Centre Informatique pour les Lettres et les Sciences Humaines

C++ : Leçon 5
Fonctions avec paramètres

Les fonctions que nous avons écrites jusqu'à présent communiquent entre elles par le biais de variables membre auxquelles elles ont toutes accès. Cette façon de procéder présente deux défauts majeurs :

- Elle n'est envisageable que dans la mesure où les fonctions qui doivent échanger de l'information sont toutes membres de la même classe.
- Les variables membre constituent un pool unique, et tout besoin de communication entre deux fonctions membre va donner lieu à la création de nouvelles variables. Lorsque la complexité d'une classe augmente, le nombre de variables membre risque donc d'augmenter encore plus vite, ce qui va se traduire par une perte de lisibilité qui entraînera tôt ou tard des erreurs de programmation.

Nous savons déjà que le langage C++ propose une autre technique de communication entre fonctions, puisque, lors de la Leçon 3, nous avons signalé que "*Les données à utiliser sont transmises à la fonction à l'aide du mécanisme de passage de paramètre*". Le moment est maintenant venu de nous intéresser à cette technique, ce qui suppose de compléter nos connaissances sur la création de fonctions.

1 - 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 d'une sorte 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 :

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

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 en aucun cas entraîner quelque conséquence que ce soit du point de vue du fonctionnement du programme. En clair : tant que toutes les variables dont les définitions figurent dans un même bloc reçoivent des noms différents, vous pouvez renommer librement les variables sans changer quoi que ce soit au fonctionnement du programme. En particulier,

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[1] 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] L'usage du mot "variable" dans ce contexte n'est pas parfaitement orthodoxe. Si vous vous adressez à un puriste, dîtes "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.

```
//exemple de fonction utilisant un paramètre
1 void maFonction(int sonParametre)
2 {
3 int saVariableLocale = 36;
  //ici prennent place les instructions décrivant le traitement effectué
4 }
```

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.

