# [Math-L312] TP 4 : Types en C - la suite

Adrien Semin
adrien.semin@math.u-psud.fr

## 1 Fonctions

### 1.1 Les fonctions à paramètre modifiable

En algorithmique, l'en-tête d'une fonction spéficie le nom, le type de son résultat, et les paramètres (avec leurs types). Ces paramètres sont considérés comme les données transmises à la fonction, c'est-à-dire l'entrée de la fonction. Il peut arriver que la fonction modifie ces paramètres lors de l'exécution de son algorithme. On parle alors de paramètre modifiable : il s'agit à la fois d'une entrée et d'une sortie du programme. La déclaration d'un tel paramètre dans la liste des paramètres se fait de la manière suivante (exemple pour le type `int` dans le code suivant, téléchargeable à l'adresse `/home/doc/semin/TP/c/add.c`)

```c
#include<stdio.h>

void add(int*, int);
int main()
{
  int i=2;
  int j=3;
  printf("Avant appel de la fonction add : \n");
  printf("Pour cette partie du programme, i=%d\n",i);
  printf("Pour cette partie du programme, j=%d\n",j);
  add(&i,j);
  printf("Apres appel de la fonction add : \n");
  printf("Pour cette partie du programme, i=%d\n",i);
  printf("Pour cette partie du programme, j=%d\n",j);
  return 0;
}

void add(int *a, int b)
{
  *a = *a + b;
  b++;
}
```

Nous pouvons remarquer que lors de l'appel de la fonction `add`, la variable `i` est modifiée, alors que la variable `j` ne l'est pas. Nous pouvons également remarquer que dans la déclaration de la fonction `add`, la variable `*a` (et pas seulement `a`) est un entier. Enfin, lors de l'appel de la fonction `add`, la variable `i` est précédée du symbole `&` pour dire que son contenu va être modifié (pensez à la fonction `scanf`).

### 1.2 La fonction main

On a vu que la fonction `main` admettait le prototype [1] suivant :

```c
int main(){

}
```

En fait, même si cette écriture – introduite à l'origine pour simplifier le premier contact avec `gcc` – est parfaitement tolérée, on lui préférera désormais le prototype suivant lorsqu'on voudra communiquer avec le terminal :

1. La notion de prototype en C fait référence au type des arguments et du résultat d'une fonction.

```
int main(int argc,char* argv[]){

}
```

On a les correspondances suivantes :

- `int main` : cet entier correspond au code de retour du processus lorsqu'on le lance. Habituellement, si à l'issue de l'exécution de votre fonction `main` vous lui faites rendre le nombre `0`, c'est que vous signalez par convention au système d'exploitation que tout s'est bien passé. Toute autre valeur correspond à un signal d'erreur. À tout moment vous pouvez utiliser l'instruction suivante de la bibliothèque `<stdlib.h>` :

  ```
  exit(<int>)
  ```

  où `<int>` est la valeur que vous souhaitez renvoyer. En exécutant cette instruction, le programme s'arrête et renvoie le code d'erreur `<int>`. Sous UNIX, si on tape dans un terminal :

  `> <commande1> && <commande2>`
  la commande `<commande1>` est exécutée, puis seulement si son code d'erreur est zéro, la commande `<commande2>` est exécutée.À l'inverse,

  `> <commande1> || <commande2>`
  exécutera la commande `<commande1>`, et `<commande2>` uniquement si `<commande1>` a provoqué une erreur.
- `int argc` : cet entier correspond au nombre d'arguments qu'on a donné au programme lorsqu'on l'a lancé (depuis un terminal par exemple). Le nom du programme lui-même est considéré comme un argument, donc `argc` vaut toujours au moins 1.
- `char argv[][]` : ce paramètre (de type tableau non borné de tableaux non bornés de `char`) correspond aux arguments donnés au programme. Par exemple, si le programme s'appelle `prog` et qu'on le lance avec la commande :
  `> prog oui non "peut être"`
  alors `argc` vaudra `4` et `argv[i]`, pour `i` variant entre 0 et 3 sera un pointeur vers les 4 chaînes de caractères : `"prog"`, `"oui"`, `"non"` et `"peut être"`.

