



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

Apprendre C++ avec Qt : Leçon 3

Fonctions sans paramètres

1 - Utilisation de fonctions	2
Pourquoi les programmes sont-ils constitués de fonctions ?	2
Déclenchement de l'exécution d'une fonction.....	2
Notions d'effet et de résultat	2
Type d'une fonction	3
2 - Définition de fonctions	3
3 - Le corps d'une fonction	4
Définition de variables à l'intérieur d'un bloc.....	4
L'opérateur d'affectation	4
Opérateurs arithmétiques élémentaires	5
Return.....	5
4 - Exemples de fonctions s'appelant les unes les autres	6
5 - Fonctions membre	7
Appeler une fonction membre.....	7
Agir sur une instance désignée explicitement	7
Ne pas désigner explicitement l'instance sur laquelle on agit	8
Définir une classe ayant des fonctions membre	9
Définir une fonction membre d'une classe	9
Fonctions membre constantes	10
6 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ?.....	10

Nous avons vu que le langage C++ organise le code en sections appelées fonctions. De plus, nous venons de voir que la création d'une classe ne prenait tout son sens que dans la mesure où cette classe comporte des fonctions membre, qui permettent d'agir directement sur les objets ayant pour type cette classe, sans avoir à accéder explicitement aux variables membre. Il est donc clair que nous ne pouvons guère aller plus loin dans notre étude du langage sans nous pencher sérieusement sur la notion de fonction, et comme les fonctions décrivent les traitements devant être effectués, il nous faudra au passage introduire quelques-uns des opérateurs permettant cette description.

1 - Utilisation de fonctions

L'utilisation de fonctions n'est pas une option pour nous : c'est ainsi que C++ organise le code, et il va nous falloir définir des fonctions ou abandonner ce langage. Il est cependant intéressant de comprendre pourquoi les concepteurs de beaucoup de langages de programmation ont adopté cette organisation. Il est en outre plus facile de comprendre comment on définit des fonctions si on a déjà une idée de comment un programme peut y faire appel.

Pourquoi les programmes sont-ils constitués de fonctions ?

Il y a deux arguments majeurs en faveur du découpage des programmes en fonctions :

- Dans un programme, il est fréquent que le même traitement doive être appliqué à plusieurs ensembles de données différents. Plus ce traitement est complexe, plus il est avantageux de n'écrire qu'une seule fois le code correspondant et de lui confier successivement les différents ensembles de données. La notion de fonction est une concrétisation directe de cette approche.
- Le découpage du programme en unités pouvant être (dans une certaine mesure) exécutées isolément permet d'envisager la création de chacune d'entre elles comme un projet autonome, que sa taille restreinte rend plus facile à maîtriser. Quand tous les morceaux sont prêts, il n'y a plus qu'à les réunir, et, idéalement, le programme est terminé.

Déclenchement de l'exécution d'une fonction

Chaque fonction porte un **nom**, ce qui permet aux autres fonctions de la désigner. Pour **appeler une fonction** (c'est à dire en déclencher l'exécution), il faut faire suivre son nom d'un couple de parenthèses. Ainsi, s'il existe une fonction nommée `maFonction`, la ligne de code suivante indique qu'il est temps d'exécuter les instructions qui la composent :

```
maFonction(); //exemple d'appel d'une fonction
```

Oui, vous avez raison : on tourne en rond. Si une fonction n'est exécutée que lorsqu'une autre fonction l'appelle, qui commence ? Il existe une fonction nommée `main()`, qui sert de "point d'entrée" au programme et à partir de laquelle toutes les autres sont appelées (directement ou indirectement).

Du point de vue de l'ordre d'exécution des instructions du programme, l'appel d'une fonction ressemble un peu à l'appel d'une note de bas de page dans un texte en français : le discours principal est suspendu¹, le texte de la note est lu, puis la lecture du texte principal reprend (ce qui nécessite de restaurer mentalement le contexte qui permet la compréhension de la phrase interrompue par l'appel de note). A la différence d'une note de bas de page, qui n'est habituellement appelée qu'en un seul point du texte, la même fonction est souvent appelée plusieurs fois¹ dans un même programme.

