Les fonctions et gestion de la mémoire

Les fonctions

Une fonction est un bloc d'instructions éventuellement avec des paramètres et pouvant fournir un résultat nommé « valeur de retour ». Elle porte un nom suivi forcément de parenthèses.

Le nom de la fonction peut apparaître à trois endroits différents dans le code :

- Déclaration
- Définition
- Appels

Déclaration de fonctions

Elle est constituée du prototype de la fonction : on déclare au compilateur l'existence de cette fonction. Généralement, ce prototype est placé dans un fichier d'entête (.h)

Exemple :

```
float CalculerRendementEnergie(float, float, int);
```

Cela permet au compilateur de ne pas signaler d'erreurs s'il trouve dans le code à la suite de cette déclaration, le nom de cette fonction et que les types des arguments sont respectés.

Dans un prototype, il n'est pas nécessaire de mettre des noms de variables pour les arguments. Seuls les types sont importants.

Définition de fonctions

Elle est constituée du corps de la fonction. Celui-ci reprend évidemment la même signature que le prototype. Il est nécessaire de donner des noms aux arguments afin d'utiliser leur valeur dans le code de la fonction. Ses paramètres correspondent à des déclarations de variables : ce sont des variables locales à la fonction.

```
float CalculerRendementEnergie(float energieConso, float energieProd, int
mode)
{
    float rendement;

    if (mode == 1 )
    {
        rendement = (energieProd*0.8/energieConso)*100;
    }
    else
    {
        rendement = (energieProd/energieConso)*100;
    }
    return rendement;
}
```

Remarques : Les variables locales à la fonction (ici *rendement*) ainsi que tous les paramètres de la fonction sont déclarés dans la pile. Ainsi leur portée (scope) est limitée à l'exécution de la fonction.

Appels de fonctions

Le but ultime d'une fonction est qu'elle soit utilisée donc appelée !

L'appel d'une fonction est forcément réalisé au sein d'une autre fonction. Cette autre fonction s'appelle : la fonction appelante.

La première fonction à s'exécuter est la fonction *main()* donc c'est elle qui débute les appels des différentes fonctions. On peut avoir par exemple, la fonction *main()* qui appelle une fonction *f1()* qui elle-même appelle une fonction *f2()*, qui elle-même appelle une fonction *f3()* etc..

Exemple de test de la fonction **`CalculerRendementEnergie() :`**

```
1   #include <iostream>
2   #include "fonctions.h"
3
4   using namespace std;
5
6   int main()
7   {
8       float unrendement;
9
10      unrendement = CalculerRendementEnergie(10.5,8.5,1);
11
12      cout << unrendement;
13
14      return 0;
15  }
```

La ligne 10 constitue l'appel de la fonction *CalculerRendementEnergie()*

Les valeurs « 10.5 » « 8.5 » et « 1 » sont des arguments passés à la fonction. Ces arguments se retrouveront dans les paramètres « `energieConso` », « `energieProd` » `et` « `mode` »

La ligne 10 effectue également une affectation : la valeur renvoyée par la fonction (`return rendement;`) est copiée dans la variable *unrendement*.

Mémoire : le tas et la pile

Le tas constitue une zone mémoire associée au programme : c'est symboliquement ce qu'on appelle la mémoire « globale » et pour être exact la mémoire allouée.

Les variables créées dynamiquement sont positionnées dans ce tas. C'est l'une des seules façons de pouvoir conserver des variables entre les exécutions de fonctions.

L'exécution d'une fonction déclenche une sauvegarde du contexte d'exécution de la fonction appelante (sauvegarde des variables locales de la fonction appelante, état des registres du µprocesseur, etc ..) Cette sauvegarde s'effectue dans la pile d'exécution (« stack frame »).

