



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

## Apprendre C++ avec Qt : Leçon 5 Opérateurs à effet et fonctions avec paramètres

1 - Opérateurs à effet .....	2
Les opérateurs "calculAffectation" .....	2
Les opérateurs "suivant" et "précédent" .....	3
2 - Définition de fonctions utilisant des paramètres.....	4
Paramètres et variables locales .....	4
Initialisations .....	4
Fonctions utilisant plusieurs paramètres .....	5
3 - Valeur par défaut d'un paramètre .....	5
Déclaration de valeurs par défaut.....	6
Utilisation des valeurs par défaut.....	6
4 - Paramètres permettant à une fonction d'accéder à des objets qui ne lui appartiennent pas .....	7
Paramètres de type "référence à..." .....	7
Paramètres de type "pointeur sur..." .....	8
Constance des paramètres .....	8
5 - Bon, c'est gentil tout ça, mais ça fait déjà 7 pages. Qu'est-ce que je dois vraiment en retenir ?.....	9

Les structures de contrôle du flux d'exécution que la Leçon 4 a mis à notre disposition nous permettent d'envisager des programmes plus ambitieux que ceux que nous avons pu écrire jusqu'à présent. Pour compléter encore notre arsenal, il nous faut maintenant faire connaissance avec quelques nouveaux opérateurs et apprendre à utiliser des paramètres.

## 1 - Opérateurs à effet

Dans la Leçon 03, nous avons introduit les opérateurs correspondant aux quatre opérations arithmétiques habituelles : +, -, \* et /, ainsi que l'opérateur % (modulo). Le langage C++ propose aussi des opérateurs arithmétiques qui ont un effet sur un de leurs opérandes.

### Les opérateurs "calculAffectation"

Les cinq opérateurs arithmétiques que nous connaissons permettent de combiner deux opérandes pour obtenir un *résultat*. L'évaluation d'une expression ainsi formée reste, bien entendu, sans *effet* sur les variables qui servent éventuellement à spécifier l'un ou l'autre des opérandes. Lorsqu'une variable doit recevoir le résultat obtenu, il nous faut donc utiliser l'opérateur d'*affectation* en plus de celui qui spécifie le *calcul* à effectuer :

```
index = index + 1; //+ produit le résultat et = produit l'effet
```

Lorsque, comme dans l'exemple ci-dessus, la même variable sert tout d'abord à spécifier l'un des opérandes puis, dans un second temps, à recueillir le résultat obtenu, le *nom* de cette variable figure de part et d'autre de l'opérateur d'affectation. Dans cette situation, il est possible d'utiliser un seul opérateur, qui indique à la fois la nature du calcul à effectuer et le fait que le résultat obtenu doit être stocké dans la variable :

```
index += 1; //+= produit le résultat et l'effet
```

L'intérêt de cette forme d'expression est qu'elle évite d'avoir à désigner deux fois la *cible de l'affectation*. L'avantage peut sembler bien mince lorsque cette cible est désignée directement par un nom de variable, mais il devient nettement plus tangible lorsqu'on a affaire à une expression plus complexe. Une instruction telle que

```
machin.fonction()->valeur = machin.fonction()->valeur + 1 ;
```

est encore plus désagréable à lire qu'à écrire, car elle nécessite un examen attentif pour vérifier que le résultat du calcul est stocké dans la zone de mémoire qui a été utilisée pour déterminer la valeur du premier opérande de l'addition. L'idée générale, "*ajouter 1 quelque part*", est bien plus visible si l'on écrit

```
machin.fonction()->valeur += 1 ;
```

Les cinq opérateurs arithmétiques élémentaires se doublent donc d'autant d'opérateurs combinant calcul et affectation, et on peut écrire des choses comme

```
index += decalage; //équivalent à index = index + decalage ;
index -= 3;       //équivalent à index = index - 3 ;
index *= 2 + k;   //équivalent à index = index * (2 + k) ;
index /= 2;       //équivalent à index = index / 2 ;
index %= 7;       //équivalent à index = index % 7 ;
```