Notions d'effet et de résultat

L'exécution du code contenu dans la fonction appelée peut avoir pour **effet** de modifier l'état de certains objets qui existaient avant et continuent à exister après cette exécution. Si c'est le cas, cet effet peut, à lui seul, justifier que la fonction existe et soit appelée.

Appeler une fonction en vue d'obtenir son effet, c'est un peu comme mettre en route un appareil qui effectue une certaine tâche et s'arrête de lui-même lorsqu'il a fini.

L'exécution d'une fonction peut aussi conduire à un **résultat**, c'est à dire à la représentation en mémoire d'une **valeur** dont le calcul est la principale raison d'être de la fonction. Dans ce cas, la fonction peut *renvoyer* la valeur en question, et **l'appel de la fonction** s'accompagne

¹ Vous venez d'exécuter l'appel de note de bas de page. Vous allez lire la présente note, puis reprendre la lecture du texte principal à l'endroit où vous l'avez interrompu.

généralement d'indications sur ce qu'il convient de faire avec la valeur qu'elle va renvoyer. On pourra, par exemple, utiliser la valeur renvoyée par la fonction pour **initialiser** une variable :

```
int unEntier = maFonction(); //appel d'une fonction et utilisation du résultat
```

Appeler une fonction en vue d'obtenir son résultat, c'est un peu lui poser implicitement une question. La valeur qu'elle renvoie est tout simplement sa réponse.

Une fonction peut avoir ou non un effet. Elle peut, par ailleurs, produire ou non un résultat. (Mais une fonction sans effet ne produisant pas de résultat serait d'un intérêt assez douteux.)

Lorsqu'une fonction qui renvoie un résultat produit aussi un effet, il peut arriver qu'elle soit exécutée uniquement en vue d'obtenir cet effet. La fonction appelante ignore alors simplement la valeur renvoyée par la fonction appelée :

```
1 double unDouble = fonctionRenvoyantUnDouble(); //appel "complet"  
2 fonctionRenvoyantUnDouble(); //appel ignorant la valeur renvoyée
```

Type d'une fonction

Par convention,

Le type de la valeur renvoyée par une fonction est ce qu'on appelle le type de cette fonction.

Appliquer la notion de type à une fonction est une pure métaphore : il ne s'agit évidemment pas de spécifier le format utilisé pour représenter la fonction en mémoire, et encore moins d'indiquer quelles opérations peuvent être effectuées sur cette fonction.

Lorsqu'une fonction est dépourvue de résultat, elle est de type **void**.

Le type void (vide, en anglais) est un type spécial, qui n'est utilisé que pour les fonctions qui, justement, ne renvoient rien. L'idée même d'une variable de type void serait absurde : la seule raison justifiant la création d'une variable est le besoin d'y stocker une information.

2 - Définition de fonctions

La définition d'une fonction repose sur la même logique que celle d'une variable : il s'agit tout d'abord d'énoncer son **type** et son **nom**. Pour une fonction, toutefois, ces deux informations ne suffisent évidemment pas, et il faut y adjoindre des précisions concernant d'une part les **données** devant être utilisées et, d'autre part, la **procédure** permettant d'obtenir le résultat.

Les données à utiliser sont transmises à la fonction à l'aide du mécanisme de passage de paramètres, qui se traduit, au niveau de la définition d'une fonction, par la présence d'un couple de parenthèses placé à droite du nom de la fonction.

C'est entre ces parenthèses que prendront place les indications nécessaires au passage de paramètres mais nous n'allons nous intéresser, dans un premier temps, qu'à des fonctions ne faisant pas intervenir ce mécanisme.

La procédure à utiliser est, elle, bien évidemment décrite par une suite d'instructions. Ces instructions sont placées entre accolades, ce qui a pour effet de former un bloc d'instructions. Ce bloc est appelé le **corps** de la fonction.

La définition d'une fonction se compose donc au minimum de l'énoncé de son **type** et de son **nom**, suivis d'un couple de parenthèses, puis d'un couple d'accolades. Optionnellement (!), les parenthèses et les accolades sont "garnies", ce qui permet à la fonction de recevoir des **paramètres** et d'effectuer un **traitement**.

