- Introduction à l'algorithmique et au langage C -

Introduction aux fonctions

TD4

1re année ESIEA - Semestre 1

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

2016 - 2017

Avant propos

Nous avons commencé à faire nos premiers codes et nous avons vu que parfois nous devons réécrire les mêmes lignes plusieurs fois pour réaliser les mêmes actions à différents endroits. Avec les fonctions, nous allons voir comment factoriser ces lignes identiques, pour créer des séquences dédiées à des tâches, que nous pourrons appeler selon nos besoins, comme les fonctions d'affichage et de récupération d'entrées clavier que nous avons déjà vues.

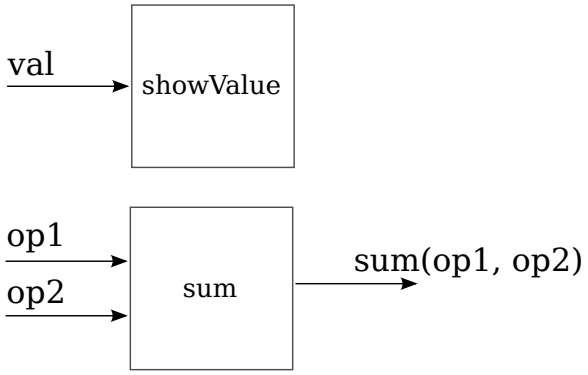
De plus, cela va nous permettre de mieux organiser notre code, plutôt que d'avoir une longue suite d'instructions dans notre `main`*. Nous allons pouvoir regrouper des actions en méta-actions qui deviennent de nouvelles actions atomiques. Le séquencement du programme sera alors décrit dans la fonction* `main` *par des appels successifs et ordonnés de nos fonctions.*

1 Procédures et fonctions : pourquoi ?

On écrit une **procédure** pour factoriser un certain nombre d'instructions au sein d'une méta-action. En C, les procédures peuvent prendre des paramètres (qui influencent leur déroulement) mais leur type de retour est `void` (pas de retour). Par exemple : `void showValue(int value)`.

Contrairement aux procédures, les **fonctions** renvoie une valeur.

Cette valeur peut être le résultat d'un calcul (un peu comme les fonctions en mathématiques) ou une valeur de statut (est-ce que tout s'est bien passé ?). Ces fonctions prennent des paramètres en entrées et renvoient une valeur unique, dont le type est défini par le **type de retour**. Par exemple :
`int sum(int op1, int op2)`.



2 Définir une procédure ou une fonction : le prototype

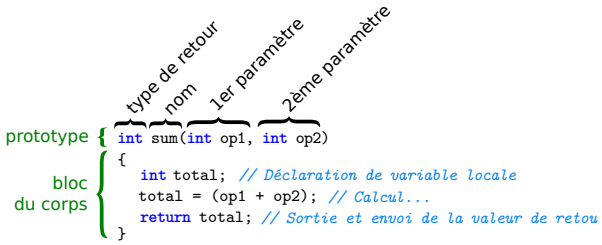
Nous écrirons toutes nos procédures et nos fonctions dans des blocs dédiés, en dehors du `main`. Les fonctions que nous avons déjà utilisées (`printf`, `scanf`) sont décrites dans une bibliothèque que nous incluons au début du programme. Pour notre part, nous décrirons nos procédures et nos fonctions « maison » directement dans notre fichier, également au début du programme, juste après les ajouts de bibliothèques & co (toutes les lignes qui commencent par `#`).

2.1 Prototype

Le bloc d'une procédure ou d'une fonction commence toujours par un prototype. Ce dernier fournit des informations précieuses sur cette dernière : son nom, la liste des types de ses paramètres (les entrées) et enfin son type de retour (sa sortie).

Nous avons déjà croisé la notion de paramètre dans le TD « *Initiation au Shell* ». Lorsque nous nous déplaçons dans l'architecture avec les commandes shell, nous envoyons le chemin du dossier en paramètre à l'utilitaire `cd` par exemple.

Voici à quoi ressemble un bloc fonction :



Dans notre exemple le prototype de la fonction s'écrit :

```
int sum(int op1, int op2)
```

Nous pouvons en déduire que la fonction prend deux nombres entiers en entrée (nommés `op1` et `op2`), et qu'elle renvoie une valeur entière (correspondant a priori à la somme des deux).

