



Centre Informatique pour les **L**ettres
et les **S**ciences **H**umaines

Apprendre C++ avec Qt : Leçon 11

Fonctions opérateur

1 - Généralités.....	2
Quelles fonctions opérateur pouvons-nous définir ?	2
Immuabilité de la précédence des opérateurs et de leur arité	3
Particularités des fonctions opérateur	3
Dans quels cas est-il judicieux de surcharger un opérateur ?	3
2 - Surcharger les opérateurs monadiques	3
Les opérateurs préfixés.....	4
Les opérateurs ++ et -- suffixés	4
3 - Surcharger les opérateurs dyadiques infixés	5
Fonction opérateur globale	6
Fonction opérateur membre d'une classe.....	6
4 - Un cas particulier : l'opérateur "parenthèses"	7
Fonctionnoïdes	7
Opérateurs de transtypage	8
5 - Opérateurs sur enum : un exemple	9
Conversion d'une valeur entière en valeur de type EJour	9
Représentation écrite de la valeur d'un EJour	10
Fonctions operator ++()	10
Utilisation du type EJour	11
6 - Bon, c'est gentil tout ça, mais ça fait déjà 9 pages. Qu'est-ce que je dois vraiment en retenir ?	11

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. 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 les fonctions qui seront exécutées pour évaluer les expressions comportant certains opérateurs.

1 - Généralités

Le code définissant le traitement qui doit correspondre à un opérateur va prendre place dans ce que l'on appelle une "*fonction opérateur*". La définition d'une telle fonction est aussi souvent qualifiée de *surcharge de l'opérateur* en question. Cette surcharge est soumise à quelques restrictions, mais ne diffère finalement qu'assez peu de la définition d'une fonction "ordinaire".

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. Nous ne pourrions 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	

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, et le type de ses arguments doit permettre de la distinguer de ses homonymes.

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.

Immuabilité de la précedence des opérateurs et de leur arité

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 précedence (ensemble de règles qui détermine dans quel ordre sont effectuées les opérations lors de l'évaluation d'une expression impliquant plusieurs opérateurs) et de l'arité (nombre d'arguments nécessaires à l'application d'un opérateur). Ainsi, quel que soit le type des objets concernés,

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

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

```
j = k && i; //&& est un opérateur nécessitant deux opérands !
```

Particularités des fonctions opérateur

Par rapport à une fonction "ordinaire", une fonction opérateur ne se distingue guère que de trois façons.

- 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 opérateur 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 a **accès aux opérands** soumis à l'opérateur : cette façon varie selon le nombre d'objets impliqués et selon si la fonction opérateur est globale ou membre d'une classe.

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'utilisation de conteneurs (cf. Leçon 8) exige parfois que l'opérateur de comparaison `==` puisse être appliqué à deux instances de la classe concernée, et l'opérateur d'affectation doit parfois impérativement être surchargé.

En effet, en l'absence d'une fonction opérateur le prenant explicitement en charge, l'opérateur d'affectation est automatiquement défini comme une copie membre à membre de l'opérande de droite vers l'opérande de gauche. Il s'agit là d'un traitement semblable à celui opéré par le constructeur par copie fourni automatiquement par le langage (cf. Leçon 10). 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 plus judicieux et celle de surcharger l'opérateur d'affectation.

2 - 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** (cf. tableau ci-contre). Les opérateurs `++` et `--` se singularisent toutefois par le fait qu'ils admettent également une **syntaxe suffixée**, qui doit être prise en charge par une fonction spécifique.

Opérateur	Exemples
!	<code>bool n = ! true;</code>
-	<code>int a = - 2;</code>
+	<code>a = + 2;</code>
&	<code>int * ptr = & a;</code>
*	<code>* ptr = 4;</code>
~	<code>char c = ~ 'x';</code>
++	<code>++ a;</code> <code>a ++;</code>
--	<code>-- a;</code> <code>a --;</code>

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 operator !()
```

La définition d'une fonction membre surchargeant un opérateur ne présente qu'une seule particularité : son **nom complet** comporte 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;
4   //modification de resultat pour calculer le résultat de l'application de !
5   return resultat;
}
```

