement être une instance de la classe à laquelle appartient la fonction opérateur.

Il n'est possible de surcharger un opérateur dyadique à l'aide d'une fonction membre d'une classe que lorsque l'opérande de gauche est une instance de cette classe.

En contrepartie de cette limitation, l'usage d'une fonction membre permet non seulement de surcharger l'un des trente opérateurs surchargeables par une fonction globale, mais aussi l'opérateur d'affectation.

Si la classe `CTruc` comporte deux variables membres nommées `m_a` et `m_b`, il peut être intéressant de définir un opérateur de test d'égalité entre instances qui renvoie `true` si et seulement si les deux membres ont des valeurs identiques dans ses deux opérandes :

```
bool CTruc::operator == (CTruc unAutre)
{
return (m_a == unAutre.m_a && m_b == unAutre.m_b);
}
```

En tant que fonction membre de `CTruc`, notre fonction opérateur dispose de privilèges lui autorisant l'accès aux membres de son paramètre (qui est également de classe `CTruc`), même si ceux-ci sont protégés (ce qui est souhaitable).

[9] Le type utilisé ici n'est pas ofstream (comme vous vous y attendiez peut-être) mais ostream, une classe qui comprend non seulement les ofstream, mais également d'autres flux. Dans un contexte "console", la fonction présentée ici permet ainsi d'afficher à l'écran le contenu d'un `CTruc` en écrivant simplement `cout << unTruc;`

Une fois cet opérateur défini, il devient possible de comparer deux instances de `CTruc` :

```
//premierTruc et secondTruc sont des instances de CTruc
if(premierTruc == secondTruc)
  //traiter le cas d'égalité
```

Rappel : aucune version par défaut de l'opérateur de comparaison n'est fournie par le langage. Si des comparaisons entre instances sont nécessaires, il faut impérativement écrire une fonction les prenant en charge. Il n'est toutefois pas indispensable de surcharger l'opérateur ==, puisqu'une fonction membre nommée `estEgalA()` et utilisant un argument du type concerné permettra d'obtenir rigoureusement le même effet, au prix, il est vrai, d'une notation un peu moins agréable à l'œil du programmeur aguerri :

```
if (premierTruc.estEgalA(secondTruc))
  //traiter le cas d'égalité
```

5 - Opérateurs sur `enum` : un exemple

La possibilité de surcharger les opérateurs n'est pas réservée aux cas où ceux-ci sont appliqués à des instances de classes. Il est tout à fait possible de créer des fonctions opérateur globales dont l'un des paramètres est d'un type énuméré, ce qui permet d"accroître considérablement l'intérêt de ce genre d'objets.

Pour bien comprendre l'exemple qui va suivre, il est nécessaire de savoir qu'il existe une correspondance entre les valeurs énumérées et les valeurs entières. Ainsi, si le type `EJour` est défini par

```
enum EJour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
```

la valeur `LUNDI` correspond à 0, `MARDI` à 1, et ainsi de suite jusqu'à `DIMANCHE`, qui correspond à 6[10]. Le langage opère automatiquement, en cas de besoin, la conversion du type énuméré vers le type `int`, comme l'illustre le fait que

```
cout << JEUDI;
```

se traduira par l'affichage[11] du nombre 3. La conversion inverse n'est en revanche pas effectuée automatiquement, et l'instruction

```
EJour unJour = 3;
```

ne provoquera pas l'initialisation de la variable `unJour` avec la valeur `JEUDI`, mais une pure et simple erreur de compilation.

Conversion d'une valeur entière en valeur de type `EJour`

La première chose à faire est donc de se doter d'une fonction permettant de convertir un entier en un `EJour` de valeur équivalente. Au passage, cette fonction peut en profiter pour ramener dans la plage de valeurs admissible les valeurs dépassant celle correspondant à `DIMANCHE` (ce qui est facilement obtenu au moyen de l'opérateur %=). Les éventuelles valeurs négatives peuvent aussi être ramenées dans la plage admissible, au moyen d'une simple addition. Un transtypage explicite permet alors de convertir la valeur entière (dont nous sommes maintenant certains qu'elle est positive et strictement inférieure à 7) en une valeur de type `EJour`.

```
1 EJour jour(int val) //fonction de transtypage d'int vers EJour
2 {
3 return static_cast <EJour> (((val%=7) < 0) ? val+7 : val);
4 }
```

[10] Il est possible de modifier cette équivalence si elle ne convient pas à l'usage envisagé. Il me semble, toutefois, que si les valeurs entières associées à chacune des valeurs possibles pour un objet d'un type énuméré ont une réelle importance, on s'écarte du principe fondamental des types énumérés.
[11] On suppose ici que l'exécution a lieu dans un environnement "console" et que le programme comporte une directive #include "iostream.h".