Les lignes suivantes définissent de façon parfaitement correcte une fonction C++ (d'intérêt très discutable, avouons-le, puisqu'elle ne fait rien du tout) :

```
1 void maFonction( ) //la fonction C++ minimale  
2 {  
3 } //remarquez l'absence de point-virgule
```

3 - Le corps d'une fonction

Pour aller au-delà de la fonction C++ minimale définie ci-dessus, il nous faut être capables d'insérer un peu de code dans le corps de la fonction.

Définition de variables à l'intérieur d'un bloc

Le code décrivant les traitements que doit effectuer une fonction est, nous l'avons vu, placé entre accolades, ce qui en fait un bloc. Ce bloc peut comporter des définitions de variables.

Une variable dont la définition se situe dans un bloc de code est **locale** à ce bloc, c'est à dire qu'elle n'est considérée comme définie qu'à l'intérieur du bloc.

En d'autres termes, si des variables sont définies dans le corps d'une fonction, il s'agit de variables appartenant en propre à cette fonction, et aucune autre fonction ne peut y accéder. Les variables locales à une fonction ont vocation à être utilisées, lors des traitements effectués par la fonction, pour stocker des résultats intermédiaires. Ces variables ne sont, en général, que des variables temporaires, qui cessent d'exister² lorsque l'exécution de la fonction s'achève.

Les variables définies dans une fonction (ou dans un bloc) sont donc recréées et réinitialisées chaque fois que la fonction (ou le bloc) en question est exécutée.

L'opérateur d'affectation

Nous savons déjà, grâce à l'**initialisation**, attribuer une valeur à une variable au moment où elle est définie. Il est évidemment possible d'attribuer une valeur à une variable *après* qu'elle a été définie. Il suffit pour cela d'utiliser l'opérateur d'**affectation**, représenté par le signe =.

```
1 int unEntier = 3;    //définition et initialisation
2 unEntier = 3;       //affectation
```

Le fait que l'**affectation** est représentée par un symbole qui peut aussi être utilisé pour l'**initialisation** ne doit pas vous laisser croire qu'il s'agit d'une seule et même opération. Une constante, par exemple, DOIT être initialisée mais NE PEUT PAS subir d'affectation.

```
const double PI = 3.14;    //OK : initialisation
PI = 3.1416;              //ERREUR : tentative de modification d'une constante !
```

Lorsqu'une expression impliquant une affectation est évaluée, les expressions situées de part et d'autre du signe égal sont d'abord évaluées indépendamment l'une de l'autre. Le résultat de gauche doit désigner une zone de mémoire possédant un type. Le résultat de droite est alors représenté conformément aux règles de ce type, puis rangé dans la zone mémoire désignée.

Lorsque l'expression de gauche est le nom d'une variable effectivement définie, la seule question qui se pose vraiment est "Est-ce que le résultat obtenu à droite peut être représenté correctement dans le type de la variable ?". Si ce n'est pas le cas, votre compilateur émettra un message d'avertissement ou d'erreur signalant une incompatibilité de types.

Si l'expression de gauche est plus complexe, un autre problème se pose tout d'abord : l'évaluation de cette expression conduit-elle à quelque chose ayant un type et une localisation en mémoire déterminés ? Si ce n'est pas le cas, votre compilateur protestera en réclamant une "lvalue", c'est à dire quelque chose pouvant légitimement apparaître à gauche de l'opérateur d'affectation (gauche se traduit par **left** en anglais, d'où le **l** de **lvalue**).

L'évaluation d'une expression comportant un opérateur d'affectation a donc un *effet*, le changement de valeur de la variable concernée, qui est en général la seule raison d'être de l'expression. Il arrive parfois qu'on utilise aussi la **valeur d'une affectation**, qui est identique à celle reçue par la variable affectée. Ainsi, dans

```
1 int x;
2 int y = x = 4;
```

la ligne 2 est analysée comme ordonnant d'**initialiser** y avec la **valeur** de l'**expression d'affectation**, c'est à dire 4 (en vertu de la règle que nous venons d'énoncer). L'évaluation de l'expression **x = 4** a, bien entendu, pour *effet* d'attribuer aussi cette valeur à la variable x.

