[Math-L312] TP 6 : Fonctions en C et intégration du système d'exploitation dans les programmes

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`)

```
#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`).

Pour un tableau, la syntaxe est diffénte. Un exemple est donné par le code suivant, téléchargeable à l'adresse `/home/doc/semin/TP/c/randtab.c`

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define T 11

void fullfill(int[T][T]);
void affiche(int[T][T]);

int main()
```

```
{
  int A[T][T];
  fullfill(A);
  affiche(A);
  return 0;
}

void fullfill(int G[T][T])
{
  int i,j;
  srand(time(NULL));
  for(i=0;i<T;i++)
    for(j=0;j<T;j++)
      G[i][j] = rand() % 4;
}
void affiche(int G[T][T])
{
  int i,j;
  for(i=0;i<T;i++,printf("\n"))
    for(j=0;j<T;j++)
      printf("%d ",G[i][j]);
}
```

Nous remarquerons au passage que la syntaxe pour la déclaration

1.2 La fonction main

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

```
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 :

```
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.

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

– `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"`.

∗ ***Question orale*** ∗ Que se passe-t-il si, dans l'exemple au-dessus, on tape `peut être` au lieu de `"peut être"` ?

2 Travail à rendre

1. `length.c` ∗ ***À Rendre!*** ∗ Ecrire une fonction `length` qui prend en argument une chaîne de caractères (type `char[]`) et qui retourne la longueur de la chaîne (exemple : l'appel `length"Toto"` doit retourner 3), et afficher les longueurs des chaînes suivantes :

```
char[32] s1 = "Bonjour!\0";
char[32] s2 = "Au revoir\0";
char[32] s3 = "Chaine\0 de\0 test\0";
```

   On affichera également chaque chaîne avec `printf("%s",chaine_a_afficher)`.
   ∗ ***Question orale*** ∗ Pourquoi la longueur de la chaîne s3 est égale à 6 ?

2. `damier.c` ∗ ***À Rendre!*** ∗ Ecrire une fonction `damier` qui prend en argument un tableau T de taille 10×10, et qui remplit T de la manière suivante :

$$T(i,j) = \begin{cases} 1 & \text{si } i+j \text{ est pair} \\ 0 & \text{sinon} \end{cases}$$

   et qui affiche le tableau.
   Le tableau sera déclaré de la manière suivante

```
#define N 10;
int T[N][N]
```

   la fonction de remplissage sera prototypée de la manière suivante

```
void damier(int T[N][N])
```

   et la fonction d'affichage sera prototypée de la manière suivantew

```
void affichage(int T[N][N])
```

   ∗ ***Question orale*** ∗ Pourquoi donne-t-on la taille du tableau par une instruction `#define` ?

3. `echo.c` ∗ ***À Rendre!*** ∗ Ecrire un programme qui affiche la liste de ses paramètres. On affichera une chaîne de caractères avec `"%s"`
4. `max.c` ∗ ***À Rendre!*** ∗ Ecrire un programme qui prend un nombre quelconque d'arguments, qui convertit chacun de ces arguments en entier en utilisant l'instruction `atoi` de la bibliothèque `<stdlib.h>`, et qui stocke ces entiers dans un tableau (rappel : on utilisera la fonction `allocation_tableau`) de la bibliothèque non standard `<allocation_tableaux.h>` vue au TP4. On affichera ensuite le tableau et on donnera son plus grand élément.