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 les noms du paramètre et de la variable éventuellement utilisée lors de l'appel n'ont strictement aucune importance, il n'en va bien entendu pas de même pour leur type. Il ne serait quand même pas très raisonnable de transmettre une chaîne de caractères ou 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 son contenu) 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);
```

Fonctions utilisant plusieurs paramètres

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

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

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 seconde valeur au second paramètre, et ainsi de suite.

2 - Déclaration de fonctions

La pleine exploitation des possibilités offertes par les fonctions et leurs paramètres nécessite que nous revenions sur le processus de création de fonctions, de façon à compléter les connaissances acquises lors de la Leçon 3.

Notion de déclaration

Le langage C++ est conçu pour permettre de développer des programmes très volumineux, et il suppose donc a priori que l'ensemble du texte source n'est pas stocké dans un unique fichier. Lorsque le texte source est réparti dans plusieurs fichiers (qui portent en général l'extension .cpp), il est nécessaire que chacun d'entre eux soit compilable indépendamment des autres.

> Il s'agit là d'une exigence très forte, mais dont la logique est claire. Si la compilation ne pouvait porter que sur l'ensemble du programme, la plupart des bénéfices liés à la fragmentation du projet disparaîtraient en effet : il deviendrait difficile d'en confier différentes parties à différentes équipes, et toute modification apportée au programme imposerait une recompilation complète du programme. Quand on sait que certains programmes ne peuvent être recompilés intégralement en moins d'une journée, on comprend qu'exiger l'autonomie de compilation des fichiers est, somme toute, un moindre mal...

Bien entendu, même lorsque le texte source est réparti dans plusieurs fichiers, il décrit toujours un programme unique. Les fonctions définies dans un fichier ont par conséquent fréquemment besoin de faire appel à des fonctions définies dans les autres fichiers source. L'autonomie de compilation du fichier exige donc que le compilateur puisse traiter l'appel d'une fonction dont il ignore la définition. Or, pour traiter un appel de fonction, le compilateur n'a réellement besoin que de deux choses : il lui faut connaître son type et ceux de ses paramètres.

> Le type de la fonction doit être connu, car, si la fonction renvoie un résultat, celui-ci devra généralement soit être stocké dans une variable, soit être utilisé directement. Dans un cas comme dans l'autre, le type de la valeur renvoyée est important, car les variables utilisables et les opérations applicables en dépendent directement. Le type des arguments est également indispensable, car il permet au compilateur de détecter certaines erreurs de programmation (passer une chaîne de caractères à une fonction qui attend une valeur numérique, par exemple), de procéder à certaines conversions de type (une constante littérale entière sera par exemple convertie en format décimal si elle doit initialiser un paramètre de type `double`) et, plus fondamentalement, de générer les instructions qui assureront effectivement la transmission des paramètres à la fonction.

Le bloc de code C++ définissant le corps de la fonction appelée, en revanche, n'est pas nécessaire à la compilation d'une ligne de code contenant un appel. Ce n'est qu'au moment où l'appel devra être *exécuté* (ce qui n'est pas du tout à l'ordre du jour pendant la compilation) que le corps de la fonction deviendra nécessaire, et ce n'est pas du code source dont il faudra alors disposer, mais de l'ensemble d'instructions exécutables généré par le compilateur lors du traitement du fichier où figure la définition de la fonction en question.

> Bien entendu, pour que la séquence d'instructions correspondant à l'appel d'une fonction soit complète, il faudrait qu'elle indique *où* se trouve l'ensemble d'instructions généré à l'occasion de la compilation de cette fonction. Du fait de l'autonomie de compilation des fichiers, cette indication *ne peut pas* être fournie par le compilateur. Compléter les appels de fonctions en fournissant les adresses nécessaires[2] est l'une des tâches qui reviennent au linker (cf. Leçon 1), programme dont l'exécution exige que *toutes* les fonctions aient été compilées.

Comment le compilateur peut-il, lors de la compilation d'un fichier, connaître le type et les arguments d'une fonction qui est définie dans un autre fichier ? Puisque le principe d'autonomie de compilation interdit au compilateur d'aller consulter cet autre fichier, il faut nécessairement fournir ces informations indépendamment de la définition de la fonction, sous la forme de ce que l'on appelle une déclaration.

Déclarer une fonction, c'est indiquer son type, son nom et les types de ses paramètres.

Concrètement, la déclaration d'une fonction ressemble d'assez près à la première ligne de la définition de la fonction en question, comme le montre l'exemple suivant :

[2] Qu'il s'agisse de fonctions ou de variables, le processeur ne peut accéder aux objets que s'il en connaît l'adresse. Les noms sont un confort réservé au texte source, ils disparaissent inéluctablement lors de la compilation. Au niveau du code exécutable, l'appel d'une fonction se traduit donc par l'ordre d'aller exécuter du code situé à une certaine adresse.

```
//exemple de déclaration d'une fonction
1 void maFonction(int unParametre);

//la définition de la fonction déclarée ci-dessus
2 void maFonction(int unParametre)
3 {
//… corps de la fonction : définitions de variables locales, instructions
4 }
```

Le point virgule qui conclut la déclaration revêt donc une importance particulière : c'est le seul indice qui permet au compilateur de reconnaître qu'il s'agit d'une simple déclaration et non de la première ligne d'une définition. La déclaration d'une fonction est parfois appelée le prototype de la fonction, un terme hérité du langage C.

On peut remarquer que, dans l'exemple précédent, la déclaration mentionne non seulement le type du paramètre, mais aussi son nom. Cette mention n'est nullement requise par la syntaxe du langage. La déclaration suivante aurait été parfaitement admissible :

```
void maFonction(int);
```

En situation réelle, toutefois, les aspects syntaxiques ne sont pas les seuls qu'il convient de prendre en compte, et le fait de donner un nom aux paramètres permet d'indiquer au lecteur quel est le rôle joué par chacun d'entre eux. Comparez les deux déclarations suivantes :