² Lorsqu'une variable cesse d'exister, la zone de mémoire que son nom désignait est considérée comme disponible pour d'autres usages.

Opérateurs arithmétiques élémentaires

Les opérateurs d'addition, soustraction, multiplication et division sont respectivement notés `+`, `-`, `*` et `/`. L'évaluation de l'expression `3 + 4` donne donc, par exemple, le résultat 7.

Lorsqu'un **nom de variable** intervient dans une expression arithmétique, c'est la valeur actuelle de la variable qui est utilisée pour évaluer l'expression.

Lorsqu'un **appel de fonction** intervient dans une expression arithmétique, c'est la valeur renvoyée par la fonction qui est utilisée pour évaluer l'expression.

Lorsqu'il est appliqué à des opérandes de type entier, l'opérateur `/` produit le résultat de la division entière. Le reste d'une telle division peut être obtenu à l'aide de l'opérateur `%`, parfois appelé **modulo** :

```
5 / 2; //cette expression a pour valeur 2
5 % 2; //cette expression a pour valeur 1
```

L'évaluation d'une expression arithmétique conduit à un résultat mais reste, sauf cas particuliers, sans effet. Ainsi, aucune variable ne se trouvant modifiée, les résultats obtenus lors d'une éventuelle exécution des deux lignes de code ci-dessus cesseront d'être disponible dès l'évaluation de l'expression suivante, et auront donc été calculés en pure perte. Dans la plupart des cas, vous utiliserez donc les expressions arithmétiques en tant que membre de droite d'une **affectation**.

```
1 double coteDuCarre = 3.75;
2 double perimetre = 0;
3 double surface = 0;
4 //exemples de calculs élémentaires
5 perimetre = 4 * coteDuCarre;
6 surface = coteDuCarre * coteDuCarre;
```

Lorsque la même ligne de code exige un **calcul** et l'**affectation** du résultat à une variable, le calcul est effectué indépendamment de la nature de la variable dans laquelle le résultat va ensuite être transféré. Ceci signifie, en particulier, que le type utilisé pour représenter le résultat du calcul ne dépend pas de celui de la variable qui va être affectée, mais seulement de celui des opérandes utilisés. Ainsi, lors de

```
double x = 5 / 2; //surprise ! x contient maintenant 2
```

le résultat de la division (entière) est représenté sous forme décimale pour être stocké dans `x`, mais le type de cette variable ne conduit pas à refaire le calcul avec une division décimale. Si c'est cette opération qui est nécessaire, il faut qu'au moins l'un des opérandes soit **décimal** :

```
double y = 5 / 2.0; //y contient maintenant 2.5
```

Return

Lorsqu'une fonction produit un résultat, elle s'achève en "renvoyant" le résultat en question à l'aide d'une instruction spéciale, `return`.

```
1 int maFonction()//une fonction C++ produisant un résultat de type int
2 {
3 return 12; //12 est une valeur convenable pour un int
4 }
```

La fonction définie ci-dessus est d'un intérêt pratique absolument nul, puisque son exécution ne produira qu'une valeur immuable et parfaitement prévisible. Dans ces conditions, écrire

```
int unEntier = maFonction();
```

n'est qu'une façon perverse d'obtenir ce qui pourrait être directement obtenu par

```
int unEntier = 12;
```

Dans un programme réel, une fonction effectue généralement des traitements qui font de son appel une opération intéressante.

Lorsqu'une fonction **ne produit aucun résultat**, elle s'achève soit par la fin du bloc d'instructions, soit par une instruction `return` dépourvue de valeur. La définition suivante est équivalente à celle proposée plus haut pour une fonction C++ minimale :

```
1 void maFonction() //void, puisqu'elle ne produit aucun résultat
2 {
3 return; //cette instruction return est facultative car, de toutes façons,
4 } //cette accolade marque la fin de la fonction
```

4 - Exemples de fonctions s'appelant les unes les autres

Les moyens de communication entre fonctions dont nous disposons pour l'instant sont extrêmement réduits, puisqu'ils se résument à la possibilité qu'a une fonction appelée de renvoyer un résultat à la fonction appelante. Les exemples suivants ont précisément pour objectif d'illustrer ces limitations en essayant (vainement) de les contourner.