L'opérateur ! étant habituellement dénué d'effet, il est vraisemblable que l'objet auquel il est appliqué ne doit pas être modifié (la fonction membre est donc **constante** et utilise une variable locale pour calculer le resultat). De même, l'opérateur ! produit généralement un resultat du même type que son opérande, ce qui explique que la fonction est 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 {
3   //modifications de parametre pour calculer le résultat de l'application de !
4   return parametre; //renvoie la valeur obtenue
5 }
```

Bien que globale, la fonction opérateur va sans doute devoir accéder à des membres non publics de l'objet concerné. La classe de cet objet doit donc la déclarer amie.

Les opérateurs ++ et -- suffixés

Les opérateurs d'incrément 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 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 l'esprit 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. Leur 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 étant la même dans les deux cas (aucun argument pour une fonction membre, un seul argument, du type concerné, pour une fonction globale), 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**.

S'il s'agit de fonctions membre, les fonctions opérateur seront donc définies ainsi :

```
1 CTruc & CTruc::operator ++() //exécutée lorsque ++ est PREFIXE
2 {
3 //modification des variables membre pour réaliser l'effet associé à ++
4 return *this; //renvoie la valeur de l'opérande modifié
5 }
```

```
1 const CTruc CTruc::operator ++(int) //exécutée lorsque ++ est POSTFIXE
2 {
3 const 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, elles prendront la forme suivante :

```
1 CTruc & operator ++(CTruc & operande) //exécutée lorsque ++ est PREFIXE
2 {
3 //modification de operande pour réaliser l'effet associé à ++
4 return operande; //renvoie la valeur de l'opérande modifié
5 }
```

```
1 const CTruc operator ++(CTruc &operande, int) //exécutée lorsque ++ est POSTFIXE
2 {
3 const 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 prenant en charge l'usage préfixé d'un opérateur ++ ou -- renvoient une **référence à l'objet sur lequel elles sont appliquées**, et non simplement la nouvelle valeur de celui-ci. Les expressions utilisant l'opérateur préfixé deviennent ainsi des lvalue, ce qui autorise des choses comme ++unObjet.fonctionMembre() (incrémente l'objet, puis exécute la fonctionMembre). On préférera aussi que les opérateurs postfixés renvoient une **constante**, de façon à interdire la compilation d'expressions pathologiques telles que unObjet++++ (qui de toute façon, n'incrémenterait pas deux fois l'objet, puisque le second ++ serait appliqué à la valeur que l'objet avait *avant* la première incrémentation...)

Bien que les opérateurs d'incrémentement et de décrémentement n'admettent qu'un seul opérande, 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). Nous allons, bien entendu, retrouver cette implication de plusieurs objets dans le cas des fonctions prenant en charge les opérateurs qui admettent deux opérandes.

3 - 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 jouant des rôles symétriques, il faut généralement définir deux fonctions 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;
```

Fonction opérateur globale

Les opérateurs dyadiques susceptibles d'être surchargés par une fonction globale 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 aussi 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 7, 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 :

```
//fichier est un QTextStream prêt à recevoir des données
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 :

```
1 QTextStream & operator << (QTextStream &out, CTruc leTruc)
2 {
3     //il suffit ici d'envoyer chacun des membres dans le flux out à l'aide
4     //d'instructions du type :
5     out << leTruc.leMembre << "\n";
    //etc
    return out;
}
```

Bien entendu, cette fonction globale ne sera autorisée à accéder aux membres de son second paramètre que si elle bénéficie d'une relation d'amitié avec la classe CTruc.

Remarquez que la fonction `operator <<` ne se contente pas d'insérer du texte 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 qui lui a été transmise. En d'autres termes, la fonction ne se contente pas de réaliser l'effet souhaité, 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 :

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, outre les trente autres opérateurs dyadiques surchargeables,

Une fonction membre peut surcharger l'opérateur d'affectation.

Les fonctions définissant l'opérateur d'affectation renvoient habituellement une référence à l'objet au titre duquel elles sont exécutées, ce qui permet de chaîner les affectations :

```
trucUn = trucDeux = trucTrois;
```

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 :