Voici le prototype d'une fonction que nous utilisons en permanence :

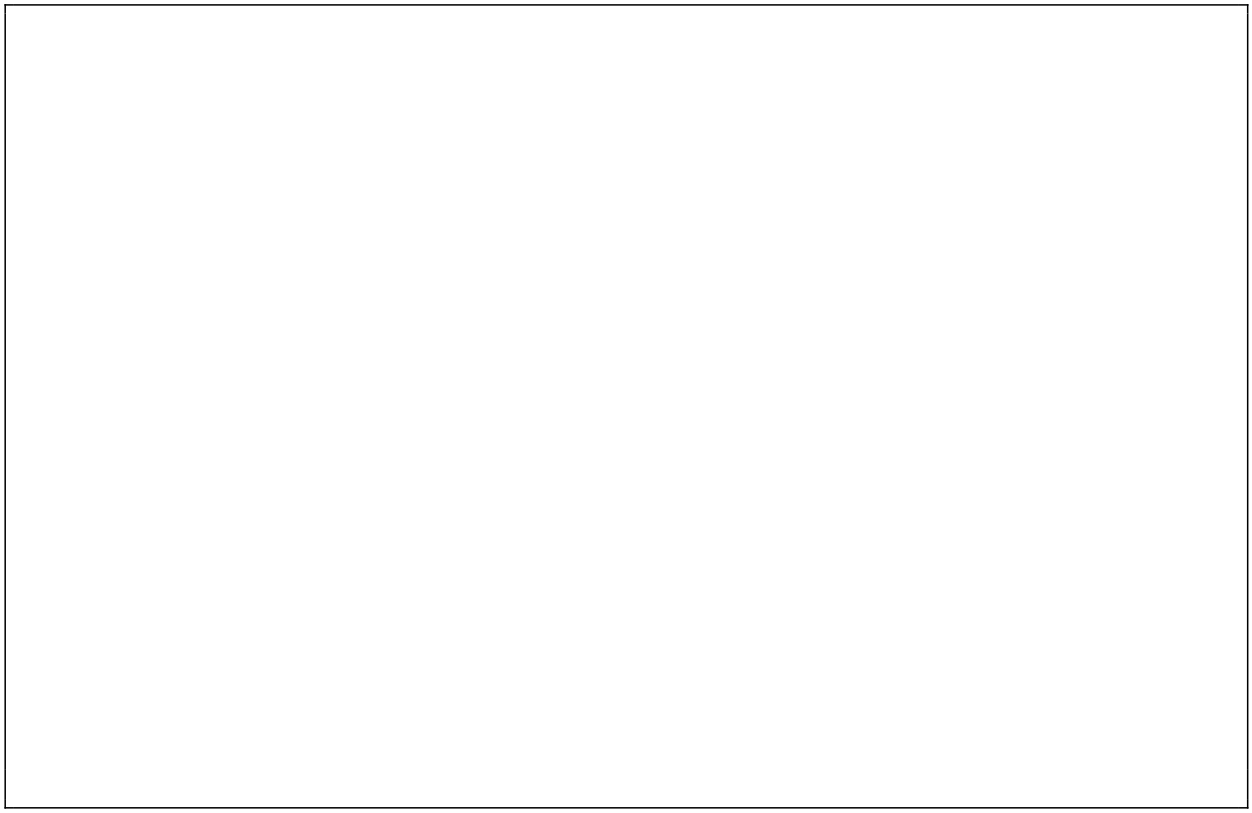
```
int main() // No parameters, return value is a status (0: OK, 1: KO)
```

Conventions de nommage

Comme pour les variables, pour des raisons conventionnelles, nous n'écrirons jamais un nom de fonction qui commence par une majuscule. Pour les mêmes raisons de lisibilité que nous avions évoquées, vous devez également donner des noms explicites à vos fonctions, ni trop longs, ni trop courts.
Par exemple : `getNonNegativeInteger`.

QUESTION 1

a) Écrire le prototype de la fonction qui calcule le maximum de deux entiers.
b) Écrire le prototype de la fonction qui calcule le maximum de trois flottants.
c) Écrire le prototype de la fonction qui teste si un entier est pair ou impair.
d) Écrire le prototype de la fonction qui teste si un entier est un multiple d'un autre.
e) Écrire le prototype de la fonction qui retourne 1 si le paramètre est négatif ou nul et 0 sinon.
f) Écrire le prototype de la fonction qui calcule la somme des n premiers entiers.