Imaginons que nous avons à manipuler des rectangles, et que leur surface nous préoccupe particulièrement. Il est facile d'écrire une fonction calculant la surface d'un rectangle³

```
1 double calculeSurface()
2 {
3 double largeur = 4;
4 double longueur = 3;
5 return largeur * longueur;
6 }
```

On peut, certes, appeler cette fonction à partir d'une autre :

```
7 void maFonction()
8 {
9 double surfaceDuRectangle = calculeSurface();
10 }
```

Mais, dans l'état actuel des choses, cet appel n'est qu'une façon compliquée d'obtenir un état qui pourrait plus simplement être atteint par

```
double surfaceDuRectangle = 12;
```

En effet, nous n'avons aucun moyen de préciser à la fonction `calculeSurface()` les dimensions du rectangle qui intéresse `maFonction()`. Il serait vain d'essayer de résoudre ce problème en définissant les variables `largeur` et `longueur` dans `maFonction()` au lieu de les définir dans `calculeSurface()` :

```
1 void maFonction()
2 {
3 double largeur = 4;
4 double longueur = 3;
5 double surfaceDuRectangle = calculeSurface(); //appel de l'autre fonction
6 }
7
8 double calculeSurface()
9 {
10 return largeur * longueur; //IMPOSSIBLE : largeur et longueur n'existent pas ici !
}
```

Les variables `largeur` et `longueur` sont maintenant locales à `maFonction()`. La fonction `calculeSurface()` ne peut donc pas les utiliser (ligne 9) pour faire son calcul.

Il n'est pas non plus possible de faire communiquer les deux fonctions en les dotant de variables portant le même nom :

```
1 double calculeSurface()
2 {
3 double largeur;
4 double longueur;
5 return largeur * longueur;
6 }
```

³ Peut-être est-ce même un peu trop facile, au point que vous ayez du mal à comprendre pourquoi ce serait une bonne idée d'écrire une telle fonction plutôt que de faire directement la multiplication. Si c'est le cas, faites un petit effort d'imagination : remplacez mentalement le calcul de la surface par trois pages de calculs complexes.

```
7 void maFonction()  
8 {  
9     double largeur = 4;  
10    double longueur = 3;  
11    double surfaceDuRectangle = calculeSurface(); //appel de l'autre fonction  
12 }
```

Les deux fonctions possèdent maintenant chacune un jeu de variables nommées largeur et longueur. Ces rapports d'homonymie ne créent aucun lien particulier entre les variables, et les valeurs stockées dans les variables locales à `maFonction()` ne sont évidemment pas disponibles (par magie ?) dans les variables locales à `calculeSurface()`. Cette dernière procède donc (ligne 5) à la multiplication de deux valeurs dont nous ignorons tout (puisque les variables utilisées n'ont pas été initialisées). Le résultat qu'elle renvoie (ligne 6) et que recueille (ligne 13) `maFonction()` n'est donc sans doute pas la surface recherchée...

Nous devons donc conclure que la localité des variables d'une fonction s'oppose à leur utilisation pour établir un transfert d'information entre fonctions. Le passage de paramètres permet une communication entre fonction appelante et fonction appelée, mais, avant d'étudier cette technique (Leçon 5), nous allons voir que, lorsque la fonction appelée est membre d'une classe, elle dispose d'un autre moyen de communication : elle peut accéder à certaines variables qui ne lui appartiennent pas.

5 - Fonctions membre

Le cas des fonctions membre présente quelques particularités syntaxiques, et il faut donc que nous apprenions d'une part à définir des classes comportant des fonctions membres, et d'autre part à définir des fonctions membres d'une classe.

Au-delà de ces détails syntaxiques, la différence essentielle entre une fonction membre et une fonction qui n'appartient à aucune classe réside dans la façon dont les fonctions membre sont appelées. C'est donc par ce point que nous allons débiter notre étude des fonctions membre.

