Centre Informatique pour les Lettres et les Sciences Humaines

C++ : Leçon 3
Fonctions

Nous avons vu que le langage C++ organise le code en sections appelées fonctions. De plus, nous venons de voir que la création de nouveaux types de données par le système des classes ne prenait tout son sens que dans la mesure où ces classes comportent des fonctions membre, qui permettent d'agir directement sur les objets, 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.

1 - Utilisation de fonctions

D'une certaine façon, 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 certains des avantages qui ont conduit les concepteurs de beaucoup de langages de programmation à adopter 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 l'appeler (c'est à dire d'en déclencher l'exécution). Ainsi, s'il existe une fonction nommée `maFonction()`, la ligne de code suivante indique qu'il est temps d'exécuter le code qu'elle contient :

```
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). Ce mécanisme ne nous concerne pas dans l'immédiat, car nous n'envisageons pas, pour l'instant, d'écrire des programmes complets, mais simplement d'ajouter (ou de modifier) des fonctions à l'intérieur d'un "squelette" de programme généré par notre système de développement (ce que l'on appelle parfois un "framework").

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 suspendu[1], 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 fois[1] 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 certaines variables 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.

[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.

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 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 aussi produire ou non un résultat. (Remarquez quand même qu'une fonction sans effet et ne produisant pas de résultat serait d'un intérêt assez douteux.)

Lorsqu'une fonction qui produit un résultat a 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. Certaines des instructions placées dans ce bloc peuvent définir des 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'exister[2] lorsque l'exécution de la fonction s'achève.

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 =.

> 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. Nous rencontrerons bientôt des situations où les règles applicables pour l'initialisation et pour l'affectation ne sont pas les mêmes.

```
1  int unEntier;       //définition
2  unEntier = 3;       //affectation
3  int unAutre = 3 ;   //définition et initialisation
4  unAutre = 4;        //affectation
```

Lorsqu'une expression comportant l'opérateur d'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 obtenu à gauche doit désigner une zone de mémoire possédant un type. Le résultat obtenu à droite est alors représenté conformément aux règles du type en question, 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 telle expression, qui est simplement définie comme étant 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 `x = 4`, 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.

Les quatre opérateurs arithmétiques élémentaires

Les opérateurs d'addition, soustraction, multiplication et division sont respectivement notées +, -, * 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.

L'évaluation d'une expression arithmétique conduit à un résultat mais reste, sauf cas particuliers, sans effet. En d'autres termes, aucune variable ne se trouvant modifiée, le résultat obtenu cessera d'être disponible dès l'évaluation de l'expression suivante et n'aura, en définitive, été d'aucune utilité. 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;
   //exemples de calculs élémentaires
4  perimetre = 4 * coteDuCarre;
5  surface = coteDuCarre * coteDuCarre;
```

Lorsque la même ligne de code exige un calcul et l'affectation du résultat à une variable, il faut bien comprendre que 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 ne dépend pas de celui de la variable qui va être affectée, mais seulement de celui des opérandes utilisés. Ainsi, après

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

la variable x contient 2, et non 2.5. En effet, les opérandes 5 et 2 sont de type `int` (cf. Leçon 2), et l'opérateur est donc interprété comme désignant la *division entière*. Le fait que, par la suite, le résultat est représenté en format décimal pour être transféré dans une variable de type `double` ne conduit évidemment pas à réévaluer l'expression en faisant une autre interprétation ! Si c'est une division décimale qui doit être effectuée, il faut veiller à ce 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  }
```

Il va sans dire que 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 un résultat qui pourrait être obtenu directement 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 = 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 simples

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 rectangle[3]

```
1  double calculeSurface()
2  {
3  double largeur = 4;
4  double longueur = 3;
5  double surface = largeur * longueur;
6  return surface;
7  }
```

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

```
1  void maFonction()
2  {
3  double surfaceDuRectangle = calculeSurface();
4  }
```

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 ne disposons d'aucun moyen de préciser à la fonction `calculeSurface()` les dimensions du rectangle qui intéresse `maFonction()`. Il est particulièrement important de bien comprendre qu'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  double calculeSurface()
2  {
3  double surface = largeur * longueur;      //IMPOSSIBLE : largeur et longueur
4  return surface;                           //n'existent pas ici !
5  }
```

```
6  void maFonction()
7  {
8  double largeur = 4;
9  double longueur = 3;
10 double surfaceDuRectangle = 0;
11 surfaceDuRectangle = calculeSurface(); //appel de l'autre fonction
12 }
```