Ces opérateurs utilisent une notation faisant intervenir deux caractères, ce qui évoque clairement l'idée qu'ils permettent à la fois de calculer le résultat et d'obtenir un effet d'affectation. Il n'en reste pas moins que chacun de ces couples de caractères constitue un symbole unique, qui désigne l'opérateur en question. Il n'est pas plus envisageable d'écrire == à la place de -= qu'il ne serait envisageable d'écrire truner au lieu de return à la fin d'une fonction. Attention, toutefois : si l'usage du mot truner n'a aucune chance de passer inaperçu du compilateur (qui ne saura qu'en faire...), une ligne telle que

```
variable -= 3.5;
```

ne déclenchera pas le moindre avertissement, puisqu'elle ordonne tout simplement d'*affecter* la *valeur* -3.5 à la variable.

### Les opérateurs "suivant" et "précédent"

L'opération consistant à ajouter (ou soustraire) un au contenu d'une variable est tellement fréquente qu'elle fait l'objet d'une notation particulière. On écrira, par exemple

```
index++; //équivalent à index = index + 1; ou à index += 1;
valeur--; //équivalent à valeur = valeur - 1; ou à valeur -= 1;
```

Les opérateurs ++ et -- peuvent être placés avant la variable sur laquelle ils opèrent (on les dit alors *préfixés*) ou après celle-ci (on les dit alors *suffixés*). Les deux instructions précédentes pourraient donc être remplacées par

```
++index; //équivalent à index = index + 1; ou à index += 1;
--valeur; //équivalent à valeur = valeur - 1; ou à valeur -= 1;
```

Si l'*effet* obtenu est le même dans les deux cas, ce n'est pas le cas de la *valeur* de l'expression : lorsque l'opérateur est suffixé, l'expression a pour valeur le contenu qu'avait la variable *avant* application de l'opérateur, alors que lorsque l'opérateur est préfixé, l'expression a pour valeur celle obtenue *après* application de l'opérateur.

Autrement dit, lorsque l'opérateur est **devant** son opérande, **il est appliqué avant** de déterminer la valeur de l'expression, alors que lorsqu'il n'apparaît qu'**après** son opérande, **il est appliqué après** avoir déterminé la valeur de l'expression.

Bien entendu, dans les exemples précédents, seul l'effet est important, puisqu'il n'est fait aucun usage des valeurs obtenues à la suite de l'évaluation des expressions. Il n'en va pas toujours ainsi, et ces opérateurs sont (trop ?) souvent employés dans des expressions où la valeur qu'ils produisent sert d'opérande à un autre opérateur. Ainsi, par exemple, après

```
int a = 2;
int b = a++; //équivalent à int b = a ; a = a + 1; donc b contient 2
```

la variable b contient 2, puisque la valeur de l'expression a++ est celle contenue dans a *avant* incrémentation. En revanche, après

```
int a = 2;
int b = ++a; //équivalent à a = a + 1 ; int b = a; donc b contient 3
```

la variable b contiendra 3 puisque la valeur de l'expression ++a est celle contenue dans a *après* incrémentation. Bien entendu, dans un cas comme dans l'autre, la variable a contient finalement 3, puisque l'effet de l'opérateur est le même, qu'il soit préfixé ou suffixé.

S'il est vrai que les opérateurs ++ et -- permettent parfois une concision qui favorise l'intelligibilité du code, il existe des circonstances où il convient de s'abstenir de les utiliser. C'est en particulier le cas des expressions où la même variable figure à plusieurs reprises : comme la norme ISO du langage C++ n'impose aux compilateurs aucun ordre particulier d'évaluation des sous-expressions, appliquer un opérateur d'incrément à une telle variable produit des résultats imprévisibles.

C'est à dire que, même si un compilateur donné a de bonnes chances de toujours donner le même résultat (et encore, même ceci n'est pas absolument garanti...), il est tout à fait normal que des compilateurs différents donnent des résultats différents.

