
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.

```
1 int a, b;
2 int* aa1 = &a;
3 int* bb1 = &b;
4
5 a = b;
6 b = b + 2;
7 int* aa2 = &a;
8 int* bb2 = &b;
9
10 printf("a==b : %d\n", a==b);
11 printf("aa1==bb1 : %d\n", aa1==bb1);
12 printf("aa1==aa2 : %d\n", aa1==aa2);
13 printf("bb1==bb2 : %d\n", bb1==bb2);
14
15 char* s = "abc";
16 char** pp1 = &s;
17 char* p1 = s;
18 char c1 = s[0];
19 char* ac1 = &c1;
20
21 s = "baba";
22 char** pp2 = &s;
23 char* p2 = s;
24 char c2 = s[1];
25 char* ac2 = &c2;
26
27 printf("p1==p2 : %d\n", p1==p2);
28 printf("pp1==pp2 : %d\n", pp1==pp2);
29 printf("c1==c2 : %d\n", c1==c2);
30 printf("ac1==ac2 : %d\n", ac1==ac2);
```

Solution

affichages obtenus :

1	a == b : 0
2	aa1 == bb1 : 0
3	aa1 == aa2 : 1
4	bb1 == bb2 : 1

1	p1 == p2 : 0
2	pp1 == pp2 : 1
3	c1 == c2 : 1
4	ac1 == ac2 : 0

À retenir Lorsqu'on modifie une variable on change sa valeur, pas l'endroit où elle est enregistrée de sorte que son adresse reste inchangée.

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.

```
1 int a1;
2 int a2;
3 int tab1[3];
4 int a3;
5 int* tab2=malloc(sizeof(int)*3);
6 int a4;
7 int* tab3;
8 int a5;
```

Solution

On donne ci-contre les adresses affichées (avec `%d`) pour chaque variable lors d'une exécution. Les valeurs particulières ne sont pas pertinentes, mais les écarts entre les adresses sont informatifs.

- Un `int` occupe 4 octets, donc l'écart entre `a1` et `a2` est de 4.
- Trois `int` occupent 12 octets, donc l'écart entre `a2` et `tab1` est de 12.
- Un `int` occupe 4 octets, donc l'écart entre `tab1[0]` et `a3` est de 4.
- Un pointeur occupe 8 octets, donc l'écart entre `a3` et `tab2` est de 8.
- Un `int` occupe 4 octets, donc l'écart entre `tab2` et `a4` est de 4.
- Un pointeur occupe 8 octets, donc l'écart entre `a4` et `tab3` est de 8.
- Un `int` occupe 4 octets, donc l'écart entre `tab3` et `a5` est de 4.

```

1 | a1 : 1210010220
2 | a2 : 1210010216
3 | a3 : 1210010200
4 | a4 : 1210010188
5 | a5 : 1210010172
6 |
7 | tab1[0]:1210010204
8 | tab1[1]:1210010208
9 | tab1[2]:1210010212
10 | tab2:1210010192
11 | tab3:1210010176

```

Les adresses de `tab1[0]` et `tab2[0]` ne sont pas comparables. L'adresse de `tab1[0]` est une adresse dans la pile car `tab1` est un tableau statique, tandis que celle de `tab2[0]` est une adresse du tas, car ce tableau a été alloué dynamiquement. En particulier l'adresse de `tab2[0]` n'est pas corellée à l'adresse de `tab2`.

```

1 | tab1[0]:1210010204
2 | tab2[0]:857621088

```

À retenir Les variables sont placées dans la pile dans l'ordre où elles sont déclarées. Les variables de type `int` occupent 4 octets. Les tableaux **statiques**, *i.e.* déclarés avec un taille littérale, sont stockés dans la pile. Ainsi déclarer un tableau statique de type `t` et de taille `n` est équivalent à `n` déclarations de variables de type `t`. Les pointeurs (vers n'importe quoi) occupent 8 octets, ainsi les tableaux (non statiques) occupent 8 octets dans la pile peu importe leur taille. Cette taille est fixée lorsque l'instruction `malloc` est exécutée, on parle d'allocation de mémoire **dynamique**, et n'impacte que l'espace alors réservé dans le tas.

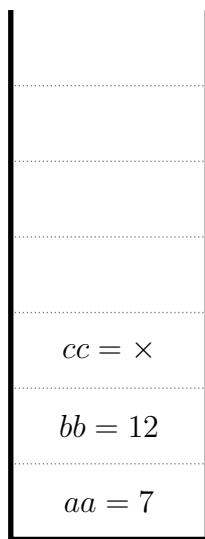
R. 5-3 Décrire l'évolution de la pile lors de l'exécution du programme ci-dessous.

```

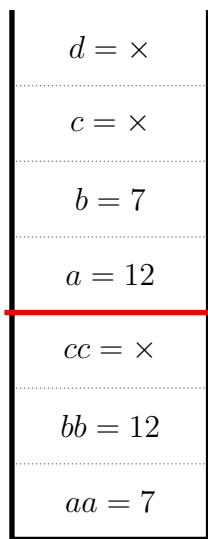
1 | int f(int a, int b){
2 |   // retourne ...
3 |   int c = a + b;
4 |   int d = a - b;
5 |   return c + d;
6 |
8 | int g(int*a, int*b){
9 |   // retourne ...
10 |  int tmp = *a;
11 |  *a = *b;
12 |  *b = tmp;
13 |  return tmp;
14 |
16 | int main(){
17 |   int aa = 7;
18 |   int bb = 12;
19 |   int cc = f(bb,aa);
20 |   cc = g(&aa,&bb);
21 |
22 |