```
float volumeTuyaux(float rayon, float longueur);
float volumePoutreCarree(float, float);
```

Il semble à peu près évident que les deux arguments de la fonction `volumePoutreCarree()` indiquent l'un la longueur de la poutre et l'autre la taille d'un côté de sa section. Mais lequel des deux est la longueur ? Pour utiliser effectivement cette fonction, nous allons devoir utiliser une autre source d'information : il va falloir lire un manuel, ou consulter le code présent dans la définition de la fonction, s'il est disponible. La déclaration de `volumeTuyaux()`, en revanche, nous donne assez d'informations pour utiliser la fonction.

Il est préférable que la déclaration d'une fonction mentionne un nom judicieusement choisi pour chacun des paramètres.

Déclaration ? Définition ?

Cette distinction ne concerne pas uniquement les fonctions, mais s'applique en fait à tous les objets portant un nom (fonctions, variables, types, …). On peut donc essayer d'en donner une description en des termes assez généraux, même si la question de la simple *déclaration* de variables et de types nous concerne moins pour l'instant.

Quoi qu'il arrive, un nom ne peut être utilisé dans un programme que s'il a préalablement été déclaré. La déclaration consiste simplement à mentionner le nom en question et le genre d'objet dont il s'agit.

Un même objet peut être déclaré plusieurs fois, à condition que toutes ses déclarations soient équivalentes (c'est à dire qu'elles soient d'accord entre elles sur la nature de l'objet).

Déclarer un objet, c'est dire au compilateur quelque chose comme : "Bon, je te préviens, il y a un `Truc` qui s'appelle `CommeCa`, tu fais comme si tu étais au courant, je t'expliquerai plus tard."

Dans le cas d'une fonction, pour "faire comme s'il était au courant", le compilateur a besoin du type, du nom et des types des paramètres. C'est pourquoi ces informations figurent obligatoirement dans la déclaration d'une fonction, qui revient donc à dire quelque chose comme : "Bon, je te préviens, il y a une `fonction` qui s'appelle `maFonction()`, elle ne renvoie rien mais elle a besoin d'une valeur de type `int` pour initialiser son unique paramètre. Tu fais comme si tu étais au courant, je t'expliquerai plus tard quel traitement il faut qu'elle effectue."

La définition d'un objet, pour sa part, commence comme une déclaration, et y ajoute toutes les informations supplémentaires qui peuvent être nécessaires pour permettre au compilateur de créer effectivement l'objet. Comme la définition comporte une déclaration, beaucoup d'objets sont en fait directement définis et ne font jamais l'objet d'une simple déclaration (les variables locales d'une fonction, par exemple, ne sont jamais déclarées avant d'être définies).

Un même objet ne peut être défini qu'une seule fois.

Définir un objet préalablement déclaré, c'est dire au compilateur quelque chose comme : "Tu te souviens du `Truc CommeCa` dont je t'avais parlé ? Et bien, voilà de quoi il s'agit exactement..."

Dans le cas d'une fonction, la seule information nécessaire qui ne soit pas déjà présente dans la déclaration, c'est le corps[3]. C'est pourquoi la définition d'une fonction ressemble beaucoup à une déclaration où le bloc de code décrivant le traitement que la fonction doit effectuer aurait pris la place du point-virgule final.

Où et quand doit-on déclarer les fonctions ?

Comme la déclaration des membres fait partie intégrante de la définition de la classe, le cas des fonctions membre est simple : leur déclaration figure nécessairement dans le fichier qui définit la classe (et qui, normalement, porte le nom de la classe suivi d'une extension ".h"). Nous connaissons déjà l'organisation générale de la définition d'une classe, et la présence de paramètres dans la déclaration de certaines fonctions n'y introduit aucun bouleversement fondamental :

```
//définition de la classe (figure normalement dans un fichier .h)
1 class CMaClasse
2 {
3 public:
4    double maFonction(int sonParametre); //déclaration de la fonction membre
5 };
```

Habituellement, la définition des fonctions membre figure dans un fichier différent :

```
//définition de la fonction membre (figure normalement dans un fichier .cpp)
1 double CMaClasse::maFonction(int sonParametre)
2 {
  //… définitions de variables locales, instructions
3 }
```

Nous avons vu (Leçon 3) que, lorsqu'une fonction membre est très brève, il arrive qu'on choisisse de la définir directement pendant la définition de la classe, plutôt que de rejeter cette définition dans un fichier .cpp. Cette façon de procéder ne déroge en rien à la règle exigeant la déclaration des membres dans la définition de la classe, puisqu'une définition "contient" une déclaration :