Considérons l'exemple suivant :

```
int a = 2;
int b = ++a + a; //b = 6, 5 ou autre chose ?
```

L'expression permettant de calculer la valeur d'initialisation de b est une **addition** faisant intervenir **deux** sous-expressions. Il existe au moins deux façons d'évaluer cette addition. Soit on commence par la gauche et l'évaluation de ++a se traduit par l'incrément de cette variable et produit la valeur 3. L'évaluation de a donnera donc ensuite 3, et le résultat de l'**addition** sera 6. Soit on commence par la droite et l'évaluation de a donne 2. L'évaluation de ++a donne ensuite la valeur 3, et conduit à incrémenter cette variable, mais ceci reste sans effet sur l'évaluation de a, qui a déjà eu lieu. Le résultat de l'**addition** est donc 5. Conclusion :

N'utilisez **JAMAIS** les opérateurs ++ et -- dans des expressions où la variable à laquelle ils s'appliquent est utilisée plusieurs fois.

## 2 - Définition de fonctions utilisant des paramètres

Une façon assez simple de décrire rapidement ce que sont les paramètres d'une fonction est de dire qu'il s'agit de variables locales qui ne sont pas définies dans le bloc de code, mais entre les parenthèses qui suivent immédiatement le nom de la fonction.

### Paramètres et variables locales

Considérons, par exemple, une fonction définie de la façon suivante :

```
1 //exemple de fonction utilisant un paramètre
2 void maFonction(int sonParametre)
3 {
4   int saVariableLocale;
   //ici prennent place les instructions décrivant le traitement effectué
}
```

Le bloc de code qui constitue le corps de `maFonction()` dispose de deux variables de type `int`, respectivement nommées `sonParametre` et `saVariableLocale`. Les règles générales d'utilisation qui s'appliquent à ces deux variables sont les mêmes : `maFonction()` peut en disposer librement (c'est à dire qu'elle peut en utiliser et en changer le contenu), mais aucune autre fonction n'a connaissance de l'existence de ces variables.

Si une autre fonction utilise aussi des variables portant ces noms, il s'agit d'une simple homonymie, qui peut éventuellement dérouter un lecteur novice, mais ne saurait entraîner quelque conséquence que ce soit du point de vue du fonctionnement du programme. En clair :

La façon dont une fonction baptise ses variables locales et ses paramètres ne regarde qu'elle.

Si les paramètres sont de simples variables locales, quel intérêt présentent-ils ? Si aucune autre fonction ne peut accéder aux paramètres d'une fonction donnée, en quoi s'agit-il d'un mécanisme de transmission d'information entre fonctions ?

L'intérêt des paramètres réside dans une particularité d'apparence assez anodine : la façon dont ces variables<sup>1</sup> sont initialisées.

### Initialisations

Lorsqu'une variable locale est initialisée, la valeur utilisée est spécifiée dans le corps même de la fonction. A chaque exécution de la fonction, c'est donc la même valeur qui est utilisée. Modifions, par exemple, la définition de notre fonction de la manière suivante :

```
1 void maFonction(int sonParametre)//exemple de fonction utilisant un paramètre
2 {
3   int saVariableLocale = 36;
   //ici prennent place les instructions décrivant le traitement effectué
}
```

Nous sommes absolument certains que, à chaque exécution de `maFonction()`, une variable de type `int` nommée `saVariableLocale` sera créée et **prendra 36 comme valeur initiale**.

Le cas des paramètres est totalement différent : c'est la fonction appelante qui décide de la valeur avec laquelle ils vont être initialisés. Un exemple d'appel de `maFonction()` pourrait être

```
maFonction(12);
```

Dans ce cas, c'est la valeur 12 qui serait utilisée pour initialiser la variable `sonParametre`, alors que dans le cas où l'appel serait

```
maFonction(29);
```

c'est bien entendu la valeur 29 qui serait utilisée.

<sup>1</sup> L'usage du mot "variable" dans ce contexte n'est pas parfaitement orthodoxe. Si vous vous adressez à un puriste, dites "paramètre formel", ça lui fera plaisir et ça vous permettra de ne pas être trop surpris si vous découvrez un jour des différences subtiles entre paramètres et variables locales qui n'auraient pas été évoquées ici.

