Feuille de révisions n°5 - Programmation en C : manipulation de pointeurs

1 Pointeurs et mémoire

R. 5-1 Décrire précisément l'évolution de la mémoire (tas et pile) au cours de l'exécution des deux programmes ci-dessous, en déduire quel affichage qu'ils produiront.

```c
int a, b;
int* aa1 = &a;
int* bb1 = &b;

a = b;
b = b + 2;
int* aa2 = &a;
int* bb2 = &b;

printf("a==b : %d\n",a==b);
printf("aa1==bb1 : %d\n",aa1==bb1);
printf("aa1==aa2 : %d\n",aa1==aa2);
printf("bb1==bb2 : %d\n",bb1==bb2);
```

```c
char* s = "abc";
char** pp1 = &s;
char* p1 = s;
char c1 = s[0];
char* ac1 = &c1;

s = "baba";
char** pp2 = &s;
char* p2 = s;
char c2 = s[1];
char* ac2 = &c2;

printf("p1==p2 : %d\n",p1==p2);
printf("pp1==pp2 : %d\n",pp1==pp2);
printf("c1==c2 : %d\n",c1==c2);
printf("ac1==ac2 : %d\n",ac1==ac2);
```

R. 5-2 On suppose que les déclarations ci-contre sont faites dans le `main` d'un programme. Comment sont placées ces variables dans la pile ? En particulier, si l'écart entre les adresses de `a1` et `a2` correspond à 4 octets, quel écart y a-t-il entre les adresses des autres variables ?
Les adresses de `tab1[0]` et `tab2[0]` sont-elles comparables ?
Résumer le comportement lors des différentes déclarations de tableaux.

```c
int a1;
int a2;
int tab1[3];
int a3;
int* tab2=malloc(sizeof(int)*3);
int a4;
int* tab3;
int a5;
```

R. 5-3 Décrire l'évolution de la pile lors de l'exécution du programme ci-dessous.

```c
int f(int a, int b){
  // retourne ...
  int c = a + b;
  int d = a - b;
  return c + d;
}
```

```c
int g(int* a, int*b){
  // retourne ...
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return tmp;
}
```

```c
int main(){
  int aa = 7;
  int bb = 12;
  int cc = f(bb,aa);
  cc = g(&aa,&bb);
  return 0;
}
```

R. 5-4 On considère le code ci-après qui définit deux fonctions jumelles : l'une codée avec un `while` et l'autre est son analogue avec une fonction auxiliaire récursive. Décrire l'évolution de la pile lors de l'exécution de l'affectation `int opp_w = oppose_while(-2)`. Même question pour l'affectation `int opp_r = oppose_rec(-2)`. Synthétiser la différence de comportement entre les boucles `while` et les fonctions récursives. Le même comportement est-il observé en OCaml ?

```c
int oppose_while(int n){
  int res = 0;
  int nn = n;
  while(nn > 0){
    nn = nn - 1;
    res = res - 1;
  }
  while(nn < 0){
    nn = nn + 1;
    res = res + 1;
  }
  return res;
}
```

```c
int aux (int n, int res){
  if (n == 0){ return res;}
  else{
    if(n > 0){
      return aux(n-1, res-1);
    }
    else{
      return aux(n+1, res+1);
    }
  }
}

int oppose_rec(int n){
  return aux(n,0);
}
```

R. 5-5 Pour les deux définitions de type suivantes (`poly1` et `poly2`), définir une fonction qui crée une occurrence du polynôme $aX^2 + bX + c$ à partir des coefficients a, b et c.

```c
struct poly_s {
  int n;
  float* tab; //de taille n
};
typedef struct poly_s poly1;
typedef struct poly_s* poly2;
```

R. 5-6 On considère le code suivant. Décrire l'évolution de la mémoire au cours de son exécution, en déduire le comportement observé (en particulier l'affichage). Même question si on on remplace la ligne 3 par la ligne 4 d'une part, et si on décommente la ligne 16 d'autre part.

```c
char* c1 = "mpsi";
char* c2 = "mpsi";
char c3[5]="mpsi";
//char c3[4]="mpsi";
char c4[5];
strcpy(c4,c1);//de string.h
char* c5 = (char*)
 ↪  malloc(5*sizeof(char));
strcpy(c5,c1);//de string.h
```

```c
printf("Adresses enregistrées par\n");
printf("c1 : %p / c2: %p\n",c1,c2);
printf("c3 : %p / c4: %p\n",c3,c4);
printf("c5 : %p\n\n",c5);

printf("Chaîne pointées par\n");
printf("c1:%s/c2:%s\n",c1,c2);
//c2[2]='2';
printf("c3:%s/c4:%s/c5:%s\n",c3,c4,c5);
c3[2]='2';
printf("c3:%s/c4:%s/c5:%s\n",c3,c4,c5);
```

2 Différents types de listes chaînées

R. 5-7 Définir une structure de cellule pour les listes **simplement** chaînées d'éléments constitués d'un entier seul. Définir ensuite un nouveau type pour de telles listes.