```
//définition de la classe (figure normalement dans un fichier .h)
1 class CMaClasse
2 {
3 public:
     //définition (et, donc, déclaration) de la fonction membre
4    double maFonction(int sonParametre) { return 3.14 * sonParametre; }
5 };
```

Dans le cas des fonctions globales (c'est à dire des fonctions qui ne sont membres d'aucune classe), on adopte généralement la même logique : les fonctions sont déclarées dans un fichier .h, et sont définies dans un fichier .cpp, la correspondance entre les deux fichiers étant traditionnellement signalée en leur donnant le même nom.

Utiliser les déclarations

Placer la déclaration d'une fonction dans un fichier .h peut sembler paradoxal, puisque nous venons de voir que cette déclaration doit figurer dans tout fichier où figure un appel à cette fonction. Or, si un fichier contient un appel à une fonction, c'est qu'il contient du code définissant une fonction appelante, ce qui signifie, d'après nos conventions, qu'il s'agit a priori d'un fichier .cpp. Si les déclarations sont utiles dans les fichiers .cpp, pourquoi les placer dans des fichiers .h ?

La justification de cette façon de procéder réside dans le fait que la déclaration d'une fonction doit figurer dans *tout* fichier où figure un appel à cette fonction. Si nous choisissons de placer

[3] Sauf, bien entendu, s'il s'agit d'une de ces déclarations mal élevées qui se dispensent de nommer les paramètres. Il faut impérativement nommer les paramètres lors de la définition de la fonction, puisque c'est indispensable pour pouvoir utiliser les valeurs transmises par la fonction appelante.

directement une déclaration de la fonction dans tous les fichiers .cpp qui en ont besoin, nous créons de multiples lignes de code déclarant la même fonction. Ces redéclarations ne posent pas de problème du point de vue du langage, mais elles risquent de créer des difficultés de gestion.

> Si l'on procède ainsi, toute modification du type d'un paramètre, par exemple, exige que toutes les déclarations de la fonction soient mises à jour. Si l'on oublie d'actualiser l'un des fichiers .cpp, celui-ci contient une déclaration qui n'est plus conforme à la réalité, et le compilateur n'est pas en mesure de détecter que les appels à la fonction contenus dans le fichier en question sont devenus, eux aussi, incompatibles avec la nouvelle version de la fonction. Dans certains cas, cette situation peut se traduire par un programme qui semble fonctionner normalement, mais dont les résultats sont faux...

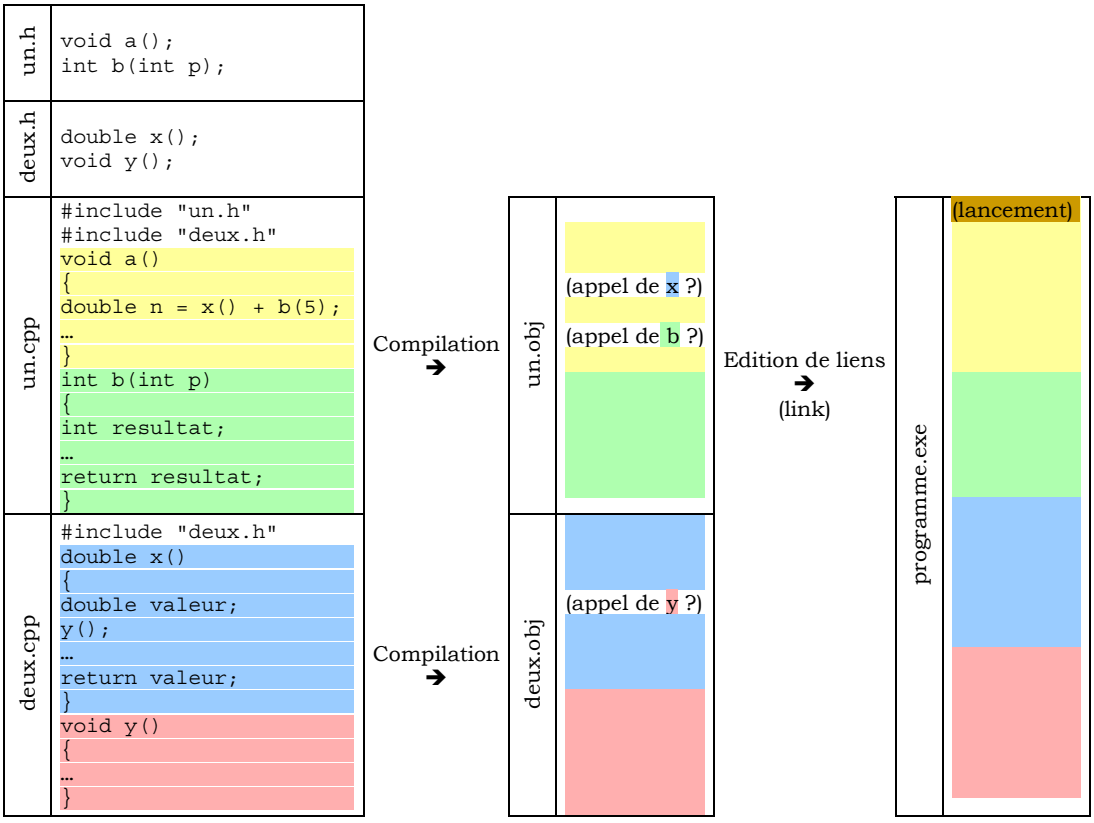
Lorsque les déclarations sont placées dans des fichiers .h, il n'est plus nécessaire de les dupliquer dans tous les fichiers qui en ont besoin : il suffit d'indiquer, partout où une déclaration serait nécessaire, dans quel fichier .h elle se trouve.

> Si l'on procède ainsi, lorsque la seule et unique déclaration d'une fonction est modifiée, tous les fichiers qui en font usage adoptent nécessairement la nouvelle déclaration, sans que le programmeur ait à s'en soucier.

Les déclarations contenues dans un fichier .h sont rendues disponibles à l'aide de la directive de compilation `#include`. Si, par exemple, un fichier .cpp contient un appel à une fonction dont la déclaration est contenue dans un fichier nommé mesDeclarations.h, on insérera en début du fichier .cpp la ligne