La *valeur* qui se trouve ainsi *transmise* à `maFonction()` peut ne pas être spécifiée littéralement mais être elle-même contenue dans une variable. Dans l'exemple suivant, `maFonction()` est appelée 100 fois, son paramètre se trouvant initialisé chaque fois avec une valeur plus élevée.

```
1 int i;  
2 for (i = 0 ; i < 100 ; i = i + 1)  
3     maFonction(i);
```

Si la coïncidence des noms du paramètre et de la variable éventuellement utilisée pour spécifier la valeur transmise lors de l'appel n'a strictement aucune importance, il n'en va bien entendu pas de même pour celle de leur type. Il ne serait quand même pas très raisonnable de transmettre une valeur décimale à `maFonction()`, puisque nous savons que cette valeur est destinée à initialiser une variable de type `int`.

Lorsque le contenu d'une variable est passé à une fonction, c'est le type de cette variable (et non simplement sa valeur actuelle) qui doit être compatible avec le type du paramètre.

L'exemple suivant provoquerait donc un message d'avertissement de la part du compilateur, qui ne peut généralement pas savoir si la valeur de `unNombre` est ou non entière au moment de l'appel de `maFonction()` :

```
1 double unNombre = 5 ;  
2 maFonction(unNombre); //problème !
```

### Fonctions utilisant plusieurs paramètres

Lorsqu'une fonction utilise **plusieurs paramètres**, leurs descriptions sont séparées par des **virgules** :

```
1 //exemple de fonction utilisant trois paramètres  
2 void maFonction(int parametreUn, char parametreDeux, float parametreTrois)  
3 {  
4     int saVariableLocale = 36;  
5     //ici prennent place les instructions décrivant le traitement effectué  
6 }
```

L'appel d'une telle fonction nécessite évidemment que des valeurs appropriées soient transmises pour initialiser chacun des paramètres. On peut, par exemple, écrire :

```
maFonction(7, 's', 2.18);
```

C'est simplement l'ordre des valeurs transmises qui assure leur mise en correspondance avec les paramètres de la fonction : la première valeur est attribuée au premier paramètre de la liste, la deuxième valeur au deuxième paramètre, et ainsi de suite.

## 3 - Valeur par défaut d'un paramètre

Nous avons vu que l'appel d'une fonction qui utilise des paramètres exige normalement de fournir une valeur appropriée pour initialiser chacun de ceux-ci. Cette exigence est un inconvénient, particulièrement lorsqu'on cherche à rendre les fonctions réutilisables (comme dans le cas des bibliothèques, qui serviront dans le cadre de nombreux projets différents).

En effet, pour rendre une fonction capable de traiter de nombreux cas particuliers, il est souvent nécessaire de la doter de nombreux paramètres permettant, justement, de spécifier quel cas particulier se présente lors d'un appel donné. Cette longue liste de paramètres présente deux inconvénients évidents :

- Le nombre de valeurs à fournir lors de chaque appel augmente, ce qui alourdit le code et multiplie les risques d'erreurs.
- Dans le cas général (qui est souvent le plus fréquent), la plupart de ces paramètres ne servent à rien, puisque leur rôle est justement de décrire les particularités d'un cas qui serait exceptionnel. Le programmeur est alors conduit à prendre en compte une multitude de détails qui ne le concernent pas à ce moment là, ce qui rend la fonction désagréable à utiliser. En réaction, il est tenté de ne pas utiliser la fonction qui existe, mais d'en réécrire une version plus simple, ce qui multiplie à nouveau les risques d'erreurs.

Le langage C++ propose deux mécanismes susceptibles de répondre à ce type de préoccupations : la surcharge et les valeurs par défaut.

La **surcharge** (overloading, en anglais) est la définition de plusieurs fonctions ayant le même nom, mais se distinguant par leurs paramètres. Ce mécanisme n'est pas spécifiquement conçu pour éviter les listes d'arguments pléthoriques, mais on peut envisager de l'utiliser dans des cas où il s'agit simplement de réduire le nombre des valeurs à transmettre.

