CM – Programmation C / C++

Chapitre 4 : Fonctions

Un programme est un ensemble de tâches, souvent répétitives. Afin d'éviter de réécrire la même tâche plusieurs fois, on peut la décrire par une fonction qui sera appelée à chaque fois qu'on en a besoin (pour faire un calcul, afficher un résultat, sauver ou lire des données, …). Créer des fonctions permet d'organiser le programme, ce qui le rend plus lisible et facile à modifier tout en limitant les erreurs.

I- Description d'une fonction :

1) Syntaxe :

Une fonction s'écrit de la manière suivante :

```
type Nom(paramètres)
{
instructions;
}
```

Le type de la fonction correspond à la valeur renvoyée par cette dernière (entière, réelle, chaîne de caractères, …). Si la fonction ne renvoie aucune valeur, son type est **void**. Elle possède un nom qui est donné par le programmeur à sa création. Entre les parenthèses `()` sont inscrits (ou non, dans ce cas il faut quand même laisser les parenthèses) les paramètres de la fonction, c'est-à-dire une liste de variables extérieures passées dans la fonction à son appel. Chaque paramètre est défini par un type et un nom, dont le dernier peut être différent des noms des variables dans le `main()`, et ils sont séparés les uns des autres par une virgule. La portée des paramètres est le bloc formant le corps de la fonction même si ces derniers sont définis avant les accolades `{}`.

2) Exemple d'utilisation de fonction :

On créé un programme qui permet d'entrer deux entiers et de dire lequel des deux est le plus petit, puis de même avec deux entiers choisis lors de l'écriture du code. La partie du code qui permet de faire la comparaison entre les deux entiers et la restitution du plus petit sera écrite dans une fonction à l'extérieur du `main()` et qui sera appelée dès que nécessaire.

```cpp
#include <iostream>

using namespace std;


int min(int a, int b)
{
    if(a<b) return a;
    else return b;
}

int main()
{
    int val1;
    int val2;
    cout << "Entrer 2 entiers :" << endl;
    cin >> val1 >> val2;

    cout << "Le plus petit est :" << min(val1,val2) << endl;

    int valeurMin=min(5,6);
    cout << "Minimum entre 5 et 6 :" << valeurMin << endl;

    cout << "Autre methode :" << min(5,6) << endl;

    return 0;
}
```

La fonction a comme type **int** donc elle doit retourner un entier. Parmi les instructions qui la compose, on trouve la commande **return** qui retourne ici une variable entière. Lorsque le programme atteint cette commande, on sort de la fonction.

La fonction est appelée dans le par son nom, suivi des parenthèses et contenant (ou non) un ou des paramètres séparés par une ou des virgules (et non pas des points comme les réels). Cela permet d'appeler plusieurs fois la fonction et donc de ne pas réécrire le même texte à chaque fois qu'il le faut.

Le résultat de la fonction peut être directement utilisé par d'autres opérateurs (ici, les opérateurs de flux de sortie pour afficher la valeur renvoyée par la fonction dans le premier et troisième cas) ou stocké dans une variable (comme dans le second cas avec la variable entière `valeurMin`).

3) Fonction qui ne renvoie rien (« void ») :

On créé un programme qui permet d'entrer deux entiers et d'afficher la somme des deux, puis on fait la même chose avec deux entiers choisis lors de l'écriture du code. La partie du code qui permet d'additionner les entiers et d'afficher le résultat sera écrite dans une fonction.

```
#include <iostream>

using namespace std;


void AfficheSomme(int a, int b)
{
    cout << "Somme =" << a+b << endl;
}

int main()
{
    int val1;
    int val2;
    cout << "Entrer 2 entiers :" << endl;
    cin >> val1 >> val2;

    AfficheSomme(val1,val2);
    AfficheSomme(5,6);

    return 0;
}
```

Comme dit précédemment, une fonction qui ne renvoie rien utilise le type **void**. Il n'y a pas besoin de commande **return** mais il est possible de l'utiliser (sans mettre d'arguments) pour sortir de la fonction avant sa fin.

Qu'une fonction renvoie une valeur ou non, elle s'appelle de la même manière. Cependant, une fonction **void** ne peut pas être utilisée par un opérateur.

