TP: structures

On prend l'habitude, pour toute fonction `f` demandée dans un exercice d'écrire une procédure `void test_f()` contenant des tests unitaires. Dans ce TP presque tous les tableaux manipulés sont dynamiques.

On commence par déclarer deux structures :

```c
// tableaux avec taille
typedef struct array_with_size aws;

struct array_with_size{
  int * t;
  int size;// la taille du tableau t
};

// matrices avec dimensions
typedef struct matrix_with_size mws;

struct matrix_with_size{
  int ** mat;
  int nbl;//nb lignes
  int nbc;// nb col
};
```

Exercice 1. 1. Tableaux :

(a) Écrire la fonction `aws* new_aws(int n)` qui renvoie un pointeur sur `aws` dont le champ `t` est un tableau de taille n rempli de zéros.

(b) Écrire la fonction `void free_aws(aws* a)` qui libère `a` et tous les pointeurs de la structure vers laquelle il pointe.

(c) Écrire la fonction `void display_aws(aws* tab)` qui affiche (sans fioriture) le contenu de l'objet `aws` pointé par `a`.

```c
void test_new_aws(){
  int n = 3;
  aws * a = new_aws(n);
  a->t[1]=5; a->t[2]=6;
  display_aws(a);
  free_aws(a);
}

```

Listing 1 – Fonction de tests à appeler dans `main`

Listing 2 – Rendu sur console

```
./a.out
0 5 6
```

2. Matrices.
   (a) Écrire la fonction `mws* new_mws(int n, int m)` qui renvoie un pointeur sur `mws` dont la matrice champ `mat` représente une matrice nulle de taille $n \times m$.
   (b) Écrire la fonction `void free_mws(mws* M)` qui libère `M` et tous les pointeurs de la structure vers laquelle il pointe.
   (c) Écrire la fonction `void display_mws(mws* M)` qui affiche (sans fioriture) le contenu de l'objet `mws` pointé par `M`.

```c
void test_new_mws(){
  int n = 3, m=2;
  mws * M = new_mws(n,m);
  // A compléter (double boucle for)
  display_mws(M);
  free_mws(M);
}


```

Listing 3 – Fonction de tests à appeler dans `main`

Listing 4 – Rendu sur console

```
./a.out
  0   1
  2   3
  4   5
```

Compilation séparée

Dorénavant, nous séparons les fichiers de notre projet :
— `bibli.h` contient les en-têtes à importer de la bibliothèque standard ;
— `tp_struct.h` contient les déclarations de types et les prototypes des fonctions de tout ce TP.
— `tp_struct_alloc.c` contient le code des fonctions et procédures de tout ce TP sauf les fonctions de tests et le `main`.
— `main1.c` contient le `main` et toutes les procédures de test.

Exercice 2. Implémenter le *Produit d'Hadamard* (Voir ce Wiki)

`void hadamard(mws* m1, mws* m2, mws* res)`

de 2 matrices de même taille (ou produit composante par composante). Le résultat est mis dans une matrice `res` modifiée pour l'occasion. Une erreur d'assertion est soulevée si les deux matrices opérandes ne sont pas de la même taille.

Écrire une procédure de test de sorte à obtenir le rendu :

```
matrice 1
   1    3    2
   1    0    0
   1    2    2
matrice 2
   0    0    2
   7    5    0
   2    1    1
matrice res
   0    0    4
   7    0    0
   2    2    2
```

Hasard ; lignes de commandes

Exercice 3. Se documenter sur le hasard en C ici.
Se documenter sur les passages d'arguments en ligne de commande ici.
Notre `main` a maintenant le prototype suivant :

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

1. Écrire la fonction `void display_command_line(int argc, char* argv[])` qui prend en argument un tableau de chaînes de caractères `argv` et un nombre d'arguments `argc` et affiche le contenu de ces différentes variables.

```
24  int main(int argc, char* argv[]){
25    //test_hadamard();
26    display_command_line(argc,argv);
27    return 0;
28  }
29
```

Listing 5 – `main`

Listing 6 – Rendu sur console

```
$ ./prog 3 4 10
Program name: ./prog
Arguments number: 4
Argument supplied: 3    4       10
```

2. Écrire maintenant la fonction `mws * randmatrix(int n, int m, int k)` qui remplit une matrice $n \times m$ de coefficients choisis aléatoirement entre 0 et $k-1$.

```
34  void test_randmatrix(){
35    mws *M = randmatrix(3,4,10);
36    display_mws(M);
37    free_mws(M);
38  }
39
```

Listing 7 – Fonction de tests à placer dans `main`

Listing 8 – Rendu sur console

```
$ ./prog 3 4 10
   2    3    9    3
   9    0    4    6
   1    0    6    5
```

3. Écrire la fonction `void test_commandline2matrix(int argc, char* argv[])` qui utilise les arguments du `main` pour construire une matrice aléatoire.

```c
int main(int argc, char* argv[]){
  //test_randmatrix();
  test_commandline2matrix(argc,argv);
  return 0;
}
```

Listing 9 – `main`

Listing 10 – Rendu sur console

```
$ ./prog 3 4 10
test_commandline2matrix
   8   0   5   6
   4   8   8   8
   6   6   3   9
```

On peut s'amuser à mettre de jolies couleurs dans le terminal (voir ce code) :

```
(base) ivan@fixe:~/.../Tp_struct$ ./prog
test_commandline2matrix
Rappels
En plus du nom de fichier, il faut passer 3 arguments
  - un nombre de lignes,
  - un nombre de lignes,
  - un nombre de colonnes.
(base) ivan@fixe:~/.../Tp_struct$
```

Exercice 4. Assemblage :

1. Écrire la fonction `mws *assembleLignes(int n,aws* tab[n])` qui prend en paramètre un tableau de n `aws*` de tailles identiques et renvoie la matrice dont les lignes sont ces objets.

```c
int main(int argc, char* argv[]){
   mws * M = assembleLignes(2,t);
  test_assembleLignes();
  return 0;
}
```

Listing 11 – `main`

Listing 12 – Rendu sur console

```
$ ./prog
   5   0   0
   0   0   3
```

2. Même question pour `mws *assembleColonnes(int n,aws* tab[n])` qui assemble les tableaux en colonnes.

```c
int main(int argc, char* argv[]){
  //mws * M = assembleColonnes(2,t);
  test_assembleColonnes();
  return 0;
}
```

Listing 13 – `main`

Listing 14 – Rendu sur console

```
$ ./prog
   5   0
   0   0
   0   3
```