TP : Pointeurs et allocations

On prend l'habitude, pour toute fonction `f` demandée dans un exercice d'écrire une procédure `void bench_f()` contenant des tests unitaires.

Exercice 1. On veut constituer un tableau d'entiers, tous nuls, dont la taille est saisie au clavier. Les VLA sont interdits.

Ecrire une fonction `void creer()` qui demande à l'utilisateur d'entrer la taille du tableau, réalise les allocations correspondantes et toutes les tâches induites et affiche le contenu du tableau.

Avec

```
int main(){
  creer();
  return 0;
}
```

Une fonction d'affichage de tableaux d'entiers doit être écrite par ailleurs.

On attend le comportement suivant :

```
Entrer la taille du tableau de zéros : 5
0 0 0 0 0
```

Solution. Voici

```
void creer(){
  int n;
  printf("Entrer la  taille  du tableau de zéros  : ");
  scanf("%d",&n);
  int * p = calloc(n,sizeof(int));
  affiche(p,n);printf("\n");
  free(p);
}

```

□

Exercice 2. 1. Ecrire une fonction `int carre (int x)` qui retourne le carré de x et affiche dans le même temps le nombre de fois où cette fonction a été appelée.

On impose la fonction de tests suivante :

```
void bench_carre(){
  for (int i = 0; i<5; i++){
    int r = carre(i);
```

```
    printf("le carré de %i est %i\n", i,r);
  }
}
```

Le retour attendu est

```
appel numéro 1 de carre : le carré de 0 est 0
appel numéro 2 de carre : le carré de 1 est 1
appel numéro 3 de carre : le carré de 2 est 4
appel numéro 4 de carre : le carré de 3 est 9
appel numéro 5 de carre : le carré de 4 est 16
```

2. Dans quelles parties de la mémoire du programme vivent les variables et paramètres de `carre` ?

Solution. Il faut déclarer une variable statique `nbcall` : elle se trouve dans le *segment de données*. Le paramètre `x` est dans la pile.

```
int carre (int x){
  static int nbcall=0;// static est important !!

  nbcall+=1;
  printf("appel numéro %d de carre : ", nbcall);
  return x*x;
}

```

□

Exercice 3. Dans cet exercice tous les nombres considérés sont des entiers.

Ecrire une procédure `void filltab1(??? array, int n, int x)` qui prend en paramètre une variable `array` (vraisemblablement un pointeur, mais sur quoi ?) une taille `n` et un paramètre d'initialisation `init`. La fonction réalise l'allocation nécessaire pour que le déréférencement de `array` puisse être considéré comme un tableau de taille n dont toutes les cases ont la valeur x.

On impose la fonction de test suivante :

```
void bench_filltab1(){
    int n =5;
    int* tab = NULL;
     filltab1 (???, n, 3);// allocation de tab; à compléter
    // tab désigne maintenant le tableau {3,3,3,3,3}
    affiche_tab(5,tab);// affichage ; à écrire
     free(tab);
}
```

Le retour attendu est celui-ci :

```
3, 3, 3, 3, 3
```

Remarque. On verra à l'exercice 5, une méthode plus simple pour allouer une zone mémoire à un pointeur en passant par une fonction.

Solution. Une règle générale en C, est que pour faire des effets de bords dans une fonction, il est nécessaire de passer en argument un pointeur sur la variable à modifier.

Ici, on veut allouer une valeur à `tab`. Puisqu'en en C, le passage des paramètres s'effectue par valeur, il faut passer l'adresse de `tab` en paramètre de l'appel à `filltab1`.

Attention aux parenthèses : il faut écrire `(*array)[i]` (`*array` est un tableau dont on prend la ième case) et non `*array[i]` : `array[i]` pour $i > 1$ n'a pas de sens car `array` n'est pas un tableau de pointeur, juste un pointeur sur un autre pointeur vu comme un tableau.

```
void filltab1(int** array, int n, int init){
    *array = (int*)malloc(n * sizeof(int));
    assert((*array)!=NULL);
    for (int i=0; i<n; i++)
        (*array)[i]=init;
}

void bench_filltab1(){
    ...
    filltab1 (&tab, n, 3);// remplir avec des 3
    ...
}

```

□

Exercice 4. On veut créer un tableau de tableaux comme

```
{{1},{1,1},{1,1,1},{1,1},{1}}

```

ce qui est impossible à faire avec des tableaux statiques.