Le principe des **valeurs par défaut** est très simple : il s'agit simplement de prévoir, lors de la conception d'une fonction, une valeur qui sera utilisée pour initialiser un paramètre lorsque la fonction appelante s'abstiendra d'en fournir une.

Les bibliothèques ont souvent recours à ces mécanismes, dont un des aspects les plus séduisants est que le programmeur qui *utilise* la bibliothèque n'a pas à s'en occuper : il bénéficie de la souplesse ainsi obtenue, les problèmes ayant été réglés par celui qui a *écrit* les fonctions.

### Déclaration de valeurs par défaut

Pour attribuer une valeur par défaut à un paramètre, il suffit, dans la **déclaration** de la fonction, d'indiquer cette valeur à la suite d'un signe "=" placé après le nom du paramètre<sup>2</sup>.

```
//cette ligne figure normalement dans un fichier .h
void maFonction(int sonParametre = 4); //DECLARATION de la fonction
```

La **définition** de la fonction, en revanche ne mentionne normalement pas de valeur par défaut.

```
1 //ces lignes figurent normalement dans un fichier .cpp
2 void maFonction(int sonParametre) //DEFINITION de la fonction
3 {
  //... définitions de variables locales, instructions
}
```

Lorsqu'une fonction comporte plusieurs arguments, ils peuvent être plusieurs à avoir des valeurs par défaut. Il existe toutefois une restriction : dès lors que l'un des paramètres possède une valeur par défaut, tous les paramètres suivants doivent en avoir une également.

```
//cette ligne figure normalement dans un fichier .h
void maFonction(float p1, int p2 = 4, char p3 = 'x');
```

La déclaration suivante, en revanche n'est pas admissible, car le premier paramètre a une valeur par défaut alors que le **deuxième** n'en a pas :

```
//cette ligne figure normalement dans un fichier .h
void maFonction(float p1 = 0.2, int p2, char p3 = 'x'); //ERREUR !
```

### Utilisation des valeurs par défaut

Lorsqu'une fonction comporte des valeurs par défaut, on les utilise simplement en faisant comme si le paramètre concerné n'existait pas. Si une classe a été définie ainsi :

```
1 class CMaClasse //extrait du fichier maClasse.h
2 {
3 public:
4     void maFonction(int sonParametre = 4);
5 };
```

et qu'il en existe une instance définie ainsi :

```
CMaClasse uneInstanceDeMaClasse;
```

la fonction membre peut être appelée comme ceci :

```
uneInstanceDeMaClasse.maFonction(); //son paramètre sera initialisé à 4
```

aussi bien que comme cela :

```
uneInstanceDeMaClasse.maFonction(8); //son paramètre sera initialisé à 8
```

<sup>2</sup> Ou directement après le type du paramètre, si vous avez pris la déplorable habitude de déclarer les fonctions sans nommer les paramètres.

Il existe toutefois une restriction à l'usage des valeurs par défaut : si l'un des paramètres a été privé de valeur lors d'un appel, tous les paramètres suivants doivent l'être également.

En clair : il n'est pas possible de "sauter" un paramètre (pour adopter sa valeur par défaut) et de reprendre ensuite l'énumération des valeurs devant être adoptées par les suivants. Ceci explique pourquoi fournir une valeur par défaut pour un paramètre implique d'en fournir aussi pour tous les suivants. En effet, l'adoption de la valeur par défaut d'un paramètre précédant un paramètre dépourvu de valeur par défaut rendrait impossible le passage d'une valeur destinée à initialiser ce second paramètre, qui ne disposerait donc ni d'une valeur par défaut ni d'une valeur transmise lors de l'appel.