4) Fonction sans paramètres :

On crée un programme similaire au programme précédent, excepté que c'est la partie du code qui déclare et affecte une valeur aux variables qui sera écrite dans une fonction.

```
#include <iostream>

using namespace std;


int saisie()
{
    cout << "Entrer un entier";
    int n;
    cin >> n;

    return n;
}

int main()
{
    int val1=saisie();
    int val2=saisie();

    cout << "Somme =" << val1+val2 << endl;

    return 0;
}
```

Dans ce cas, il n'y a pas de variables en paramètres, mais on doit garder les parenthèses afin que le compilateur comprenne que c'est une fonction et pas un variable.

La fonction s'appelle avec des parenthèses vides, toujours pour dire que ce n'est pas une simple variable. Il faut aussi faire attention à ne pas confondre les différentes variables : `n` est la variable utilisée dans la fonction, `val1` et `val2` sont deux variables appartenant au `main()` auxquelles on affecte une valeur en appelant la fonction `saisie()`. `n` n'est pas connue par le `main()` tandis que `val1` et `val2` ne sont pas connues par la fonction.

II- Prototypage des fonctions :

1) Principe du prototypage :

Puisque l'on peut créer des fonctions qui sont écrites en dehors du `main()` d'un programme, on peut aussi les écrire dans un fichier différent pour les utiliser de l'un à l'autre. Or, il faudrait écrire ces mêmes fonctions dans les deux fichiers, ce qui pose un problème car on ne doit pas dupliquer la définition d'une fonction (si un dossier de projet est créé avec deux fichiers ayant les mêmes fonctions, le compilateur ne saura pas quelle fonction utiliser entre les deux fichiers).

Pour régler ce problème, on utilise la notion de « déclaration » de la fonction ou « prototypage » :

```cpp
int min(int a, int b)
{
    if(a<b) return a;
    else return b;
}

int abs(int a)
{
    if(a>0) return a;
    else return -a;
}
```

```cpp
#include <iostream>

using namespace std;


int min(int a, int b);
int abs(int a);

int main()
{
    int val1, val2;
    cout << "Entrer 2 entiers :";
    cin >> val1 << val2;

    cout << "Le plus petit est ";
    cout << min(val1,val2) << endl;
    cout << "Valeur absolue de " << val1 << "=" << abs(val1) << endl;

    return 0;
}
```

Le premier fichier, nommé « Utile.cpp », contient les fonctions utilisées dans le second fichier, appelé « main.cpp ». Dans ce dernier, les fonctions sont déclarées (généralement écrites au-dessus de la partie du `main()`) mais le compilateur ne sait pas comment elles fonctionnent (pas d'instructions), il sait juste qu'elles existent quelque part. On appelle cela les « prototypes » des fonctions. Le nom des variables dans les prototypes n'est pas nécessaire (on aurait pu écrire `int abs(int);`), mais il faut ne pas oublier le point-virgule `;` en fin de ligne.

C'est un autre programme appelé « linker » qui fera le lien entre les deux fichiers. Généralement, les prototypes sont écrits dans un fichier d'entête, aussi appelé « header » et utilisant l'extension « .h ». Ce fichier de prototypes sera alors inclus dans les autres fichiers via la commande `#include`.

```cpp
#include <iostream>

using namespace std;

#include "Utile.h"


int main()
{
    int val1, val2;
    cout << "Entrer 2 entiers :";
    cin >> val1 << val2;

    cout << "Le plus petit est ";
    cout << min(val1,val2) << endl;
    cout << "Valeur absolue de " << val1 << "=" << abs(val1) << endl;

    return 0;
}
```

```cpp
#ifndef UTILE_H_INCLUDED
#define UTILE_H_INCLUDED

int min(int a, int b);
int abs(int a);

#endif // UTILE_H_INCLUDED
```

Le premier fichier est le fichier « main.cpp » modifié avec l'inclusion du fichier « Utile.h », nouveau fichier qui contient les prototypages des fonctions. Le fichier « Utile.cpp » n'est pas modifié.

Pour les headers, on utilise des guillemets « » au lieu des <> qui sont réservés aux bibliothèques du C++. On peut faire une inclusion de plusieurs fichiers, ce qui permet d'éviter de faire du copier-coller.