```
1 bool CTruc::operator == (CTruc unAutre)
2 {
3   return (m_a == unAutre.m_a && m_b == unAutre.m_b);
4 }
```

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).

Une fois cet opérateur défini, il devient possible de comparer deux 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 écrire une fonction les effectuant. Il n'est pas indispensable de surcharger l'opérateur ==, puisqu'une fonction membre nommée estEgalA() et utilisant un argument du type concerné permettra d'obtenir le même effet :

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

La lisibilité de cette version est un peu moindre et, comme nous le savons, certaines fonctionnalités des conteneurs seront rendues disponibles par une fonction operator ==(), mais pas par une fonction estEgalA().

4 - Un cas particulier : l'opérateur "parenthèses"

Les fonctions operator () présentent deux caractéristiques remarquables, ayant trait l'une au nombre de leurs paramètres et l'autre à leur éventuelle exécution à la suite d'un appel implicite. Ces caractéristiques sont préférentiellement exploitées pour répondre à deux types de préoccupations bien différents, et la seconde est obtenue au prix d'une syntaxe tout à fait particulière. Une règle fondamentale s'applique cependant dans tous les cas :

L'opérateur () ne peut être défini que par une fonction membre.

Fonctionnoïdes

Les fonctions surchargeant l'opérateur "parenthèses" peuvent recevoir plusieurs paramètres. En effet, bien que cet opérateur soit dyadique, son opérande de droite est une liste, ce qui permet de fournir des valeurs permettant d'initialiser un nombre quelconque de paramètres. Si la classe CTruc possède une fonction opérateur du genre

```
void CTruc::operator () (int a, int b, int c, double x) {}
```

il devient possible d'écrire des choses comme

```
1 CTruc unTruc;
2 int a = 7;
3 unTruc(2, a, a+2, 2.4);
```

Comme le montre le fragment de code ci-dessus, l'utilisation normale de l'opérateur () ressemble à l'appel d'une fonction membre dépourvue de nom : le nom de l'instance concernée est directement suivi du couple de parenthèses encadrant les valeurs transmises. Il reste bien entendu possible d'appeler explicitement la fonction en utilisant son nom :

```
unTruc.operator () (2, a, a+2, 2.4);
```

Si l'opérateur "parenthèses" est parfois appelé "opérateur d'appel de fonction", on voit donc bien ce que cette expression a d'abusif : surcharger l'opérateur "parenthèses" revient simplement à simuler l'existence d'une ou plusieurs fonction(s) membre anonyme(s), et ceci n'a aucun effet sur le processus d'appel des autres fonctions. Une instance d'une classe surchargeant ainsi l'opérateur parenthèses se comporte donc comme une fonction, et les objets de ce genre sont souvent appelés "foncteurs", ou même "fonctionoïdes".

Opérateurs de transtypage

Il arrive très fréquemment que tout ou partie de l'information contenue dans une variable d'un certain type doive être transmise à une fonction disposant d'un paramètre d'un autre type. Lorsque le type attendu est une classe, on peut rendre possible la création automatique d'une valeur adéquate en dotant cette classe d'un constructeur de transtypage (cf. Leçon 10).

L'opérateur () permet d'obtenir un effet analogue mais, comme il est défini dans la classe source, il permet de traiter les cas où il n'est pas envisageable d'ajouter un constructeur de transtypage au type attendu (parce que ce n'est pas une classe, ou parce que c'est une classe faisant partie d'une librairie que l'on ne peut pas modifier, par exemple).

Imaginons, par exemple, une classe quelconque :

```
1 class CData
2 {
3 public:
4     //interface de la classe...
5 protected:
6     int a;
7     double x;
8 };
```

Si nous souhaitons disposer facilement d'une représentation textuelle de l'état d'une instance de cette classe, nous ne pouvons pas créer un constructeur de QString prenant pour argument une valeur de type CData (sauf, bien sûr, si nous sommes prêts à recompiler la librairie Qt et à vivre avec notre propre version de la classe QString...). L'instruction

```
//il existe un objet de type CData nommé uneData
QString texte(uneData);
```

peut néanmoins être rendue acceptable en ajoutant un opérateur de transtypage à CData.

La déclaration d'une telle fonction prend une forme inhabituelle, car le **type de la valeur renvoyée** (QString, dans notre exemple) abandonne sa place en tête de déclaration pour venir s'insérer entre le mot operator et les parenthèses complétant **le nom de la fonction** :

```
1 class CData
2 {
3 public:
4     //interface de la classe...
5     operator QString (); //comme si c'était une fonction nommée QString()
6 protected:
7     int a;
8     double x;
9 };
```

La définition de l'opérateur n'a, pour sa part, rien de bien original :

```
1 CData::operator QString()
2 {
3     QString texte = "a: %1, x:%2";
4     return texte.arg(a).arg(x);
5 }
```

Cette fonction peut être invoquée de différentes façons :

```
//il existe un objet de type CData nommé uneData
1 QString texte(uneData); //appel implicite de CData::operator QString()
2 texte = uneData.operator QString (); //appel explicite
```