```
#include "mesDeclarations.h"
```

L'utilisation des différents types de fichiers lors de la création d'un programme peut être résumée par la figure suivante :



Des fichiers source au programme exécutable

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 ces paramètres. Cette exigence est un inconvénient, particulièrement lorsqu'on cherche à mettre au point des fonctions réutilisables

(c'est à dire qui ne sont pas destinées à un programme particulier, mais serviront dans le cadre de nombreux projets différents).

> En effet, pour rendre une fonction capable de traiter un grand nombre de cas particuliers, il est souvent nécessaire de la doter d'un grand nombre de paramètres qui permettent, 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[4] et les valeurs par défaut.

La surcharge (opération consistant à définir plusieurs fonctions ayant le même nom, mais se distinguant par le type de leurs paramètres) n'est pas spécifiquement conçue pour répondre au problème des listes d'arguments pléthoriques. Même si on peut envisager de l'utiliser dans des cas où il s'agit simplement de réduire le nombre des valeurs à transmettre, son ambition va au-delà de ce problème, et nous y reviendrons lors de la Leçon 13.

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 MFC ont très fréquemment recours à ces deux mécanismes, dont un des aspects les plus séduisants est que le programmeur qui *utilise* les fonctions n'a pas à s'en occuper : il se contente de bénéficier de la souplesse d'utilisation ainsi obtenue, les problèmes ont été pris en charge, une fois pour toutes, par celui qui a *écrit* la (ou les) fonction(s).

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[5].

```
//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.