> Les variables `largeur` et `longueur` sont maintenant locales à `maFonction()`. La fonction `calculeSurface()` ne peut donc pas les utiliser (ligne 3) 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  double surface = largeur * longueur;
6  return surface;
7  }
```

[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

```
8  void maFonction()
9  {
10 double largeur = 4;
11 double longueur = 3;
12 double surfaceDuRectangle = 0;
13 surfaceDuRectangle = calculeSurface();    //appel de l'autre fonction
14 }
```

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()` a donc vraiment très peu de chances de correspondre à la surface recherchée.

Le passage de paramètres permet une communication entre fonction appelante et fonction appelée. Avant d'en venir à cette technique, qui fera l'objet de la Leçon 5, nous allons voir que, lorsque les fonctions concernées sont membres d'une classe, le langage C++ offre d'autres possibilités de communication.

5 - Fonctions membre

Nous venons de constater que le fait qu'une fonction n'a accès qu'à ses propres variables locales[4] s'oppose à l'utilisation des variables pour établir un transfert d'information entre fonctions. Cette limitation d'accès présente toutefois une exception : lorsqu'une classe comporte des fonctions membre, celles-ci ont non seulement accès à leurs variables locales, mais aussi aux variables membre de la classe. Avant de voir comment cette particularité peut être utilisée pour faire communiquer deux fonctions, il nous faut toutefois apprendre d'une part à définir une classe comportant des fonctions membre et, d'autre part, à définir une fonction qui est membre d'une classe.

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.

```
   //exemple de définition d'une classe comportant une fonction membre
1  class rectangle
2  {
3  public:
4     double largeur;              //une variable membre,
5     double longueur;             //une autre variable membre et
6     double calculeSurface( );    //une fonction membre !
7  };
```

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.

[4] En toute honnêteté, il faudrait peut-être mentionner la possibilité de créer des variables globales. C'est maintenant chose faite, et je n'ajouterai que ceci : si vous utilisez des variables globales dans un programme, ne comptez pas sur moi pour essayer de comprendre pourquoi il ne fonctionne pas.

```
//exemple de définition d'une fonction membre
double rectangle::calculeSurface()
{
double surface = 0;
surface = largeur * longueur; //on peut accéder aux variables membre
return surface;
}
```

L'intérêt d'utiliser son nom complet lorsqu'on définit une fonction membre est assez facile à comprendre : imaginez que notre programme utilise également une classe `cercle`, et que celle-ci comporte, elle aussi, une fonction permettant de calculer la surface. L'usage du nom complet permet à la classe `cercle` d'utiliser le même nom de fonction, `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.

Lorsque le code qui doit prendre place entre les accolades délimitant le corps de la fonction est très bref, il arrive qu'on préfère définir la fonction membre directement *dans* la définition de la classe. Dans notre exemple, choisir cette option pourrait conduire à écrire :

```
//exemple de définition d'une classe comportant la DEFINITION d'une fonction
class rectangle
{
public:
   double largeur;            //une variable membre,
   double longueur;           //une autre variable membre et
   double calculeSurface() { return largeur * longueur; }
};
```

Si l'aspect général de la définition de la classe est peu modifié par la présence de la définition de la fonction, on adopte en revanche généralement une présentation en une seule ligne de la définition de la fonction, qui prend alors une forme qu'il faut apprendre à reconnaître. Remarquez, en particulier, que l'accolade de fermeture du bloc n'est suivie d'aucun point virgule, à la différence de celle qui conclut la définition de la classe.

Communiquer avec une fonction membre depuis l'extérieur de la classe

Lorsque la fonction appelante n'est pas membre de la classe dont fait partie la fonction appelée, l'appel n'est possible que si la fonction appelante dispose d'une variable qui est une instance de cette classe. Contrairement à ce que la phrase précédente pourrait laisser croire, il s'agit là d'une règle très simple[5] :