Appeler une fonction membre

La nécessité des fonctions membre s'est imposée à nous, à la fin de la Leçon 2, parce que *"Pour que des ordres puissent être appliqués directement à des variables dont le type est une classe, il faut que la définition de cette classe indique comment ces ordres doivent être exécutés."*

La raison d'être d'une fonction membre est donc d'agir sur une instance. Mais laquelle ?

La désignation de l'objet "cible" de l'exécution d'une fonction membre peut, selon les circonstances, être explicite ou rester implicite. Le premier cas correspond à la situation où une fonction a en sa possession une instance et souhaite lui appliquer le traitement effectué par une fonction membre. Le second correspond à la situation où une fonction membre a été appelée pour agir sur une instance et souhaite elle-même faire appel à une de ses collègues pour agir sur la même instance.

Agir sur une instance désignée explicitement

Imaginons qu'il existe une classe nommée `CRectangle` et que cette classe comporte non seulement deux variables membres nommées longueur et largeur, mais aussi une fonction membre nommée `calculeSurface()` renvoyant le produit des deux variables membre.

Imaginons également qu'une fonction quelconque dispose de deux variables locales de type `CRectangle`, variables auxquelles auraient été confiées des dimensions :

```
1 CRectangle salon;  
2 salon.largeur = 4;  
3 salon.longueur = 5.6;  
4 CRectangle cuisine;  
5 cuisine.largeur = 3;  
6 cuisine.longueur = 3;
```

Pour donner une largeur (2 et 5) ou une longueur (3 et 6) à un `CRectangle`, il est évidemment nécessaire de préciser sur quel objet on agit. Cette indication est donnée en sélectionnant le membre concerné de l'une des instances disponibles.

Pour calculer une surface en utilisant la fonction membre prévue à cet effet, il est également nécessaire d'indiquer sur quel `CRectangle` doit porter le calcul. Cette indication est donnée en appelant la **fonction membre au titre** de l'instance concernée :

```
7 double surfaceTotale = salon.calculeSurface() + cuisine.calculeSurface();
```

Les opérateurs de sélection permettent d'appeler une fonction membre d'une classe *au titre* d'une instance de la classe en question, ce qui désigne cette instance comme étant l'objet sur lequel doit agir la fonction au cours de l'exécution déclenchée par cet appel.

Lors des deux exécutions successives déclenchées par l'exécution de la ligne 7, la fonction `calculeSurface()` ne travaillera donc pas sur le même rectangle. Remarquez que, à la différence de la situation explorée pages 6, la fonction appelante peut ici fixer elle-même les dimensions du rectangle sur lequel `calculeSurface()` va opérer. Les variables membre permettent donc un transfert d'information de l'appelant vers la fonction membre appelée⁴.

Ne pas désigner explicitement l'instance sur laquelle on agit

Les instructions placées dans le corps de la fonction `calculeSurface()` vont donc devoir accéder tantôt aux membres de la variable `salon`, tantôt aux membres de la variable `cuisine`, sans parler de tous les autres `CRectangle` au titre desquels la fonction est susceptible d'être appelée. Il est donc totalement impossible que ces instructions mentionnent le nom de l'instance concernée (qui n'est pas toujours le même, et auquel elles n'ont de toute façon pas accès puisqu'il s'agit d'une variable appartenant à la fonction appelante).

Pour contourner cette difficulté, les fonctions membre reçoivent, lors de chaque exécution, l'adresse de l'instance au titre de laquelle elles sont appelées, et **cette adresse** est **implicitement utilisée** lorsque le code présent dans le corps de la fonction mentionne le **nom d'un membre** sans préciser quelle est l'instance concernée. Ainsi, si l'instruction

```
return longueur * largeur;
```

figure dans la fonction `calculeSurface()`, elle donnera lieu au calcul de

```
(&salon)->longueur * (&salon)->largeur
```

lorsque la fonction est appelée au titre de l'instance `salon`, mais au calcul de

```
(&cuisine)->longueur * (&cuisine)->largeur
```

lorsque la fonction est appelée au titre de l'instance `cuisine`.