Si la classe a été définie ainsi

```
1 class CMaClasse    //extrait du fichier maClasse.h
2 {
3 public:
4     void maFonction(char p1 = 'x', float p2 = 7.8);
5 };
```

maFonction() pourra être appelée comme ceci :

```
uneInstanceDeMaClasse.maFonction('k',12); //p1 vaudra 'k' et p2 vaudra 12
uneInstanceDeMaClasse.maFonction('w');    //p1 vaudra 'w' et p2 vaudra 7.8
uneInstanceDeMaClasse.maFonction();       //p1 vaudra 'x' et p2 vaudra 7.8
```

mais pas comme cela :

```
uneInstanceDeMaClasse.maFonction(3.14);
//on ne peut pas accepter la valeur par défaut de p1 et refuser celle de p2
```

#### 4 - Paramètres permettant à une fonction d'accéder à des objets qui ne lui appartiennent pas

Nous avons vu que les paramètres, tout comme les variables locales, sont des objets qui appartiennent exclusivement à la fonction dans laquelle ils sont définis. La transmission à une fonction d'une valeur qu'elle va utiliser pour initialiser un de ses paramètres permet cependant de fournir à la fonction un moyen d'accéder à des objets appartenant à la fonction appelante.

##### Paramètres de type "référence à..."

Nous savons que la création d'une référence permet d'associer un nouveau nom à l'objet qui est utilisé pour initialiser la référence. Si une fonction utilise une **référence** comme paramètre, le **nom** de celle-ci devient une étiquette désignant l'objet avec lequel a été initialisé ce paramètre. Comme **l'objet en question** est fourni par la fonction appelante(), la référence devient, pour la fonction appelée(), un moyen de désigner (et donc **d'agir sur**) un objet qui appartient à la fonction appelante().

```
1 void appelante()
2 {
3     int maVariableAMoi = 4;
4     appelee(maVariableAMoi); //maVariableAMoi contient maintenant 5
5 }
```

```
1 void appelee(int & objetEtranger)
2 {
3     objetEtranger += 2;
4 }
```

La fonction appelée() peut, bien entendu, l'être autant de fois que nécessaire et l'objet utilisé pour initialiser son paramètre référence n'est pas forcément toujours le même.

Un paramètre étant un objet local à la fonction, une **nouvelle référence** est créée chaque fois que la fonction est exécutée. Le fait que cette référence ne désigne pas forcément le même objet lors de chaque exécution ne constitue donc pas une violation du principe selon lequel une référence, une fois créée, désignera toujours le même objet. Ce sont en fait des références *différentes* (correspondant à une même définition dans le texte source, mais n'existant pas au même moment pendant l'exécution) qui sont susceptibles de désigner des objets différents.



### Paramètres de type "pointeur sur..."

Lorsqu'une fonction dispose d'un paramètre de type "pointeur sur...", la fonction appelante doit, évidemment, fournir l'adresse d'un objet du type adéquat pour initialiser ce paramètre. Si la fonction appelée **déréférence** ce pointeur, elle est en mesure d'**agir sur** l'objet dont on lui a passé l'adresse, exactement comme si elle disposait d'une référence désignant cet objet :

```
1 void appelante()  
2 {  
3   int maVariableAMoi = 4;  
4   appelee(& maVariableAMoi);  
   //maVariableAMoi contient maintenant 5  
5 }
```

```
1 void appelee(int * monPointeur)  
2 {  
3   * monPointeur += 2;  
4 }
```

L'effet obtenu en passant une adresse est donc tout à fait équivalent à celui produit par l'utilisation d'une référence, aux détails de notation près. Dans le cas d'un pointeur, la fonction appelante doit explicitement passer une adresse, ce qui attire l'attention des programmeurs sur le fait que la fonction appelée est en mesure de modifier l'objet concerné. Dans le cas d'une référence, en revanche, le code appelant se présente exactement de la même façon que si la fonction appelée ne recevait que la valeur de la variable mentionnée, et cette ambiguïté pousse certains programmeurs à préférer utiliser un pointeur lorsque la fonction appelée doit être en mesure de modifier un objet qui ne lui appartient pas.