```
// On suppose que la classe rectangle et la fonction
// rectangle::calculeSurface() ont été définies comme précédemment
void maFonction()
{
//définitions de variables locales
rectangle monJoliRectangle;//une instance de la classe rectangle
double surface = 0;
//on peut fixer nous-mêmes la taille du rectangle
monJoliRectangle.longueur = 18.75;
monJoliRectangle.largeur = 3.2;
//la fonction calculeSurface peut nous renvoyer la surface de CE rectangle
surface = monJoliRectangle.calculeSurface();
}
```

On accède aux fonctions membre (pour les appeler) exactement comme on accède aux variables membre (pour leur affecter une valeur) : on utilise un nom composé obtenu en connectant celui d'une instance de la classe à celui du membre à désigner à l'aide d'un opérateur de sélection, le point. On dit alors que la fonction est appelée *au titre* de l'instance.

[5] Un des aspects du langage C++ qui me plaisent le moins est que les phénomènes les plus anodins sont capables de donner naissance à des phrases effrayantes lorsqu'on cherche à les décrire en français.

Dans le cas de l'accès à une variable membre, la nécessité de disposer d'une instance s'impose assez naturellement : de quelle `longueur` et de quelle `largeur` pourrions-nous parler en l'absence de rectangle ? Et, s'il existe plusieurs variables de type `rectangle`, il faut bien préciser de laquelle nous parlons lorsque nous souhaitons fixer une des dimensions. Il est donc évident qu'on ne peut accéder aux variables membre qu'en précisant le nom de l'instance concernée.

Le cas des fonctions membre nécessite peut-être un peu plus de réflexion, car il faut prendre en compte le fait que la raison d'être habituelle d'une fonction membre est d'*opérer des traitements impliquant les variables membre*. Nous sommes donc ramenés à la question précédente : de quel rectangle devons nous calculer la surface, s'il n'en existe aucun (ou s'il en existe plusieurs) ? Une fonction membre (telle que `calculeSurface()`, par exemple) ne peut être appelée qu'en précisant le nom de l'instance concernée.

Contrairement à la situation explorée pages 6 et 7, `maFonction()` est ici en mesure de faire effectuer par la fonction `rectangle::calculeSurface()` un calcul portant sur un rectangle dont c'est `maFonction()` qui fixe les dimensions. Les variables membre ont donc bien permis un transfert d'information de la fonction appelante vers la fonction appelée.

Le transfert d'information de la fonction appelée vers la fonction appelante emprunte ici le chemin du renvoi du résultat par la fonction `rectangle::calculeSurface()`. Il aurait aussi été possible d'ajouter une variable membre (nommée `surface`, par exemple) à la classe rectangle. La fonction `rectangle::calculeSurface()` aurait alors pu stocker le résultat de son calcul dans cette variable, et `maFonction()` aurait eu accès à cette valeur qu'elle aurait connue sous le nom de `monJoliRectangle.surface`.

Communiquer entre fonctions membre de la même classe

Ce cas de figure suppose, évidemment, que la classe soit dotée d'au moins deux fonctions. Munissons-nous, par exemple, d'un calcul de prix, celui-ci étant défini comme le produit de la surface par un prix au mètre carré (largeur et longueur étant exprimés en mètres).

```
//exemple de définition d'une classe comportant deux fonctions membre
class rectangle
{
public:
   double largeur;              //exprimée en mètres
   double longueur;             //exprimée en mètres
   double prixAuM2;             //en euros par mètre carré
   double calculeSurface();     //renvoie une mesure en mètres carrés
   double calculePrix();        //renvoie un prix en euros
};
```

La fonction `calculePrix()` peut alors être définie ainsi :

```
double rectangle::calculePrix()
{
//définition des variables locales
double surface = 0;
double prix = 0;
//obtention de la surface DU RECTANGLE DONT ON CHERCHE LE PRIX
surface = calculeSurface();
//calcul du prix
prix = surface * prixAuM2;
return prix;
}
```

Cette fonction ne comporte qu'une particularité méritant d'être soulignée : elle appelle *(5)* la fonction membre `calculeSurface()` et elle accède *(6)* à la variable `prixAuM2` *sans utiliser de nom composé* faisant intervenir le nom d'une instance.

Lorsqu'une fonction membre appelle une autre fonction membre (de la même classe) ou accède à une variable membre sans spécifier l'instance concernée, c'est automatiquement l'instance sur laquelle opérait la fonction appelante qui est utilisée pour cet appel.

Dans notre exemple, cet automatisme assure simplement que la surface calculée est bien celle du rectangle dont on cherche à déterminer le prix.

La règle énoncée ci-dessus relève du simple bon sens : si une fonction membre est invoquée au titre d'une instance, les traitements qu'elle effectue (modification de variables membre ou appel d'autres fonctions membre) doivent a priori s'appliquer à cette instance, et non à une autre !

La fonction `calculePrix()` ne pose aucune difficulté particulière d'utilisation :

