Maîtrise de la Programmation en C – Travaux dirigés

5 Pointeurs, arithmétique des pointeurs

Exercice 20. [Arrangement mémoire des éléments d'un tableau à plusieurs dimensions]
Soit la déclaration d'un tableau

```
int b[3][5];
```

En considérant que l'allocation du tableau se fait linéairement en mémoire (les 3 "tranches" de `b` sont allouées à des adresses contiguës), donnez l'état du tableau `b` après l'exécution du code C suivant :

```
int *a = *b, i;

for (i=0 ; i<15 ; *a++ = i++)
    ;
**b = 15;           **(b+1) = 16;         *(b[0]+1) = 17;
*(*b+8) = 18;       *(b[1]+2) = 19;       *(*(b+1)+5) = 20;
*(b[2]+3) = 21;     *(*(b+2)+2) = 22;
```

Exercice 21. [Adresses mémoires des types construits]
Soit le programme `tab_duo_trio.c` suivant

```
#include <stdlib.h>
#include <stdio.h>

struct duo_s{
  int x;
  int y;
};

union trio_u{
  int n;
  char c;
  float x;
};

int main(){
  int tab[10];
  struct duo_s duo;
  union trio_u trio;

  printf("sizeof(int) : %lu\n", sizeof(int));
  printf("tab :\t%p\n", tab);
  printf("&duo :\t%p\n", &duo);
  printf("&trio :\t%p\n", &trio);

  exit(EXIT_SUCCESS); /* i.e. return 0; */
}
```

dont une exécution produit :

```
$ ./tab_duo_trio
sizeof(int) : 4
tab :   0x7ffeebe6a6d0
&duo :  0x7ffeebe6a6c8
&trio : 0x7ffeebe6a6c0
```

1. Quelle est l'adresse du champ `duo.x` ? `duo.y` ?
2. Quelle est l'adresse du champ `trio.n` ? `trio.c` ? `trio.x` ?
3. Quelles sont les valeurs des expressions `&trio.n + 1` et `&trio.c + 1` ?
4. Quelle est l'adresse de l'élément `tab[5]` du tableau ? De manière générale, quelle est l'adresse de l'élément `tab[i]` ?
5. Soit la déclaration et initialisation :

```
int *ptr;
ptr = tab;
```

En utilisant uniquement la variable `ptr`, et l'arithmétique des pointeurs, proposez une boucle qui va écrire la valeur `0` dans toutes les cases du tableau.

Exercice 22. [Pointeurs et chaînes de caractères]

1. Écrire une fonction `char *strend(char *str)` qui renvoie l'adresse du zéro terminal de la chaîne `str`.
2. (Pourquoi le prototype de cette fonction ne peut-il être `char *strend(const char *str)` ?)
3. En utilisant la fonction `strend()`, proposez une fonction qui renvoie le nombre de caractères d'une chaîne donnée - le caractère `'\0'` final non compris :

   ```
   int mstrlen(const char *str);
   ```

   Pour information. Cette fonction `mstrlen()` est similaire à la fonction `strlen()` fournie par la bibliothèque standard `string.h`.
4. La bibliothèque `string.h` fournie également les fonctions suivantes :

   ```
   /* recopie le contenu de src dans dest renvoie dest */
   char *strcpy(char *dest, const char *src);

   /* recopie src à la fin de dest (concat) renvoie dest */
   char *strcat(char *dest, const char *src);
   ```

   Proposez des fonctions `mstrcpy()` et `mstrcat()`, réécritures de ces fonctions.