3 Exécuter l'action : l'appel

Un fonction ou une procédure décrit une méta-action, mais ce n'est pas parce que celle-ci est **décrite** qu'elle sera **exécutée**.

Une fonction est exécutée lorsqu'elle est **appelée** lors du déroulement du programme.

Nous appelons une fonction avec **son nom**, suivi des valeurs ou des variables que nous lui passons **en argument**.

Par exemple :

— Appel avec les valeurs 3 et 4 :

```
int result;
result = sum(3, 4); // Function call
```

— Appel avec des variables (ce sont les valeurs de ces variables qui sont utilisées) :

```
int result;
int var1 = 2;
int var2 = 19;
result = sum(var1, var2); // Function call
```

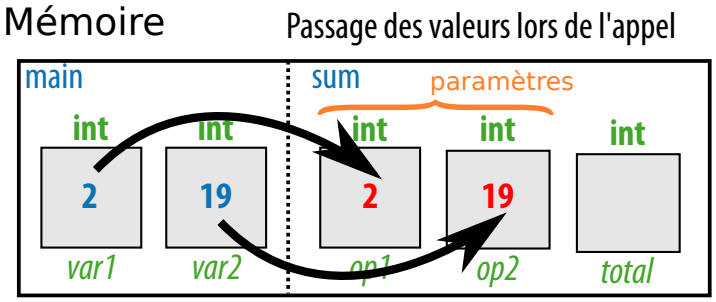
— Appels imbriqués avec des variables et la valeur retournée par une autre fonction :

```
int result;
int var1 = 2;
int var2 = 13;
result = sum(var1, sum(var2, 7)); // Function call
```

3.1 Paramètres et arguments

Paramètres et arguments :

Dans le prototype, les variables qui correspondent aux entrées de la fonction sont appelées paramètres de la fonction. Lors de l'appel, les valeurs envoyées à la fonction sont appelées arguments de la fonction. En conséquence, les arguments de la fonction servent d'initialisation aux paramètres lors du déroulement de la fonction.



QUESTION 2

Écrire les appels des prototypes que vous avez écrits à la question précédente.

3.2 Portée d'une variable

En C, les variables déclarées dans un bloc (délimité par `{` et `}`) **ne sont valables** qu'au sein de ce bloc. Nous disons que leur portée est **locale**.

Cela veut dire que les variables déclarées dans le bloc du corps des fonctions ne sont accessibles que depuis la fonction.

```c
int sum(int op1, int op2) {
   int total;
   total = op1 + op2;
   return total;
}
int main() {
   int op1 = 4; // Local declaration of op1
   total = 4; // total is not accessible from main !
   return 0;
}
```

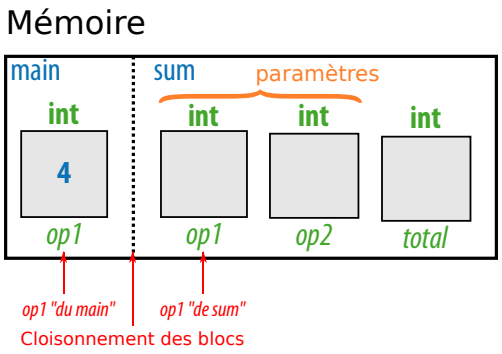


Figure 1 – ***Portée des variables.***

Attention !

La portée locale des variables implique que deux variables **de même nom** peuvent être déclarées dans des blocs différents (par ex : dans deux fonctions différentes).

3.3 Passage par valeur

En langage C, les fonctions utilisent le **passage par valeur** des arguments, c'est-à-dire que la valeur des arguments est **copiée** dans les paramètres lors de l'appel. Comme les variables du `main` ne sont pas accessibles dans la fonction, il est donc impossible (sauf à récupérer la valeur de retour) de modifier les variables du `main` depuis les fonctions.

QUESTION 3

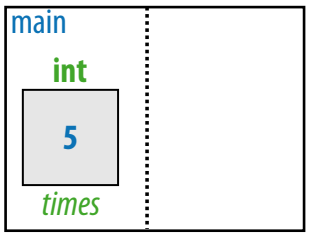
Qu'affiche le code suivant ?

```c
1  void count(int times) {
2     while(times > 0) {
3        printf("Loop: %d", times);
4        times = times - 1;
5     }
6  }
7  int main() {
8     int times = 5;
9     printf("Before: %d\n", times);
10    count(times);
11    printf("After: %d\n", times);
12    return 0;
13 }
```

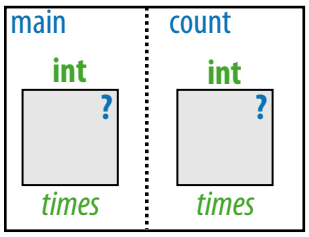
QUESTION 4

Déroulez le code en remplissant les schémas mémoire associés (comme sur la figure 1 p. 5).