Voici la fonction de tests à appliquer une fois les fonctions désirées écrites :

```
void bench_filltab2(){
  int **p=NULL;// on veut que p désigne {{1},{1,1},{1,1,1},{1,1},{1}}
  int sizes[5]={1,2,3,2,1};

  /*--- ALLOCATION-------------------------*/
  filltab2(???,5,sizes);// que mettre à la place de '???'
  // p doit désigner {{1},{1,1},{1,1,1},{1,1},{1}}
  affiche_tabdb(5,sizes,p);// écrire cette fonction

  /*------------ LIBERATIONS---------------*/
  freetab2(???,5); // libérer p
}

```

1. Ecrire la procédure `void filltab2(???, int n, int sizes[n])` de façon à respecter l'esprit de la fonction de test donnée.
2. Ecrire la procédure `void freetab2(???, int n)` qui libère un tableau de façon à respecter l'esprit de la fonction de test donnée.

3. Ecrire la procédure d'affichage
`void affiche_tabdb(int n, int sizes[n], int *t[n])`.

Solution. La signature est `void filltab2(int*** tab, int n, int sizes[n])`.
L'appel : `filltab2(&p,5,sizes);`. La libération `filltab2(&p,5,sizes);`

```c
void affiche_tab(int n, int tab[n]){
  for (int i=0; i<n; i++){
    printf("%d, ", tab[i]);
  }
  printf("\n");
}

void affiche_tabdb(int n, int sizes[n], int *t[n]){
   for (int i = 0; i < n; i++)
     affiche_tab(sizes[i], t[i]);
}

void filltab2(int*** tab, int n, int sizes[n]){
  //créer dynamiquement un tableau de tableaux de la dimension demandée

  (*tab) = malloc(n * sizeof(int*));
  assert((*tab)!=NULL);
  for (int i=0; i<n; i++){
      (*tab)[i]=(int*)malloc(sizes[i]*sizeof(int));
      assert((*tab)[i]!=NULL);
  }

  for (int i=0; i<n; i++){// initialiser le tableau
    for(int j=0; j<sizes[i]; j++)
      (*tab)[i][j]=1;
      }
 }

//plus concis
void filltab2_bis(int*** tab, int n, int sizes[n]){
  *tab = malloc(n * sizeof(int*));
  //tab pointe sur un tableau de n ptr d'entiers
  assert((*tab)!=NULL);
  for (int i=0; i<n; i++){
    filltab1(&tab[0][i], sizes[i],1);
    // filltab1(&(*tab)[i], sizes[i],1);
  }
}


void freetab2(int*** tab, int n){
   for (int i = 0; i < 5; i++)
     {free((*tab)[i]);}
  free(*tab);
}

```

□

Exercice 5. Comme on l'a vu aux exercices 3 et 4, la méthode qui consiste à passer en argument d'une fonction `f` l'adresse d'un pointeur `p` pour allouer à `p` une zone mémoire est contraignante.

Il est beaucoup plus simple de faire les allocations dans la fonction et de retourner un pointeur sur la zone allouée. L'allocation se fait alors beaucoup plus naturellement en écrivant `p = f(x,x,...)`

Ecrire la fonction `int * add_vect(int n, int V1[n], int V2[n])` qui prend en paramètres deux tableaux de n entiers V_1 et V_2 et retourne un tableau créé dynamiquement dont les éléments sont la somme des éléments correspondants dans V_1 et V_2.

Solution. Voici

```
int * add_vect(int n, int V1[n], int V2[n]){
  int * V = malloc(sizeof(int[n]));
  for (int i=0;i<n;i++){
    V[i] = V1[i] + V2[i];
  }
  return V;
}

void test_add_vect(){
  int V1[3]={1,2,3};
  int V2[3]={-1,2,1};
  int *V = add_vect(3,V1,V2);
  /*Observer la simplicité de la méthode ci-dessus :
  pas besoin de passer l'adresse du pointeur V en paramètre. */
  printf("V1="); affiche_tab(3,V1); printf("\n");
  printf("V2="); affiche_tab(3,V2); printf("\n");
  printf("V1+V2="); affiche_tab(3,V); printf("\n");
  free(V);
}

```

□