6 Allocation dynamique & structures autoréférentes

Exercice 23. [Allocation dynamique] Soit le programme *incorrect* suivant:

```
/* Renvoie un tableau à valeurs:
   {0, 1, 2, ..., 9} */
int *creer_sequence_10(){
  int t[10];
  int i;
  for (i=0; i<10; i++)
    t[i] = i;
  return t;
}
```

```
int main(void){
  int *tab;
  int i;
  tab = creer_sequence_10();
  for (i=0; i<10; i++)
    printf("%d\n", tab[i]);
}
```

1. Pourquoi ce programme est-il incorrect ?
2. Proposez une correction de ce programme utilisant `malloc()` et `free()`.

Exercice 24. [Allocation et libération de structures.]
Soit les déclarations suivantes:

```
struct point_s{
  int p_x;
  int p_y;
};
```

```
struct point_s *new_point(int x, int y);
void free_point(struct point_s *p);
```

La fonction `new_point()` alloue et renvoie l'adresse d'une nouvelle structure de type `struct point_s`, dont les champs sont initialisés avec les valeurs des paramètres `x` et `y`. La fonction `free_point()` se charge de libérer la mémoire allouée correspondante à `p`.

1. Donnez une définition de la fonction `new_point()`.

2. Donnez une définition de la fonction `free_point()`.

Soient les déclarations suivantes :

```
struct etudiant_s {
    char *etu_nom;
    char *etu_prenom;
    int   etu_no_etudiant;
};
struct etudiant_s *new_etudiant(const char *n, const char *p, int no);
void free_etudiant(struct etudiant_s *etu);
```

La fonction `new_etudiant()` alloue et renvoie l'adresse d'une nouvelle structure de type `struct etudiant_s`. Les champs `etu_nom` et `etu_prenom` sont initialisés à l'aide d'une *copie* des valeurs des paramètres `nom` et `prenom`.
La fonction `free_etudiant()` se charge de libérer la mémoire allouée correspondante à `e`.

3. Donnez une définition de la fonction `new_etudiant()`.
4. Donnez une définition de la fonction `free_etudiant()`.
5. Proposez une modification du type `struct etudiant_s` et des fonctions d'allocation et désallocation associées pour associer à un étudiant un certain nombre de notes (valeurs de type `int` par exemple).

Exercice 25. [Noms temporaires] Donnez la définition d'une fonction qui retourne à chaque invocation une chaîne de caractères différente, par exemple en vue de l'utiliser comme un nom de fichier temporaire.

Exercice 26. [Librairie d'arbres binaires de flottants] Considérons le fichier d'entête `tree.h` suivant:

```
#ifndef TREE_H
#define TREE_H
typedef struct tree_s* tree_t;

void init(tree_t *); /* initialise un arbre */
tree_t new_tree(float); /* cree une feuille */
int is_empty(tree_t); /* vraie ssi l'arbre est vide */
int is_leaf(tree_t); /* vraie ssi l'arbre est une feuille */
tree_t fils_gauche(tree_t); /* renvoie le fils gauche de l'arbre */
tree_t fils_droit(tree_t); /* renvoie le fils droit de l'arbre */
float value_root(tree_t); /* renvoie la valeur de la racine de l'arbre */

void destroy(tree_t); /* désalloue tout l'arbre */

int cmp_tree(tree_t,tree_t); /* vrai ssi les deux arbres sont identiques */
int print_tree(tree_t); /* affiche les elements contenus dans l'arbre
de gauche a droite. Renvoie le nombre de noeuds de l'arbre  */

#endif /*  TREE_H */
```

Donnez le code du fichier `tree.c` associé (i.e. qu'il vous faut définir `struct tree_s` et l'ensemble des fonctions déclarées ici).

7 Pointeurs génériques, pointeurs de fonctions

Exercice 27. [Comparaison générique]L'objet de l'exercice est de proposer une fonction `mmemcmp()` qui compare deux zones mémoire octet par octet.

1. Donnez un prototype possible pour cette fonction.
2. Donnez une définition de cette fonction. La fonction renverra une valeur nulle si et seulement si les deux zones sont égales.

Pour information. La bibliothèque `string.h` propose une fonction `memcmp()` qui réalise une telle comparaison.