2) *Protection des headers :*

On prend le cas où l'on a trois headers différents avec des noms différents, « Mult.h », « Puiss.h » et « Reso.h », ayant chacun une fonction précise et dont les deux derniers sont inclus dans un fichier « main.cpp ». Le fichier « Mult.h » est, quant à lui, inclus dans les deux headers « Puiss.h » et « Reso.h », ce qui fait qu'on a la définition de la même fonction deux fois et ce n'est pas possible. Il faut mettre une protection au header pour qu'il soit inactif après le premier passage.

```
#ifndef MULT_H_INCLUDED
#define MULT_H_INCLUDED

int multiplier(int x, int y);

#endif // MULT_H_INCLUDED
```

Ces protections sont les commandes `#infdef` (« if not defined », si non défini) et `#endif` qui permettent, si on repasse une seconde fois dans le fichier qui est déjà défini, de directement se retrouver à la fin et de ne pas le relire. Cette protection est mise automatiquement par Code::Blocks, ce qui est très utile.

3) *Surcharge de fonction :*

Dans un même fichier, il est possible d'avoir plusieurs fonctions portant le même nom à condition que la liste des paramètres soit différente pour chacune, en termes de nombre ou de type. Cependant, on ne peut pas différencier deux fonctions par le nom des paramètres ou le type que retourne la fonction.

```
void Affiche(int a);
void Affiche(float a);
void Affiche(int a, int b);
void Affiche(int a, float b);
void Affiche(float a, int b);
```

```
int min(short a, short b);
short min(short a, short b);
```

```
int min(short a, short b);
int min(short c, short d);
```

Les cinq fonctions `Affiche()` sont toutes différentes car les paramètres sont tous différents les uns des autres, tandis que les quatre fonctions `min()` sont identiques car changer le type de ce qui est renvoyé ou le nom des variables ne différencie pas les fonctions entre elles.

III- Passage des arguments :

1) *Passage par valeur :*

Les arguments (paramètres) d'une fonction sont passés par valeur, c'est-à-dire que ces variables copient la valeur des variables utilisées lors de l'appel de la fonction. Si les valeurs des arguments sont modifiées, les valeurs des variables utilisées pour l'appel de la fonction ne le seront pas.

```cpp
#include <iostream>

using namespace std;


void incremente(int i)
{
    i=i+1;
    cout << "Valeur dans la fonction = " << i << endl;
}


int main()
{
    int k=10;

    cout << "Valeur avant appel = " << k << endl;
    incremente(k);
    cout << "Valeur apres appel = " << k << endl;

    // Appel avec une valeur.
    incremente(4);

    return 0;
}
```

Dans ce code, on crée la fonction `incremente()` qui ne renvoie rien et qui prend en argument la variable entière `i` pour l'incrémenter et afficher sa valeur dans la fonction. Dans le `main()`, on déclare une nouvelle variable `k` initialisée à `10`, puis on affiche sa valeur avant appel de la fonction, on appelle la fonction et on affiche sa valeur après appel. On remarque que `k` vaut `10` avant l'appel, que `i` lors de l'appel vaudra `11` et qu'après l'appel, `k` vaudra toujours `10`, ce qui est normal car sa valeur a été copiée pour être utilisée par la fonction mais vu qu'elle ne renvoie rien, `k` reste identique.

On remarque aussi que l'on peut appeler une fonction en mettant comme argument une valeur directement.

2) Passage par référence :

Le passage des arguments par valeur n'est pas toujours souhaitable quand la fonction doit changer la valeur d'un ou des arguments ou quand l'argument de la fonction est un objet de grande taille, ce qui induit des coûts en temps et en mémoire très élevés. Ainsi, on peut passer les arguments par référence, c'est-à-dire que le paramètre de la fonction est un nouveau nom de la variable utilisée comme argument. Toute modification des arguments entraînera une modification des variables utilisées pour la fonction dans le `main()`.

Pour la syntaxe, on ajoute un juste après le type de l'argument lors de l'écriture de la fonction. L'appel de la fonction reste inchangé.

```cpp
void incremente(int& i)
{
    i=i+1;
    cout << "Valeur dans la fonction = " << i << endl;
}
```