Lorsqu'une fonction membre mentionne le nom d'un autre membre de la même classe sans dire de quelle instance elle parle, elle accède implicitement à l'instance au titre de laquelle elle est en train d'être exécutée.

Cette règle s'applique indifféremment, qu'il s'agisse d'accéder à une variable membre ou d'appeler une autre fonction membre. En revanche, de par sa nature même, cette règle ne concerne que le code figurant dans une fonction membre. Les fonctions qui ne sont pas membre d'une classe ne peuvent accéder à un membre de celle-ci qu'en spécifiant explicitement l'instance visée.

Bien que la transmission de l'adresse de l'instance au titre de laquelle la fonction membre est appelée reste toujours invisible aux yeux des programmeurs, il est possible (et parfois indispensable) d'utiliser explicitement cette adresse dans le corps d'une fonction membre : elle y est disponible sous le nom de `this`.

En d'autres termes, notre hypothétique instruction

```
return longueur * largeur;
```

pourrait figurer dans la fonction `calculeSurface()` sous la forme

```
return this->longueur * this->largeur; //correct, mais inutilement complexe
```

La Leçon 9 nous donnera l'occasion de revenir plus longuement sur ce mécanisme.

⁴ Comme les fonctions membre peuvent accéder aux variables membre de l'instance au titre de laquelle elles sont appelées, il est aussi possible d'utiliser ces variables pour effectuer un renvoi d'information vers la fonction appelante.

Définir une classe ayant des fonctions membre

Pour inclure une fonction membre dans la définition d'une classe, il suffit d'y indiquer son **type**, son **nom** et la liste de ses **paramètres**. Etant donné que, pour l'instant, nous n'utilisons pas le passage de paramètres, les parenthèses entourant cette liste restent vides.

N'oubliez pas ces parenthèses, car, comme le montre bien l'exemple ci-dessous, elles seules permettent de faire la différence entre une *variable* membre et une *fonction* membre.

```
1 class CRectangle //une classe comportant une fonction membre
2 {
3 public:
4     double largeur;           //déclaration d'une variable membre
5     double longueur;         //déclaration d'une autre variable membre
6     double calculeSurface();  //déclaration d'une fonction membre
7 };
```

Comme nous l'avons vu au cours du TD 2, les classes sont habituellement définies dans des fichiers portant l'extension .h. Cette définition est essentiellement la déclaration de l'ensemble des membres constituant la classe et, par conséquent,

Un fichier .h contient des définitions de types et des déclarations de fonctions et de variables.

Les fonctions qui ne sont membres d'aucune classe sont, elles-aussi, généralement déclarées dans des fichiers .h : la #inclusion de ces fichiers permet de disposer des déclarations nécessaires pour appeler ces fonctions à partir d'autres fonctions définies dans différents fichiers .cpp.

Définir une fonction membre d'une classe

La définition d'une fonction membre ne diffère en rien de la définition d'une fonction qui n'est pas membre d'une classe. La seule chose à laquelle il faut prendre garde est que **le nom complet de la fonction inclut celui de la classe**, et que c'est ce nom complet (on dit aussi parfois "nom qualifié") qui doit être utilisé pour définir la fonction. Le nom complet est obtenu simplement en liant le nom de la classe et celui de la fonction à l'aide de l'opérateur **::**, opérateur sur lequel nous aurons l'occasion de revenir plus tard.

```
1 double CRectangle::calculeSurface() // définition d'une fonction membre
2 {
3     return largeur * longueur;
4 }
```

L'intérêt des noms complets est assez facile à comprendre : si notre programme utilise également une classe CCercle qui comporte, elle aussi, une fonction de calcul de surface, il sera possible de définir une fonction cercle::calculeSurface(), sans pour autant créer la moindre ambiguïté. Cette possibilité d'homonymie partielle présente deux avantages : lorsque les classes sont conçues de façon coordonnée, il est possible de décharger les programmeurs qui les utiliseront de l'obligation de mémoriser des noms de fonctions différents pour des opérations de même nature et, lorsque les classes sont conçues de façon indépendante, elles ne courent pas le risque d'interférer les unes avec les autres à cause de coïncidences malencontreuses dans le choix des noms.