Exercice 28. [Appliquer une fonction à un tableau d'entiers]
Soit `func_t` le type des fonctions à un paramètre entier qui renvoient une valeur entière :

```
typedef int (func_t)(int);
```

L'objet de la fonction `map_int()` que nous allons écrire est d'appliquer une fonction de type `func_t` à chacun des éléments d'un tableau d'entiers. Le résultat sera produit dans autre tableau d'entiers.

1. Étant données cette fonction `map_int()` qui produit dans le tableau `res` l'application de la fonction `f` à chacun des `size` éléments du tableau `arg`:

   ```
   void map_int(const int arg[], int res[], unsigned int size, func_t *f);
   ```

   et les définitions suivantes:

   ```
   #define MAX 128
   int t[MAX], t2[MAX];
   int carre(int n){
      return n*n;
   }
   ```

   En supposant le tableau `t` initialisé, donnez le code d'un appel à `map_int()` pour produire dans `t2` les carrés des valeurs de `t`.

2. Proposez une définition de `map_int()`.

Exercice 29. [Map générique] La fonction `map_gen()` est une version générique de la fonction `map_int()`, capable de traiter des tableaux de type quelconque.
Cette fonction `map_gen()` utilise une fonction de type `funcgen_t` pour appliquer une opération sur chaque élément d'un tableau :

```
typedef void (funcgen_t)(const void*, void*);
```

Le principe d'une telle fonction est de considérer son premier paramètre comme une référence vers un argument, et renvoyer son résultat via le second paramètre.
Le prototype de `map_gen()` est le suivant :

```
void map_gen(const void *arg, void *res, unsigned int nbelem,
             unsigned int elemsize, funcgen_t *f);
```

Les paramètres `arg` et `res` sont des tableaux de `nbelem` élements, chaque éléments occupe `elemsize` octets. La fonction produit la valeur `f(arg[i])` dans l'élément `i` du tableau `res`.

1. Soient les définitions suivantes :

   ```
   #define MAX 128
   int t[MAX], t2[MAX];
   void carre_int_v1(const void *src, void *dst){
     int* src_int = (int*) src;
     int* dst_int = (int*) dst;
     *dst_int = *src_int * *src_int;
   }
   ```

En supposant le tableau `t` initialisé, donnez le code d'un appel à `map_gen()` pour produire dans `t2` les carrés des valeurs de `t`.

2. Faites de même en utilisant maintenant la fonction suivante :

```
void carre_int_v2(const int *src, int *dst){
  *dst = *src * *src;
}
```

3. Donnez la définition de fonctions et le code permettant de produire un tableaux des carrés de valeur de type `float`.

4. Proposez une définition de la fonction `map_gen()`.

8 Implémentation (basique) de notre propre `malloc`

Dans ce TD, nous allons implémenter de deux manières différentes un mécanisme d'affectation dynamique de mémoire. Nous supposerons disposer d'une zone mémoire:

```
#define NULL ((void *) 0)
#define BUFFERSIZE 65536
static char buffer[BUFFERSIZE];
```

Dans les deux exercices, nous souhaiterons coder les deux fonctions suivantes:

```
/* retourne un pointeur generique sur un espace memoire de taille n
   Retourne NULL si cete quantite n'est pas disponible.
   Les zones memoires ne doivent a aucune moment se recouvrir */
void* monmalloc(unsigned int n);

/* libere l'espace memoire associe au pointeur p */
void monfree(void *p);
```

Aucune autre fonction ou partie du code n'a à accéder au tableau `buffer`: ce dernier n'est manipulé que par l'intermédiaire des fonctions `monmalloc` et `monfree`.