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

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 !
```

[4] Le terme anglais est "overloading".
[5] Ou directement après le type du paramètre, si vous avez pris la déplorable habitude de déclarer les fonctions sans nommer les paramètres.

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 :

```
//extrait du fichier maClasse.h
1  class CMaClasse
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
```

Il existe toutefois une restriction à l'usage des valeurs par défaut : si l'un paramètre 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 initiales devant être adoptées par les paramètres suivants.
>
> Cette contrainte explique pourquoi, lorsqu'un paramètre possède une valeur par défaut, tous les paramètres suivants doivent en avoir une également. 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. Celui-ci ne disposerait donc ni d'une valeur par défaut ni d'une valeur transmise lors de l'appel, et l'exécution de la fonction deviendrait impossible.

Si la classe a été définie ainsi

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

`maFonction()` pourra être appelée comme ceci :

```
6  uneInstanceDeMaClasse.maFonction('k',12); //p1 vaudra 'k' et p2 vaudra 12
7  uneInstanceDeMaClasse.maFonction('w');    //p1 vaudra 'w' et p2 vaudra 7
8  uneInstanceDeMaClasse.mafonction();       //p1 vaudra 'x' et p2 vaudra 7
```

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 - Bon, c'est gentil tout ça, mais ça fait déjà 9 pages. Qu'est-ce que je dois vraiment en retenir ?

1) 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.
2) Les paramètres d'une fonction sont comme des variables locales qui seraient initialisées à l'aide de valeurs transmises par la fonction appelante.

3) 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.

4) La définition d'une classe inclut la déclaration de chacun de ses membres.

5) La déclaration d'une fonction inclut nécessairement son type, son nom et les types de ses paramètres.

6) En C++, le texte contenu dans un fichier ne peut rien utiliser qui n'ait été préalablement déclaré. Plutôt que de répéter explicitement les mêmes déclarations dans de multiples fichiers, on regroupe ces déclarations dans des fichiers .h qu'on `#include` là où les déclarations qu'ils contiennent sont nécessaires.

7) Pour qu'un paramètre possède une valeur par défaut, celle-ci doit être mentionnée dans la déclaration de la fonction[6].

8) Un paramètre ne peut avoir de valeur par défaut qu'à condition que tous les paramètres suivant en aient aussi.

9) Lorsqu'un paramètre possède une valeur par défaut, on peut appeler la fonction en omettant de fournir une valeur pour initialiser ce paramètre.

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. Ils doivent donc tous adopter la valeur par défaut qu'ils possèdent nécessairement (en vertu du point 8).

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

Le mécanisme de passage de paramètres à une fonction est présenté dans le chapitre 5 du livre de Horton (pages 183 et 184). La distinction entre définition et déclaration n'est pas présentée formellement dans le manuel. Comme beaucoup d'auteurs, Horton traite la question de façon diffuse (par petites touches successives, lorsque le besoin se fait sentir) et fait preuve d'un grand laxisme au niveau terminologique : le mot "déclaration" est utilisé des centaines de fois pour désigner de véritables définitions. Cet abus est très répandu (et partiellement justifié par le fait que toute définition inclut une déclaration), mais il n'est certainement pas favorable à l'éclaircissement de ces notions dans l'esprit du débutant... L'utilisation de valeurs par défaut pour les paramètres d'une fonction est, elle, exposée au cours du chapitre 6 (page 216).

> La présentation du langage proposée par Horton est si différente de la mienne que, pour bénéficier pleinement de ses explications, vous serez sans doute conduit à lire de larges portions du manuel traitant de points qui n'ont pas encore été abordés dans les Leçons[7]. Mais, comme d'habitude, si vous avez vraiment du mal à suivre les Leçons, c'est sans doute une approche radicalement différente qu'il vous faut. J'espère que celle de Horton vous conviendra mieux.

6 - Pré-requis de la Leçon 6

Il y a un examen partiel avant la leçon 6.

> Il vaudrait mieux que vous ayez fait les TD 1 à 5 et que vous soyez à peu près au clair avec les points essentiels des Leçons 1 à 5...
>
> Les "Annales" proposent des sujets utilisés lors de sessions passées, ainsi que des propositions de corrigés.

[6] 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 programmes peuvent se retrouver en désaccord quant aux valeurs par défaut d'un même paramètre d'une même fonction....)

[7] Et même de points qui ne seront *jamais* traité dans aucune Leçon (sauf dans le cas, fort improbable, où l'Université de Provence me demanderait de concevoir un module "Programmation en C++ – niveau 7").