```
1  void maFonction()
2  {
   //définitions de variables locales
3  rectangle unRectangle;
4  rectangle unAutre
5  float prixDUnRectangle;
6  float prixDUnAutre;

   //on fixe la taille des rectangles et leur prix au mètre carré
7  unRectangle.longueur = 18.75;
8  unRectangle.largeur = 3;
9  unRectangle.prixAuM2 = 36.50;
10 unAutre.longueur = 125;
11 unAutre.largeur = 348;
12 unAutre.prixAuM2 = 142.36;

   //la fonction calculePrix peut opérer sur n'importe quel rectangle
13 prixDUnRectangle = unRectangle.calculePrix();
14 prixDUnAutre = unAutre.calculePrix();
15 }
```

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) Les opérations qu'une fonction doit effectuer sont décrites par un bloc d'instructions (c'est à dire un fragment de texte source placé entre accolades).
4) Lorsqu'une fonction est appelée, les instructions qui la définissent sont exécutées avant que l'instruction qui suit l'appel ne le soit.
5) L'exécution d'une fonction produit soit un effet, soit un résultat, soit les deux.
6) Le type d'une fonction est le type du résultat qu'elle produit.
7) Si une fonction ne produit aucun résultat, elle est de type `void`.
8) Une fonction renvoie son résultat à l'aide de l'instruction `return`.
9) L'affectation d'une valeur à une variable est obtenue à l'aide de l'opérateur =. C'est une opération moins simple qu'elle en a l'air.
10) Les opérations arithmétiques usuelles sont notées comme d'habitude.
11) Lorsqu'un nom de variable ou de fonction est utilisé dans une expression arithmétique, celle-ci est évaluée en utilisant la valeur de la variable ou le résultat de la fonction.
12) 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).
13) Les fonctions membre d'une classe sont appelées au titre d'une instance de la classe en question.
14) Les fonctions membre d'une classe accèdent directement (c'est à dire sans avoir besoin de re-préciser le nom de l'instance) aux membres de l'instance au titre de laquelle elles sont invoquées.

Si l'opération d'affectation (point 8) est moins simple qu'elle en a l'air, les points 13 et 14, sont, contrairement à l'impression que vous en avez peut-être, d'une simplicité extrême. Les deux caractéristiques qu'ils décrivent se complètent mutuellement pour donner à la classe le

fonctionnement le plus normal qui soit : on ne peut raisonnablement pas demander le calcul de la surface ou du prix sans dire de quel rectangle on parle (point 13) et, une fois qu'on l'a dit, le calcul effectué ne porte évidemment pas sur un autre rectangle (point 14).

7 - J'ai rien compris, est-ce que quelqu'un d'autre pourrait m'expliquer ça un peu plus clairement ?

J'ai bien peur de ne pas avoir grand chose à ajouter à ce que j'ai écrit à ce propos à la fin de la Leçon 2.

Horton introduit la notion de fonction dans le chapitre 5 de notre manuel (pages 175 à 206). Comme à l'accoutumé, il en dit beaucoup plus que moi, ce qui rend son texte peu utilisable pour un lecteur ne maîtrisant pas les différentes notions auxquelles il fait allusion, mais le rend utile à qui recherche, justement, une explication sur l'interaction entre deux notions élémentaires.

8 - Pré-requis de la Leçon 4

La Leçon 4 ne dépend pas de façon critique du contenu de la Leçon 3.

N'en faites quand même pas un prétexte pour prendre du retard ! La suite du cours exige, dès la Leçon 5, une bonne maîtrise du contenu de la Leçon 3 et du TD qui lui est associé.