Exemple 1 :

```
/* Remarque : param est un paramètre; il est considéré comme une variable
locale à la fonction.
 * Donc param est déclaré dans la pile (stack frame)
 * */

/* la variable locale varloc est déclarée dans la pile d'exécution de la
fonction f1()
 * A la sortie de la fonction f1(), varloc disparaît puisqu'on aura dépilé
pour revenir à la fonction appelante.
```

```
 * Ainsi toute modification sur varloc est perdue : la fonction ne sert à
rien !
 * */
void f1(int param)
{
    int varloc;
    varloc = param + 1;
}
```

A l'appel de la fonction f1(), on a :

```
int main()
{
    int unevaleur = 0;

    //Tentative d'appel de f1()
    f1(10);
    unevaleur = varloc; //NE COMPILE PAS : varloc est inconnue dans main()
    cout << unevaleur;

    return 0;
}
```

Exemple 2 :

```
/* la variable locale varloc est déclarée dans la pile d'exécution de la
fonction f2()
 * A la sortie de la fonction f2(), varloc disparaît puisqu'on aura dépilé
pour revenir à la fonction appelante.
 * Par contre, f2() retourne la valeur de varloc en sortant (return
varloc).
 * La modification est donc récupérée dans la fonction appelante en tant
que RValue d'une affectation par exemple.
 * */
int f2(int param)
{
    int varloc;
    varloc = param + 1;
    return varloc;
}
```

A l'appel de la fonction f2(), on a :

```
int main()
{
    int unevaleur = 0;

    //Tentative d'appel de f2()
    unevaleur = f2(10); //COMPILE ET S'EXECUTE
    cout << unevaleur;

    return 0;
}
```

Exemple 3 :

```
/* la variable locale varloc est déclarée dans la pile d'exécution de la
fonction f3()
 * A la sortie de la fonction f3(), varloc disparaît puisqu'on aura dépilé
pour revenir à la fonction appelante.
```

```
 * Avant de sortir, f3() retourne l'adresse de la variable varloc.
Autrement sa position mémoire dans la pile !
 * Le programme plante puisque même si on recupère l'adresse de la
variable, celle-ci n'existe plus!
 * */
int* f3(int param)
{
    int varloc;
    varloc = param + 1;
    return &varloc;
}
```

A l'appel de la fonction f3(), on a :

```
int main()
{
    int unevaleur = 0;
    int* pUneValeur;

    //Tentative d'appel de f3()
    pUneValeur = f3(10); //COMPILE ET PLANTE
    cout << *pUneValeur;

    return 0;
}
```

Exemple 4 :

```
/* la variable pointée par varloc est déclarée dans le tas (mémoire
allouée) grâce à new. Le pointeur varloc (variable locale)
 * contient l'adresse de cette variable. On "travaille" sur la variable
située dans le tas. Celle-ci n'est pas détruite à la sortie de f4().
 * Par contre le pointeur, lui, est détruit à la sortie de f4()
 * Il faut donc, dans la fonction appelante, récupérer l'adresse contenu
dans le pointeur varloc. C'est pourquoi on fait un return varloc
 * */
int* f4(int param)
{
    int *varloc;
    varloc = new int;

    *varloc = param + 1;

    return varloc;
}
```

A l'appel de la fonction f4(), on a :

```
int main()
{
    int unevaleur = 0;
    int* pUneValeur;

    //Tentative d'appel de f4()
    pUneValeur = f4(10); //COMPILE ET S'EXECUTE
    cout << *pUneValeur;

    return 0;
}
```

Seuls les appels à f2() et à f4() sont fonctionnels. La différence entre ces deux versions dépend de ce qu'on voudra faire du résultat de l'exécution. Si le but de votre fonction est de modifier une variable qui sera exploitée par une autre fonction alors il faudra renvoyer et/ou passer l'adresse de la variable.

Cas des tableaux

La déclaration d'un tableau suit le même principe que pour les variables locales.

Exemple 5

```
/* T est un tableau déclaré statiquement donc en local à la fonction f5()
 * Même si on renvoie l'adresse du premier élément du tableau, tout le
tableau disparaît à la fin
 * de la fonction.
 **/

int* f5(int param)
{
    int T[5];

    for (int i = 0 ; i < 5 ; i++)
    {
        T[i] = param;
    }

    return T;
}
```

A l'appel de la fonction f5(), on a :

```
int main()
{
    int unevaleur = 0;
    int* pUneValeur;

    //Tentative d'appel de f5()
    pUneValeur = f5(10); //COMPILE ET PLANTE
    for (int i = 0 ; i < 5 ; i++)
    {
        cout << pUneValeur[i];
    }
    return 0;
}
```