# 2 Allocations dynamiques

Observons tout d'abord par un exemple,téléchargeable à l'adresse `/home/doc/semin/TP/c/dyn_n.c` :

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(int argc, char* argv[])
5  {
6    int n;
7    printf("Donnez le nombre d'éléments de la suite de Fibonnaci à calculer:");
8    scanf("%i",&n);
9    long* Fibo = malloc(n * sizeof(long));
10   Fibo[0]=1;
11   Fibo[1]=1;
12   int i;
13   for(i=2; i<n; i++)
14     Fibo[i]=Fibo[i-1]+Fibo[i-2];
15   for(i=0; i<n; i++)
16     printf("Fibo[%i]=%li\n",i,Fibo[i]);
17   free(Fibo);
18   return 0;
19 }
```

Regardons maintenant le code :

- ligne 2 : nous incluons l'en-tête `stdlib.h`, qui permet d'utiliser les fonctions d'allocation mémoire `malloc` et `free`.
- ligne 9 : nous déclarons et initialisons le tableau `Fibo` de type `long` de manière à pouvoir stocker `n` éléments. Les éléments sont accessibles par `Fibo[0]`, `Fibo[1]`, ..., `Fibo[n-1]`. À la différence d'un tableau statique (les tableaux que nous avons vu lors du TP précédent), la taille n'a pas besoin d'être précisée lors de l'exécution.

– lignes 10-14 : nous remplissons le tableau.
– lignes 15-16 : nous affichons les éléments du tableau. Comme ceux-ci sont de type `long`, le masque `%li` (ou `%ld`) est le masque à utiliser.
– ligne 17 : nous désallouons la mémoire utilisée pour l'allocation du tableau.

La fonction `malloc` essaie d'allouer une quantité de mémoire en octets donnée par son argument, et retourne un pointeur vers la zone mémoire allouée. La fonction `sizeof` prend en argument un type et retourne la taille en octets que le type occupe. Si la fonction `malloc` n'arrive pas à allouer la mémoire, elle retourne un pointeur nul (`NULL`).

# 3 Travail à faire

1. `swap.c` ***À Rendre!*** Écrire une fonction `swap` qui prend en argument deux nombres réels $a$ et $b$, et qui échange le contenu de $a$ et de $b$. On veillera à ce que la fonction prenne le moins de temps possible. Ls tester sur deux nombres que l'on passera en argument lors de l'exécution du programme. Par exemple, l'exécution suivante

   ```
   > ./swapcompile 123.57 4523.38
   ```

   devra afficher

   ```
   Avant échange: a=123.57, b=4523.38
   Après échange: a=4523.38, b=123.57
   ```

   On utilisera la fonction `atof` de l'en-tête `stdlib.h` pour convertir une chaîne de caractères en `float`. Typiquement, nous aurons :

   ```
   float a=atof(argv[1]);
   ```

2. `sequence.c` ***À Rendre!*** Écrire un programme `sequence.c` qui prend en argument un entier $N$ (on utilisera la fonction `atoi` pour convertir une chaîne de caractères en `int`) et un nombre réel $x$, et qui calcule les $N$ premiers termes de la suite définie par :

   $$u_0 = x \quad \text{et} \quad u_{n+1} = \sqrt{0.5 + 0.5u_n}$$

   Afficher les 100 premiers termes de cette suite pour différentes valeurs de $x$. Quelle est la limite de cette suite ?

3. `min.c` ***À Rendre!*** Écrire un programme `max.c` qui prend un nombre quelconque d'arguments, qui convertit chacun de ces arguments en entier, et qui stocke chacun de ces arguments dans un tableau. On affichera ensuite le tableau et on donnera son plus petit élément.