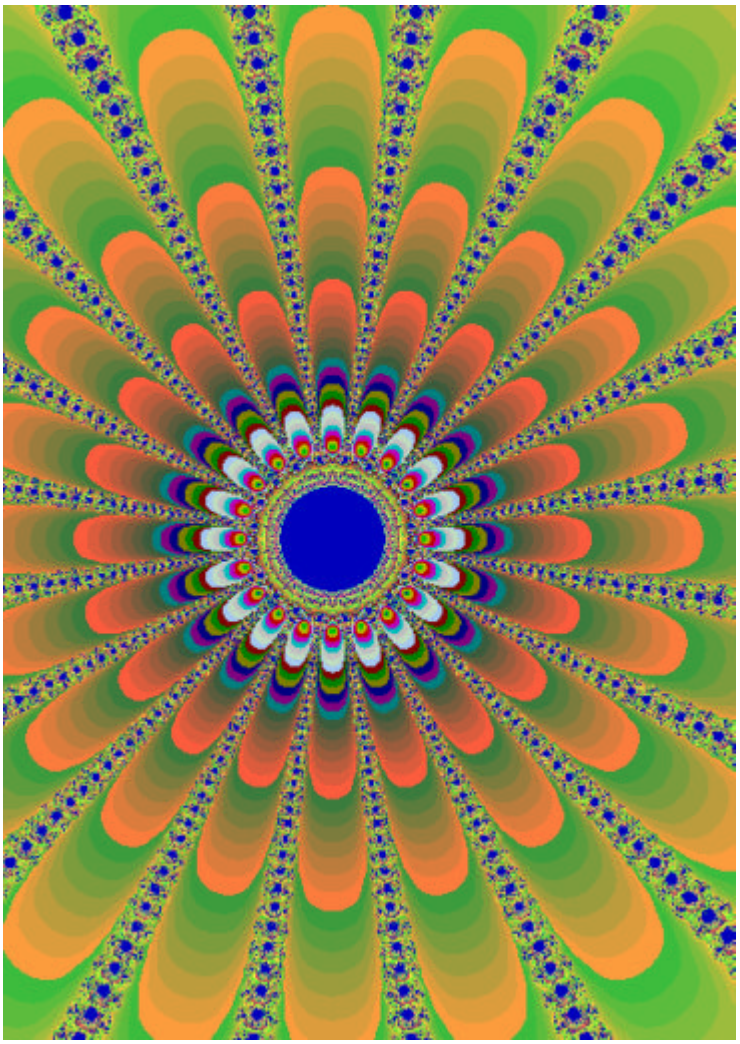
CHAPITRE 8

# *Opérateurs standard*



C++ définit quelques nouveaux opérateurs par rapport à C. La liste ci-dessous énumère les opérateurs définis dans C++, donc y compris les opérateurs déjà définis pour C. Certains de ces opérateurs acquièrent avec C++ un sens plus étendu, voire une complexité nouvelle. Dans ce tableau, les opérateurs sont classés par ordre de précédence, la colonne de gauche indique le niveau de précédence. Deux opérateurs de niveau 15, par exemple, sont de précédence égale, c'est-à-dire que l'ordre d'évaluation entre eux n'est pas défini.

La lettre figurant à droite du niveau de précédence indique l'associativité de l'opérateur. R signifie Right-To-Left (de droite à gauche) et L Left-To-Right (de gauche à droite). Ainsi, dire que l'opérateur = (assignation) a une associativité R signifie que si l'on écrit :

```
int  i, a, b;
     i = a = b = 3;
     // La ligne çi-dessous est identique à celle çi-dessus :
     i = (a = (b = 3));
```

| Niveau | Opérateur | Fonction |
|---|---|---|
| 17R | :: | Sélection de contexte global |
| 17L | :: | Sélection de contexte de classe |
| 16L | ->, . | Sélecteurs de membres |
| 16L | [] | Index de tableau |
| 16L | () | Appel de fonction |
| 16L | () | Construction de type |
| 15R | sizeof | Taille en bytes |
| 15R | ++, -- | Incrémentation, décrémentation |
| 15R | ~ | NOT binaire, bit à bit |
| 15R | ! | NOT logique |
| 15R | +, - | Signe plus, moins |
| 15R | *, & | Déréférence de pointeur, opérateur addresse |
| 15R | () | Conversion de type, casting |
| 15R | new, delete | Gestion de mémoire |
| 14L | ->*, .* | Sélecteurs de pointeurs de membres |
| 13L | *, /, % | Opérateurs multiplication, division, modulo |
| 12L | +, - | Opérateurs arithmétiques |
| 11L | <<, >> | Décalage bit à bit |
| 10L | <, <=, >, >= | Opérateurs relationnels |
| 9L | ==, != | Egalité, Inégalité |
| 8L | & | AND bit à bit |
| 7L | ^ | XOR bit à bit |
| 6L | \| | OR bit à bit |
| 5L | && | AND logique |
| 4L | \|\| | OU logique |
| 3L | ?: | IF arithmétique |

```
2R        =, *=,    Opérateurs d'assignation
          /=, %=,
          +=, -=,
          <<=,
          >>=,
          &=, |=,
          ^=
1L        ,         Opérateur virgule
```