Exemple 6

```
/* Grâce à l'opérateur new T pointe vers un tableau déclaré dynamiquement
dans le tas ; le tableau continue d'exister après la fonction f6()
 * Par contre, la variable pointeur T disparaît à la fin de la fonction
f6()
 * Il faut donc renvoyer ce pointeur en valeur de retour
 * */
int* f6(int param)
{
    int *T;
    T = new int[5];

    for (int i = 0 ; i < 5 ; i++)
    {
```

```
        T[i] = param;
    }

    return T;
}
```

A l'appel de la fonction f6(), on a :

```
int main()
{
    int unevaleur = 0;
    int* pUneValeur;
    //Tentative d'appel de f6()
    pUneValeur = f6(10); //COMPILE ET S'EXECUTE
    for (int i = 0 ; i < 5 ; i++)
    {
        cout << pUneValeur[i];
    }
    return 0;
}
```

Passage de paramètres aux fonctions

Le passage d'argument à une fonction s'effectue que par valeur en C et par valeur et référence en C++.

Passage par valeurs

Tous les arguments en C sont passés aux paramètres par valeur.

```
int main()
{
    float unrendement;

    unrendement = CalculerRendementEnergie(10.5,8.5,1);

    cout << unrendement;

    return 0;
}
```

On communique à la fonction *CalculerRendementEnergie()* les arguments « 10.5 » « 8.5 » et « 1 »

Le code suivant aurait produit le même résultat :

```
int main()
{
    float unrendement;
    float energC, energP;
    int mode;

    energC = 10.5;
    energP = 8.5;
    mode = 1;

    unrendement = CalculerRendementEnergie(energC,energP,mode);

    cout << unrendement;

    return 0;
}
```

Remarque : *energC*, *energP* et *mode* sont des variables locales à la fonction *main()*. Elles sont donc déclarées dans la pile de la fonction *main()*. Elles n'ont donc rien à voir avec les variables locales de la fonction *CalculerRendementEnergie()* même si l'une d'elles (mode) porte le même nom ! Le seul rapport qu'il y a entre les variables locales de la fonction *main()* et les variables locales de la fonction *CalculerRendementEnergie()* est qu'on recopie les valeurs des premières dans les secondes.

Paramètres IN, INOUT et OUT

On peut considérer les trois cas de figure suivants :

- On fournit à une fonction des paramètres en entrée (IN) : On effectue un passage par valeurs

```
/* Fonction avec un paramètre IN
 * */
void fct1(int val)
{
    cout << val;
}
```

Appel de fct1() :

```
int main()
{
    int unevaleur = 1;
    fct1(unevaleur);
    return 0;
}
```

- On désire qu'une fonction renvoie une ou plusieurs valeurs ; paramètres OUT ; deux solutions :
    - La fonction renvoie une valeur : valeur de retour de la fonction

```
/* Fonction avec une valeur de retour
 * */
int fct2()
{
    int data = 10;

    return data;
}
```

Appel de fct2() :

```
int main()
{
    int uneautrevaleur;

    uneautrevaleur = fct2();
    cout << uneautrevaleur;

    return 0;
}
```

    - La fonction renvoie plusieurs valeurs : il faut alors passer des adresses de variables en paramètre. Les adresses sont passées par valeur
- On désire fournir à une fonction des valeurs en entrée et elle doit modifier ces paramètres (INOUT) :

Il faut passer les adresses de variables dont on a renseigné les valeurs : les adresses sont passées par valeur

```
/* Fonction avec :   pval1 paramètre OUT
 *                   pval2 paramètre INOUT
 * */
void fct3(int *pval1, int *pval2)
{
    (*pval2)++;
    *pval1 = *pval2;
}
```

Appel de fct3() :

```
int main()
{
    int unevaleur;  //OUT
    int uneautrevaleur = 20;    //INOUT

    fct3(&unevaleur, &uneautrevaleur);

    cout << unevaleur << endl;
    cout << uneautrevaleur << endl;
    return 0;
}
```

En C++, il existe un autre type de passage : passage par référence avec le &. Cela permet de vraiment passer les adresses de variables et de permettre à une fonction de travailler sur les variables et plus sur des copies de variables.