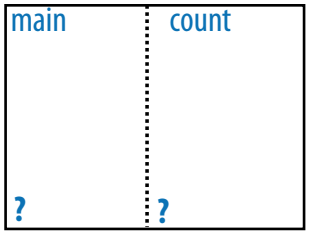
Ligne 9 :



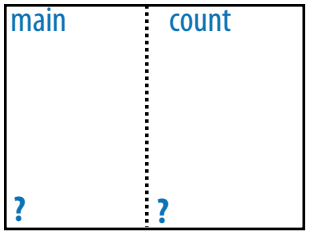
→ Ligne 1 :



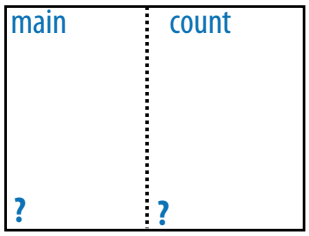
Ligne 3 :



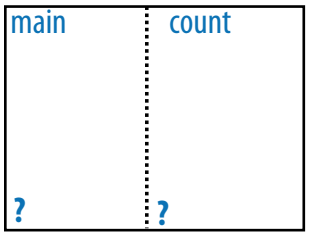
→ Ligne 3 (bis)



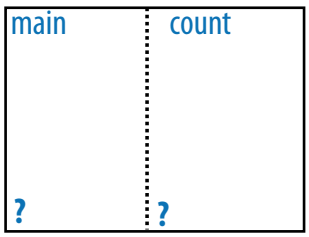
Ligne 3 (ter)



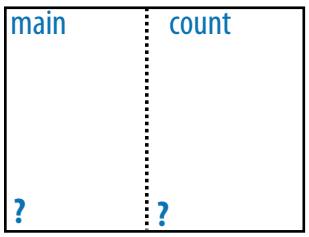
→ Ligne 3 (quart)



Ligne 3 (cinq)



→ Ligne 11



4 Last but no least

EXERCICE 1

 Créez un fichier `pyramid_fn.c` et écrivez-y le code du `Hello World!`.

EXERCICE 2

 Ajoutez la fonction `getIntegerGreaterThan` qui permet de récupérer un entier au clavier et qui réitère la demande `"Please enter a value greater than XX:\n"` (où XX correspond au seuil `threshold`) tant que ce dernier est inférieur ou égal à une borne minimale `threshold` passée en paramètre. (En cas de saisie incorrecte vous écrirez l'instruction : `exit(-1);`. Cette instruction, présente dans la bibliothèque `stdlib.h` permet de quitter le programme en produisant le code de retour passé en argument, contrairement au mot clé `return` qui ne ferait que quitter la fonction et poursuivre le programme.)

EXERCICE 3

 Ajoutez la procédure `printStar` qui prend en paramètre un entier `nbStar` et qui affiche `nbStar` caractères `'*'` sans retour à la ligne.

EXERCICE 4

 Ajoutez la procédure `printSpace` qui prend en paramètre un entier `nbSpace` et qui affiche `nbSpace` caractères `' '` sans retour à la ligne.

EXERCICE 5

 En utilisant les exercices précédents, modifiez la fonction `main` pour correspondre à l'énoncé de l'exercice *Pyramide inversée* du TP2 :

- Demandez `"Height in lines of the pyramid?"` puis récupérez un entier strictement positif.
- Si `scanf` renvoi autre chose que 1, vous afficherez `"Input error\n"` puis vous quitterez le programme avec `exit(-1)` (la fonction `exit` est décrite dans `stdlib.h`).
- Affichez la pyramide inversée correspondant à la hauteur récupérée.

```
Height in lines of the pyramid?
Please enter a value greater than 0:
-2
Please enter a value greater than 0:
5
*********
 *******
  *****
   ***
    *
```

EXERCICE 6

 Soumettez votre code à l'autocorrect (dans la bonne section). Vos fonctions doivent avoir des prototypes corrects (ils doivent être strictement identiques à ceux mentionnés dans le sujet) et faire **exactement** ce qui est attendu d'elles.