Il est généralement jugé sage d'utiliser avec parcimonie les précédences d'opérateurs, donc de définir soi-même la précédence des opérateurs par des parenthèses appropriées, même quand elles sont superfétatoires. Le code source résultant est généralement plus clair, plus facile à lire, et pose moins de problèmes et de risques d'erreurs lorsque l'on modifie des expressions. Précisons encore que cette opinion n'est pas universellement partagée.

## 8.1 *Opérateurs arithmétiques et logiques*

### 8.1.1 *Addition +, Soustraction -, Multiplication *, Division /*

Ces opérateurs peuvent prendre comme arguments des entiers ou des réels, et fournissent un résultat entier si les deux opérandes sont entiers, réel dans le cas contraire. Dans le cas de la division entière, le résultat sera tronqué.

```
int       i, j, k;
i = 3;
j = 5;
k = i + j;     // k == 8
i = k * i;     // i == 24
j = i / j;     // j == 4
j = i - j;     // j == 20
```

### 8.1.2 *Modulo %*

Cet opérateur retourne le reste de la division entière. Il ne peut être utilisé que pour des nombres entiers.

```
int  j;
cout<<"Entrer un nombre entier "; cin>>j; cout<<endl;
if (j % 2) cout<<j<<" est un nombre impair";
else cout<<j<<" est un nombre pair";
cout<<endl;
```

### 8.1.3 *Egalité ==*

L'égalité se teste par l'opérateur ==.

```
int  a, b;
a = b = 3;
if (a == b)
     cout<<"Nous avons verifie une evidence"<<endl;
else
     cout<<"L'informatique n'est plus ce qu'elle etait..."<<endl;
```

Une faute souvent commise est de tester une valeur au moyen de l'opérateur = (habitude en provenance d'autres langages). Or, si dans beaucoup de cas, le compilateur acceptera le test, le résultat ne sera pas celui attendu :

```
int  a;
....
if (a = 3)
     {
     ... faire quelque chose...
     }
```

Ce qui est exécuté effectivement, c'est que a se voit assigner la valeur 3. Comme de surcroît, l'opérateur d'affectation (cf "Affectation simple =", page108) retourne une valeur, dans ce cas entière, et que les booléens sont implémentés par des entiers en C++, le compilateur se trouve satisfait. Le code correct devrait être :

```
int  a;
....
if (a == 3)        // Test logique d'égalité
     {
     ... faire quelque chose...
     }
```

Certains compilateurs émettent un warning dans ce genre de situation (Siemens-Nixdorf, par exemple). Pour supprimer ce warning dans le cas où l'assignation est effectivement le but recherché, on écrira :

```
int  a;
....
if ((a = 3))       // Assignation, puis test logique != 0
     {
     ... faire quelque chose...
     }
```

### 8.1.4 *Comparaisons (>, >=, <, <=)*

La signification de ces opérateurs est relativement aisée à comprendre et ne comporte pas de difficultés particulières.

```
int       a, b, c;
a = 3;
b = 4;
c = 5;
if (b++ < c)
     {
     cout<<b<<" est inferieur a "<<c<<endl;
     // Imprime : 5 est inferieur a 5 (effet de bord !!!)
     }
if (b >= a)
     {
     cout<<b<<" est superieur ou egal a "<<a<<endl;
     // Imprime : 5 est superieur ou egal a 3
     }
```

Les opérateurs de comparaison, tout comme l'opérateur de test d'égalité, ont une valeur de retour de type entier.

### 8.1.5 *Opérateurs logiques (&&, ||)*

Ces opérateurs correspondent au ET logique (&&, AND) et au OU logique (||, OR). Ces

opérateurs évaluent d'abord l'opérande de gauche, puis celui de droite. Si la valeur de l'opérande de gauche rend toute possibilité de succès ( = TRUE) du résultat impossible, alors le membre de droite n'est pas évalué.