R. 5-8 Définir une structure de cellule pour les listes **doublement** chaînées d'éléments constitués d'un entier seul. Définir ensuite un nouveau type pour de telles listes, qui doivent pouvoir être parcourues depuis la tête ou depuis la queue.

R. 5-9 Définir une structure de cellule pour les listes **simplement** chaînées d'éléments constitués d'un entier **et** d'une chaîne de caractères. Définir ensuite un nouveau type pour les listes chaînées de tels éléments.

R. 5-10 Définir une structure de cellule pour les listes **simplement** chaînées d'éléments constitués d'un entier et d'un tableau. On peut imaginer par exemple qu'un élément est un sommet muni du tableau de ses voisins dans un graphe. Définir ensuite un nouveau type pour les listes chaînées de tels éléments.

3 Listes d'entiers simplement chaînées

R. 5-11 Définir une fonction qui crée une liste chaînée d'entiers réduite à une cellule. L'entier à enregistrer dans cette cellule sera pris en argument.

R. 5-12 Définir une fonction qui affiche une liste chaînée d'entiers. Cette fonction doit pouvoir s'exécuter sur une liste vide, et fournir alors un affichage adéquat.

R. 5-13 Définir une fonction qui prend en argument un entier $n \in \mathbb{N}$ et qui renvoie une liste chaînée contenant les n premiers entiers naturels. L'itération de cette fonction devra être réalisée avec une boucle `for`.

R. 5-14 Définir une fonction qui libère l'espace mémoire alloué pour une liste chaînée d'entiers.

R. 5-15 Définir une fonction qui crée une liste chaînée d'entiers à partir d'un tableau d'entiers.

R. 5-16 Définir une fonction qui crée un tableau d'entiers à partir d'une liste chaînée d'entiers. L'itération de cette fonction devra être réalisée avec une boucle `for`.

R. 5-17 Définir une fonction qui ajoute un à un les éléments d'une liste `l1` dans une autre liste `l2`. Les listes `l1` et `l2` ne doivent pas être dégradées par cette procédure. On renverra la nouvelle liste, dans laquelle les éléments de `l1` apparaissent avant ceux de `l2`, en ordre inverse. Par exemple pour `l1` la liste $1/2/3$ et `l2` la liste $4/5/6$ le résultat attendu est la liste $3/2/1/4/5/6$.

R. 5-18 Définir une fonction qui calcule le miroir d'une liste `l1` sans la dégrader.

R. 5-19 Définir une fonction qui crée une copie d'une liste `l1` sans la dégrader.

R. 5-20 Définir une fonction qui prend en argument une liste et un entier naturel k inférieur à la taille de la liste, et qui supprime les éléments de cette liste au delà des k premiers. On veillera à désallouer l'espace mémoire des cellules supprimées de la liste.

R. 5-21 Définir une fonction qui prend en argument une liste et un entier naturel k, et qui supprime les éléments de cette liste au delà des k premiers, s'il y en a. On veillera à désallouer l'espace mémoire des cellules supprimées de la liste.

R. 5-22 Définir une fonction qui prend en argument une liste passée par référence et la transforme en place en la liste miroir. Cette fonction ne devra pas allouer d'espace mémoire supplémentaire, mais pourra en revanche faire un usage habile de la pile d'appels.

4 Types abstraits implémentés par listes chaînées

R. 5-23 Définir un type puis des fonctions permettant de réaliser les opérations élémentaires d'une PILE dans le cadre d'une implémentation par liste chaînée. Préciser la complexité de chaque opération.

R. 5-24 Définir un type puis des fonctions permettant de réaliser les opérations élémentaires d'une FILE dans le cadre d'une implémentation par liste chaînée. Préciser la complexité de chaque opération.

R. 5-25 Définir un type puis des fonctions permettant de réaliser les opérations élémentaires d'une LISTE dans le cadre d'une implémentation par liste chaînée. Préciser la complexité de chaque opération.