La définition d'une fonction membre prend normalement place dans un fichier .cpp portant le nom de la classe.

Un fichier .cpp contient des définition de fonctions, c'est à dire du texte que le compilateur va traduire en code exécutable.

Lorsque le corps de la fonction est très bref, il est possible de l'inclure directement *dans* la définition de la classe. Dans notre exemple, choisir cette option conduirait à écrire :

```
1 class CRectangle
2 {
3 public:
4     double largeur;           //déclaration d'une variable membre
5     double longueur;         //déclaration d'une autre variable membre
6     double calculeSurface() { return largeur * longueur; } //définition !
7 };
```

Remarquez que cette façon de procéder conduit, à titre dérogatoire, à faire figurer la définition d'une fonction dans un fichier .h. Si l'aspect général de la définition de la classe est peu modifié par la présence du **corps de la fonction**, on adopte généralement pour celle-ci une présentation en une seule ligne, qui lui confère une forme qu'il faut apprendre à reconnaître. Remarquez, en particulier, que l'accolade de fermeture du bloc (6) n'est suivie d'aucun point virgule, à la différence de celle qui conclut (7) la définition de la classe.

Fonctions membre constantes

Certaines fonctions membre ne modifient en aucune façon l'état de l'instance au titre de laquelle elles sont exécutées. Il est possible d'annoncer officiellement cette caractéristique de la fonction en ajoutant le mot `const` après la liste de ses paramètres :

```
double calculeSurface() const { return largeur * longueur; }
```

Rendre une fonction membre constante présente deux avantages principaux :

- toute instruction qui pourrait, par inadvertance, conduire à une modification de l'objet au titre duquel la fonction est exécutée provoquera une erreur de compilation ;
- il devient possible d'exécuter la fonction au titre d'une constante.

6 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Chaque fonction porte un nom, qui permet de l'appeler.
- 2) Le passage de paramètres permet d'indiquer à une fonction sur quelles données elle doit opérer, mais on verra ça plus tard.
- 3) Un fragment de texte source placé entre accolades constitue un bloc d'instructions.
- 4) Les opérations qu'une fonction doit effectuer sont décrites par un bloc d'instructions appelé le corps de la fonction.
- 5) Lorsqu'une fonction est appelée, les instructions qui la définissent sont exécutées avant que l'instruction qui suit l'appel ne le soit.
- 6) L'exécution d'une fonction produit soit un effet, soit un résultat, soit les deux.
- 7) Le "type" d'une fonction est le type du résultat qu'elle produit.
- 8) Si une fonction ne produit aucun résultat, elle est de type `void`.
- 9) Une fonction renvoie son résultat à l'aide de l'instruction `return`.
- 10) L'affectation d'une valeur à une variable est obtenue à l'aide de l'opérateur `=`.
- 11) Les opérations arithmétiques usuelles sont notées comme d'habitude.
- 12) La division de deux valeurs de type entier donne toujours un résultat entier.
- 13) Lorsque le nom d'une variable ou l'appel d'une fonction figure dans une expression arithmétique, celle-ci est évaluée en utilisant la valeur courante de la variable ou le résultat renvoyé à l'issue de l'exécution de la fonction.
- 14) Les variables définies dans un bloc de code sont locales à ce bloc (c'est à dire qu'elles ne sont utilisables que par des instructions figurant dans le bloc en question).
- 15) Les fonctions membre d'une classe sont appelées au titre d'une instance de cette classe.
- 16) Les fonctions membre d'une classe accèdent implicitement (c'est à dire sans avoir besoin de préciser de nom d'instance) aux membres de l'instance au titre de laquelle elles sont en cours d'exécution.
- 17) Les fonctions membre constantes ne peuvent pas modifier l'instance au titre de laquelle elles sont exécutées.
- 18) Normalement, les fichiers **.cpp** contiennent du texte que le compilateur va **traduire en code exécutable**, alors que les **.h** ne contiennent que des consignes indiquant au compilateur **comment faire cette traduction**.