Cette non-évaluation peut dans certains cas provoquer des erreurs difficiles à cerner, en particulier si l'on fait usage d'effets de bord. L'effet de bord aura lieu ou non en fonction des variables impliquées dans l'opération. On ne peut que répéter que les effets de bord doivent être utilisés avec parcimonie, dans la mesure où on les utilise.

```
int  a, b, c;
a = 1;
b = 2;
c = 3;
if ((a <= b) && (c > b))
               // Ici, les deux conditions sont evaluees
     {
     // Code execute
     }
else
     {
     // Code non execute
     }

if ((a <=  b) || (a > b))
               // Seule la premiere expression est evaluee
     {
     // Code execute
     }
else
     {
     // Code non execute
     }
```

Attention à ne pas confondre && avec l'opérateur ET binaire &, et || avec l'opérateur OU binaire |.

### 8.1.6 *Opérateur logique ! et !=*

Cet opérateur implémente la négation logique (et non pas l'inversion des bits, cf "Opérations binaires (~, &, ^, |)", page110). Il s'agit de l'opération de négation :

```
while (a != b) { ... }          // while "a not equal b"
while (!(a == b)) { ... }       // même test que précédemment
if (!(a > b)) { ... }           // ou if (a <= b) !!!
```

## 8.2 *Affectations*

### 8.2.1 *Affectation simple =*

L'instruction d'affectation a pour effet d'assigner à l'opérande de gauche la valeur de l'opérande de droite. A noter que l'opérateur d'affectation retourne une valeur : le type de cette valeur est le type de l'opérande de gauche, alors que la valeur de retour est la valeur de l'opérande de gauche après affectation.

### 8.2.2 *Affectation avec opération arithmétique (+=, -=, *=, /=)*

Il s'agit pour ces opérateurs d'une contraction de notation : ainsi

```
a    +=   b;        // est équivalent à
a    =    a + b;
```

D'une manière générale :

```
a    op=  b         ===>      a = a op b;
```

### 8.2.3 *Incrémentation (++) et décrémentation (--)*

Ces deux opérateurs incrémentent ou décrémentent une variable. Il y a deux variantes :

- la post-incrémentation et post-décrémentation
- la pré-incrémentation et prédécrémentation

```
int      unInt = 0;
unInt++;               // Post-incrémentation. Ajoute 1 à unInt et
                       // retourne l'ancienne valeur ( == 0 )
++unInt;               // Pré-incrémentation. Ajoute 1 à unInt et
                       // retourne la nouvelle valeur ( == 2 )
--unInt;               // Pré-decrémentation. Ote 1 à unInt et
                       // retourne la nouvelle valeur ( == 1 )
unInt--;               // Post-decrémentation. Ote 1 à unInt et
                       // retourne l'ancienne valeur ( == 1 )
```

ou encore :

```
int      a, b;
a = 1;
b = a++;               // b == 1
b = ++a;               // b == 3, puisque a = 2 apres ++
```

Cette notation permet de compacter certaines expressions. En réalité, la volonté d'optimiser le code n'est plus une justification pour utiliser cette notation, comme à l'origine, la plupart des compilateurs étant sans autre capables de détecter une incrémentation à partir d'une expression du type

```
a = a + 1;
```

et de générer un code adéquat en conséquence.

Ces opérateurs sont souvent utilisés en combinaison avec des pointeurs. Dans le cas de pointeurs, l'incrément (ou décrément) reste 1, mais cette quantité est à interpréter comme un objet pointé.

Pour des pointeurs sur des caractères (char*), le pointeur sera incrémenté (décrémenté) de 1 byte à chaque fois, alors que sur une machine gérant des entiers à 32 bit, un pointeur sur des entiers sera incrémenté (décrémenté) de 4 byte à chaque fois. Plus généralement, un pointeur sur un objet de taille N bytes sera incrémenté de N à chaque opération. Ainsi, une manière fréquemment utilisée pour calculer la longueur d'une chaîne de caractères peut-elle s'écrire :

```
int  stringLength(char *str)
     {
     for (int length = 0; *str++; length++);
     return length;
     }
```

## 8.3 *Opérateurs sur des bits*

### 8.3.1 *Décalage (<<, >>)*

L'opérateur de décalage permet le décalage vers la gauche ou vers la droite d'une suite de bit. La suite décalée correspond à l'opérande de gauche, le décalage à l'opérande de droite :

```
int  bitWord = 9;
bitWord>>2;        // bitWord vaut 2.
```

Attention à ne pas faire de confusion avec les opérateurs surchargés << et >> de iostream.(Voir à ce sujet "Surcharge d'opérateurs", page162).

### 8.3.2 *Opérations binaires (~, &, ^, |)*

L'opérateur ~ correspond au NOT binaire. Il effectue une inversion de tous les bit de son opérande.

```
unsigned char c = 127;
c = ~c;                    // c = 0;
```

L'opérateur & correspond au ET logique bit à bit de deux expressions. L'opérateur ^ correspond à l'opération OU EXCLUSIF, et l'opérateur | correspond au OU bit à bit. Attention, là encore, à ne pas faire de confusion entre les opérateurs binaires et les opérateurs logiques.