### Constance des paramètres

Il arrive que l'on souhaite permettre à une fonction d'accéder à un objet qui ne lui appartient pas, mais sans pour autant l'autoriser à modifier cet objet. En d'autres termes, on souhaite que la fonction voie l'objet comme étant "read only", c'est à dire constant. Pour obtenir cet effet, il suffit que la fonction dispose d'un paramètre (référence ou pointeur) garantissant la constance de l'objet qu'il désigne :

```
1 int inoffensive (const int & para)  
2 {  
   //para = para + 1; //cette ligne provoquerait une erreur de compilation !  
3   return para + 1;  
4 }
```

```
1 int inoffensiveBis(const int * para)  
2 {  
   //*para = *para + 1; //cette ligne provoquerait une erreur de compilation !  
3   return para + 1;  
4 }
```

Si la fonction ne doit pas modifier l'objet qui ne lui appartient pas, pourquoi lui donner accès à cet objet ? Le comportement souhaité peut être obtenu en écrivant simplement :

```
int inoffensive(int para)  
{  
  return para + 1;  
}
```

Il est vrai que le passage d'une valeur servant à initialiser un **objet local à la fonction** permet à celle-ci de disposer de la valeur sans pour autant être en mesure de modifier un objet qui ne lui appartient pas. Remarquez, toutefois, que le passage d'une valeur exige la création d'un objet local et son initialisation. Certains objets sont si volumineux et complexes que leur création est coûteuse en espace mémoire et leur initialisation coûteuse en temps. On préfère alors utiliser un paramètre de type référence ou pointeur, pour éviter la duplication de l'objet causée par le passage d'une valeur. Il existe en outre des circonstances où l'initialisation d'un objet à l'aide de la valeur d'un autre objet du même type n'est pas possible. Le passage d'une adresse où l'initialisation d'une référence sont alors les seules solutions disponibles.



## 5 - Bon, c'est gentil tout ça, mais ça fait déjà 7 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Les opérateurs arithmétiques possèdent chacun une version permettant d'obtenir une affectation en plus du calcul qu'ils spécifient.
- 2) Les opérateurs ++ et -- modifient d'une unité la valeur de l'objet auquel ils sont appliqués.
- 3) La valeur d'une expression utilisant ++ ou -- est différente selon si l'opérateur est placé avant ou après son opérande.
- 4) Lorsqu'une fonction utilise des paramètres, sa définition en énumère les types et noms entre les parenthèses qui s'interposent entre le nom et le corps de la fonction.
- 5) Les paramètres d'une fonction sont comme des variables locales qui seraient initialisées à l'aide de valeurs transmises par la fonction appelante.
- 6) Lorsqu'une fonction utilise des paramètres, on ne peut normalement l'appeler qu'à condition de fournir, dans l'ordre, les valeurs d'initialisation devant être utilisées pour chacun des paramètres.
- 7) Pour qu'un paramètre possède une valeur par défaut, celle-ci doit être mentionnée dans la déclaration de la fonction<sup>3</sup>.
- 8) Lorsqu'un paramètre possède une valeur par défaut, on peut appeler la fonction en omettant de fournir une valeur pour initialiser ce paramètre.
- 9) Un paramètre ne peut avoir de valeur par défaut qu'à condition que tous les paramètres suivants en aient aussi.
- 10) Lorsqu'on omet de fournir une valeur pour initialiser un paramètre qui a une valeur par défaut, on ne peut plus fournir aucune valeur pour initialiser les paramètres suivants.
- 11) Les paramètres de types pointeurs ou référence permettent à une fonction d'accéder à des objets qui ne lui appartiennent pas.
- 12) Lorsqu'une fonction accède à des objets qui ne lui appartiennent pas, on peut lui interdire de les modifier en faisant des ses paramètres des pointeurs sur (ou des références à) des objets constants.

---

<sup>3</sup> Bien que syntaxiquement admissible, la pratique consistant à fixer les valeurs par défaut des paramètres lors de la *définition* d'une fonction doit être déconseillée aux débutants, car elle est source d'erreurs difficiles à détecter (différentes parties du programme peuvent se retrouver en désaccord quant aux valeurs par défaut d'un même paramètre d'une même fonction).