```

3 texte = (QString) uneData; //transtypage explicite (syntaxe héritée du C)
4 texte = QString (uneData); //transtypage explicite (syntaxe C++ primitive)
5 texte = static_cast<QString> (uneData); //transtypage explicite (C++ actuel)

```

Si le contexte d'utilisation n'est pas propice au transtypage implicite, c'est incontestablement la forme 5 qui doit être préférée.

Les formes 3 et 4 ne sont destinées qu'à assurer la compatibilité des textes sources antérieurs à la norme ISO du C++. La forme 2 est admissible, mais très inhabituelle (comme tous les appels explicites de fonctions opérateur).

La même classe peut se trouver munie de plusieurs opérateurs de transtypage permettant d'obtenir des valeurs de divers types. Il arrive alors parfois qu'il soit nécessaire de choisir explicitement l'un des transtypages disponibles.

Supposons, par exemple, qu'une classe C dispose d'opérateurs de transtypage vers les types T1 et T2. Supposons également qu'il existe deux fonctions homonymes dont les signatures ne diffèrent que parce que l'une attend une valeur de type T1 et l'autre une valeur de type T2. Si l'on tente de passer une valeur de type C à l'une de ces fonctions, le compilateur se trouve face à un dilemme : il peut soit générer une valeur de type T1 et appeler la première fonction, soit générer une valeur de type T2 et appeler la seconde. Cette ambiguïté ne peut être résolue qu'en invoquant explicitement l'un des opérateurs `c::operator ()`.

La disponibilité de plusieurs opérateurs de transtypages différents dans une même classe crée une situation tout à fait exceptionnelle en C++, puisqu'il s'agit de fonctions homonymes ayant la même signature. C'est le seul cas où le langage admet des fonctions différentes ne se distinguant que par le type de la valeur qu'elles renvoient.

C'est sans doute pour souligner ce privilège exceptionnel conféré au type de la fonction que la syntaxe lui fait prendre une place inhabituelle dans la déclaration de celle-ci.

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. Le langage opère automatiquement, en cas de besoin, la conversion du type énuméré vers le type `int`, comme l'illustre le fait que

```
int x = JEUDI;
```

initialise la variable `x` avec la valeur 3. La conversion inverse n'est en revanche pas effectuée automatiquement, et l'instruction

```
EJour unJour = 3; //ERREUR !
```

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`. 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 }

```

Le type EJour n'étant pas une classe, il ne peut pas disposer de fonctions membre, ce qui nous empêche de le doter d'un véritable opérateur de transtypage. Cette restriction ne nous prive toutefois que de la possibilité de transtypage implicite, l'appel explicite de la fonction jour() assurant exactement le traitement requis.

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.

```

1 QTextStream & operator << (QTextStream & destination, EJour unJour)
2 {
3   switch (unJour)
4   {
5     case LUNDI : destination << "Lundi"; break;
6     case MARDI : destination << "Mardi"; break;
7     case MERCREDI : destination << "Mercredi"; break;
8     case JEUDI : destination << "Jeudi"; break;
9     case VENDREDI : destination << "Vendredi"; break;
10    case SAMEDI : destination << "Samedi"; break;
11    case DIMANCHE : destination << "Dimanche"; break;
12  }
13  return destination;
14 }

```

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().

```

1 EJour & operator ++ (EJour &unJour)           //++ préfixé
2 {
3   unJour = jour(unJour + 1);
4   return unJour;
5 }

1 EJour operator ++ (EJour &unJour, int)        //postfixé ++
2 {
3   EJour avant = unJour;
4   ++unJour; //utilisation de l'opérateur préfixé défini ci-dessus !
5   return avant;
6 }

```

La conversion automatique est 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 :

```
1 //fichier est un QTextStream prêt à recevoir des données
2 EJour aujourd'hui = MERCREDI;
3 int n;
4 for (n = 0 ; n < 9 ; n++)
    fichier << aujourd'hui ++ << ", ";
```

L'exécution d'une telle boucle insérerait dans fichier le texte suivant :

Mercredi, Jeudi, Vendredi, Samedi, Dimanche, Lundi, Mardi, Mercredi, Jeudi

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 EJour.

6 - Bon, c'est gentil tout ça, mais ça fait déjà 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 et l'opérateur == sont les seuls dont la surcharge est parfois réellement 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) Doter une classe d'un opérateur () peut ouvrir la possibilité de transtypages automatiques générant une valeur d'un type prédéfini à partir de la valeur d'une instance de cette classe.