### 8.3.3 *Affectation avec opération binaire (|=, &=, ^=)*

Ce type d'affectation composée correspond à l'affectation composée arithmétique. Il s'agit également d'une contraction de notation : ainsi

```
a    &=   b;       // est équivalent à
a    =    a & b;
```

D'une manière générale :

```
a    op=  b        ===>      a = a op b;
```

## 8.4 *Opérateur de référence et de déréférence (& et *)*

L'opérateur de référence, placé à **gauche** d'un identificateur, retourne l'adresse de la position-mémoire où est stocké l'objet correspondant à l'identificateur.

```
int  *ptr;
int  anInt;
ptr = &anInt;      // ptr prend pour valeur l'adresse de anInt.
```

L'opérateur de déréférence retourne la valeur sur laquelle pointe un objet. Ansi pour le pointeur ptr du paragraphe précédent :

```
if (*ptr == anInt) { };      // Vrai !
```

Ces notations peuvent receler certains pièges : admettons que nous voulions effectuer la division d'un entier quelconque (ici, int1) par la valeur sur laquelle pointe un pointeur sur des entiers appelé ptr : On écrira donc :

```
int      int1;
int*     ptr;
int  resultatDeLaDivision = int1/*ptr;
```

Ce que le compilateur sanctionnera par un message d'erreur (ou plus probablement une série de messages d'erreurs). Il se trouve que la séquence /* correspond à un début de commentaire, et que le préprocesseur a supprimé du code. Le compilateur n'a donc pas eu l'occasion de traiter tout le texte suivant l'identificateur int1, avec les résultats que l'on peut imaginer. Une solution à ce problème serait l'utilisation de parenthèses, ou d'espaces :

```
int  resultatDeLaDivision = int1 / *ptr;
int  resultatDeLaDivision = int1 / (*ptr);
```

Ce qui ne nuit pas non plus à la lisibilité.

## 8.5 *Autres opérateurs*

### 8.5.1 *Opérateur sizeof*

Cet opérateur permet de calculer la taille en octets d'une expression, ou d'un type. Il retourne une quantité de type `size_t` (qui se résoud généralement en `unsigned int`) représentant le nombre d'octets occupé par une variable ou un type.

```
cout<<sizeof(char)<<endl;        // affiche 1
cout<<sizeof(int)<<endl;         // affiche 4 sur une machine 32 bit
```

### 8.5.2 *Opérateur virgule (,)*

L'opérateur virgule retourne la valeur de l'opérande de droite. Il est utilisé pour regrouper plusieurs expressions dont seule la valeur de la dernière (la plus à droite) importe. Quelle utilité ? Aucune, si ce n'est que l'expression (les expressions) de gauche peuvent également, par effet de bord, avoir un effet. En fait, il permet de compacter l'écriture du programme.

L'utilisation de l'opérateur virgule, par le fait qu'elle favorise les effets de bord, et donc nécessite souvent une documentation séparée, est à déconseiller, sauf dans certains cas simples.

### 8.5.3 *Opérateur ? :*

L'opérateur **`?:`** est le seul opérateur ternaire de C++. Sa forme générale est

*`condition ? expression1 : expression2`*

Si `condition` est évalué à TRUE (différent de 0), alors c'est `expression1` qui est exécutée, `expression2` dans le cas contraire. Il s'agit, ici aussi, d'une manière de compacter le code :

```
int      min(int a, int b)
     {
     return (a < b ? a : b);
     }

int      otherMin(int a int b)
     {
     if (a < b) return a;
     return b;
          // Réalise la même chose.
     }
```

## 8.6 *Test*

1. Quel est le résultat des opérations suivantes ?

```
int  a = 3, b = 4, c = 5;
if ((a < c) && (a < ++b)) a++;
if (a <= b) c--;
if (a == c) b++;
```

2. Quelle opération effectue le code suivant ?

```
#include <iostream.h>

void      uneProcedure(int une Valeur)
     {
     int i = 1, k;
     for (k = 0; k < sizeof(int) * 8; k++, i = i<<1)
          cout<<((uneValeur & i) > 0 ? '1' : '0');
     cout<<endl;
     }
```

3. Quelle est la sortie du programme suivant ?

```
#include <iostream.h>

int       uneProcedure(int une Valeur)
     {
     return (uneValeur = 3);
     }

void main()
     {
     cout<<uneProcedure(4)<<endl;
     cout<<uneProcedure(3)<<endl;
     cout<<uneProcedure(2)<<endl;
     }
```