Exercice 30. [Implantation rudimentaire du type pile (LIFO)]
Pour commencer, on propose une implantation rudimentaire du type pile: les espaces alloués s'empilent dans le tableau et la désallocation consiste en un dépilement. Ainsi, l'utilisateur doit veiller à ce que les appels de la fonction `monfree` se fassent dans l'ordre inverse exacte de ceux de la fonction `monmalloc`.
Ce type d'implantation simple se base sur les définitions :

```
static char *nextfreebyte = buffer;
```

Le pointeur `nextfreebyte` pointe sur le premier octet non alloué du tableau `buffer`.

1. Pourquoi le pointeur `nextfreebyte` est-il de classe d'allocation `static` ?

2. Implantez les fonctions `monmalloc` et `monfree` basées sur le principe ci-dessus.

Remarque. Ce principe ne permet pas une gestion simple de la mémoire dynamique notamment par la contrainte sur les appels. L'exercice suivant propose une implantation plus souple des fonctions `monmalloc` et `monfree`.

Exercice 31. [Implantation à base de listes doublement chaînées]
Nous allons utiliser une structure de donnée auxiliaire permettant de stocker l'ensemble des blocs définis dans l'espace mémoire `buffer`. Chaque bloc commence par une entête définie par une structure de type `freelist_t` constituée par les champs:

- `isfree` pour savoir si le bloc est libre ou pas ;
- `size` pour la taille en octets de cette zone ;
- `next` pour un pointeur sur l'entête du bloc suivante de la liste ;
- `prev` pour un pointeur sur l'entête du bloc précédent de la liste.

Ainsi, dans un bloc libre de taille totale n octets, ne sont réellement disponibles que $n - t$ octets avec :

```
t = sizeof(freelist_t) = 2*sizeof(unsigned int) + 2*sizeof(freelist_t *);
```

On suppose exister une variable globale permettant de pointer sur la première entête disponible:

```
static freelist_t *Head = NULL;
```

De plus, une entête dont le champs `next` (resp. `prev`) pointe sur `NULL` est la dernière (resp. première) de la liste. La liste des blocs est vide si `Head` pointe sur `NULL`.

1. Donnez la déclaration de la structure de l'entête. Définissez le type `freelist_t` correspondant.
2. Que fait la fonction définie ci-dessous ?

```
1 static void AllocatorInit(void){
2   Head = (freelist_t *) buffer;
3   Head->isfree = 1;
4   Head->size = BUFFERSIZE;
5   Head->next = NULL;
6   Head->prev = NULL;
7   return;
8 }
```

   On affecte `BUFFERSIZE` en ligne 4. Pourquoi pas `BUFFERSIZE-sizeof(freelist_t)` ?

3. Donnez la définition de la fonction `monmalloc`. L'idée est la suivante (nous notons n la taille demandée, et t la taille de la structure):
   (a) Pointer le premier bloc libre de taille suffisante (i.e. supérieure à $t + n$) dans la liste des blocs. Ce bloc est le bloc courant. Retourner NULL si ce n'est pas possible;
   (b) Si l'espace libre du bloc courant est strictement supérieur à $2t + n$, le bloc courant doit être scindé en deux blocs. Le premier est le nouveau bloc courant et sert à satisfaire la requête. Le second est ajouté à la liste des blocs juste derrière le bloc courant.
   (c) Retourner un pointeur sur l'espace libre positionné juste après l'entête du bloc courant.

   Au préalable, appelez la fonction `AllocatorInit` si nécessaire.
4. Donnez la définition de la fonction `monfree` qui prend en entrée un pointeur et libère le bloc correspondant. Cette fonction s'assurera qu'il n'y a pas deux blocs consécutifs libres dans la liste de blocs.
5. Comment s'assurer que le pointeur donné à `monfree` pointe bien après une entête ?
6. Que risque-t-il de se passer si un bloc de n octets de mémoire est alloué et que l'on manipule ce bloc sans précaution comme dans l'allocation de D suivante :

```
unsigned int i, n = 48;
char *D = (char *) monmalloc(n);
for(i=0;i<65000;i++)
  D[i] = 1;
```