Représentation écrite de la valeur d'un EJour

Le transtypage automatique d'une valeur de type EJour en la valeur entière équivalente ne permet pas toujours d'obtenir l'effet souhaité. Un problème fréquemment rencontré est celui de l'insertion d'un EJour dans un flux : l'apparition de valeurs numériques dans celui-ci ne correspond sans doute pas à l'attente du programmeur qui utilise des objets de type EJour. Ce problème peut être résolu en surchargeant l'opérateur d'insertion dans les flux, de façon à ce que ceux-ci reçoivent une chaîne de caractères plus évocatrice de la signification de la valeur.

```
ostream & operator << (ostream &out, EJour unJour)
{
const char * N[7] = {"Lundi", "Mardi", "Mercredi", "Jeudi",
                     "Vendredi","Samedi","Dimanche"};
out << N[unJour];
return out;
}
```

Remarquez l'utilisation de unJour comme index permettant d'accéder à l'élément du tableau qui nous intéresse. Ceci n'est bien entendu possible que parce que C++ sait transformer automatiquement les valeurs de type CJour en valeurs de type int...

Fonctions operator ++()

Passer d'un jour à l'autre est une opération que l'on peut incontestablement qualifier de quotidienne. L'expression de ce phénomène dans un programme sera simplifiée si l'opérateur d'incrémentation est capable d'agir sur les EJour. La définition des fonctions opérateur rendant ceci possible est simplifiée par la disponibilité de la fonction jour().

```
EJour & operator ++ (EJour &unJour)      //++ préfixé
{
unJour = jour(unJour + 1);
return unJour;
}
```

```
EJour operator ++ (EJour &unJour, int) //postfixed ++
{
EJour avant = unJour;
++ unJour; //utilisation de l'opérateur préfixé défini ci-dessus !
return avant;
}
```

La conversion automatique est ici encore mise à contribution : pour évaluer le résultat de l'addition, la valeur de unJour doit être convertie en une valeur entière (puisque nous n'avons pas défini d'opérateur + prenant en charge les EJour). Ce résultat est ensuite reconverti en une valeur de type EJour, qui peut ensuite être affectée à l'objet cible. L'effet de l'opérateur étant obtenu, il reste à fournir son résultat, ce qui est simplement réalisé en renvoyant la référence reçue comme paramètre.

Utilisation du type EJour

Une fois ces quelques fonctions définies, il est possible d'écrire du code ressemblant à ceci :

```
EJour aujourdhui = MERCREDI;
int n;
for (n = 0 ; n < 10 ; n++)
   cout << aujourdhui ++ << endl;
```

L'exécution d'une telle boucle produirait l'affichage suivant :

```
Mercredi
Jeudi
Vendredi
Samedi
Dimanche
Lundi
```

```
Mardi
Mercredi
Jeudi
Vendredi
```

Selon les besoins, on peut bien entendu envisager de surcharger d'autres opérateurs (`--`, `+=`, `-=` et l'extraction depuis un flux sont les premiers qui viennent à l'esprit), de façon à augmenter encore les possibilités d'utilisation des variables de type `EJours`.

6 - Bon, c'est gentil tout ça, mais ça fait quand même 9 pages. Qu'est-ce que je dois vraiment en retenir ?

1) Les fonctions opérateur permettent de définir la façon dont certains opérateurs s'appliquent aux types définis par l'utilisateur, qu'il s'agisse de classes ou de types énumérés.
2) Une fonction surchargeant un opérateur est soit globale soit membre de la classe dont l'opérande de gauche est une instance.
3) Du point de vue de la mise au point du code qui la définit, une fonction opérateur n'est en rien différente d'une fonction "ordinaire" qui ferait le même travail.
4) L'opérateur d'affectation est le seul dont la surcharge est parfois indispensable.
5) La disponibilité d'opérateurs surchargés peut accroître considérablement l'agrément d'utilisation des types définis par l'utilisateur, à condition que l'usage fait des opérateurs reste suffisamment proche de la signification usuelle de ceux-ci.
6) L'utilisation d'opérateurs surchargés réalisant des traitements qui s'éloignent trop du sens habituel du symbole utilisé peut rendre les programmes parfaitement incompréhensibles.
7) La surcharge de certains opérateurs s'écarte un peu des règles générales et est décrite dans l'Annexe 7 : Les fonctions opérateur `->`, `()`, `[]`, `new` et `delete`