```

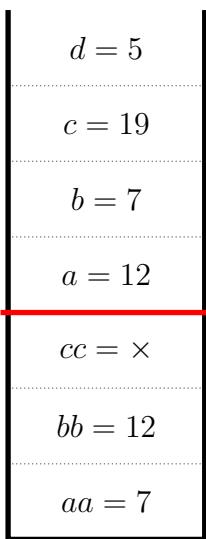
Solution



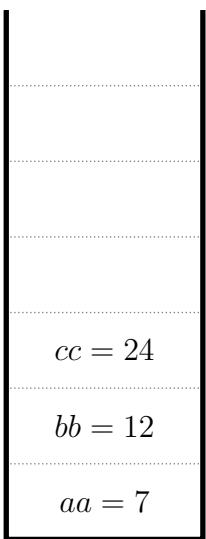
1.21



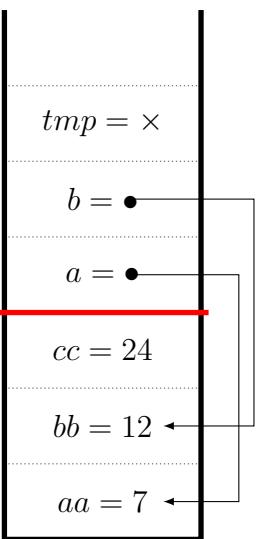
1.4



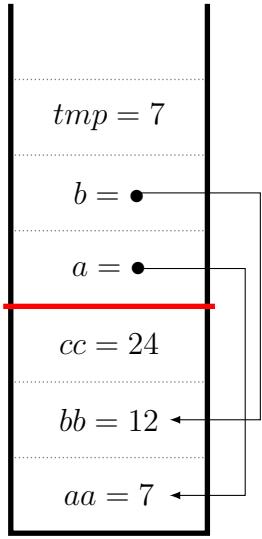
1.7



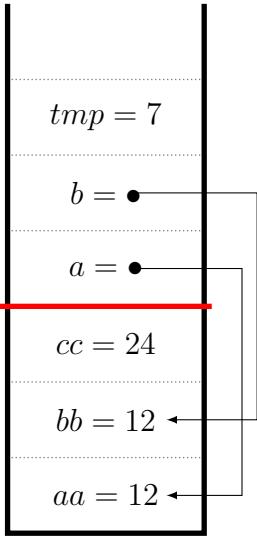
1.22



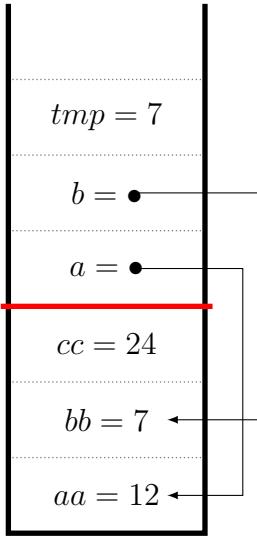
1.11



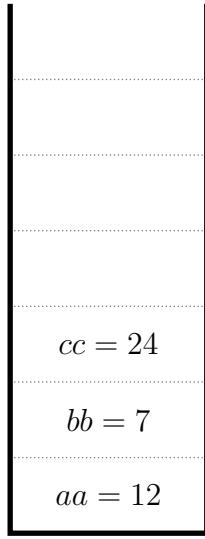
1.13



1.14



1.15



1.23

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 ?

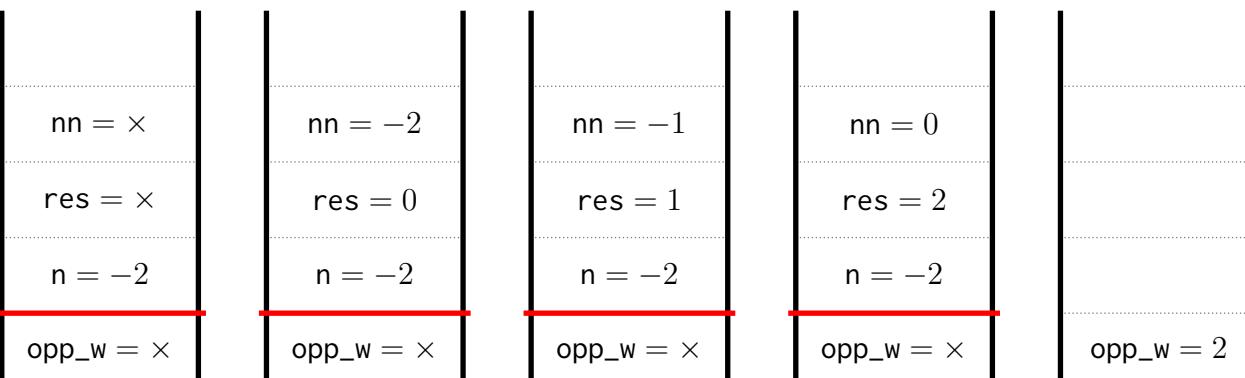
```

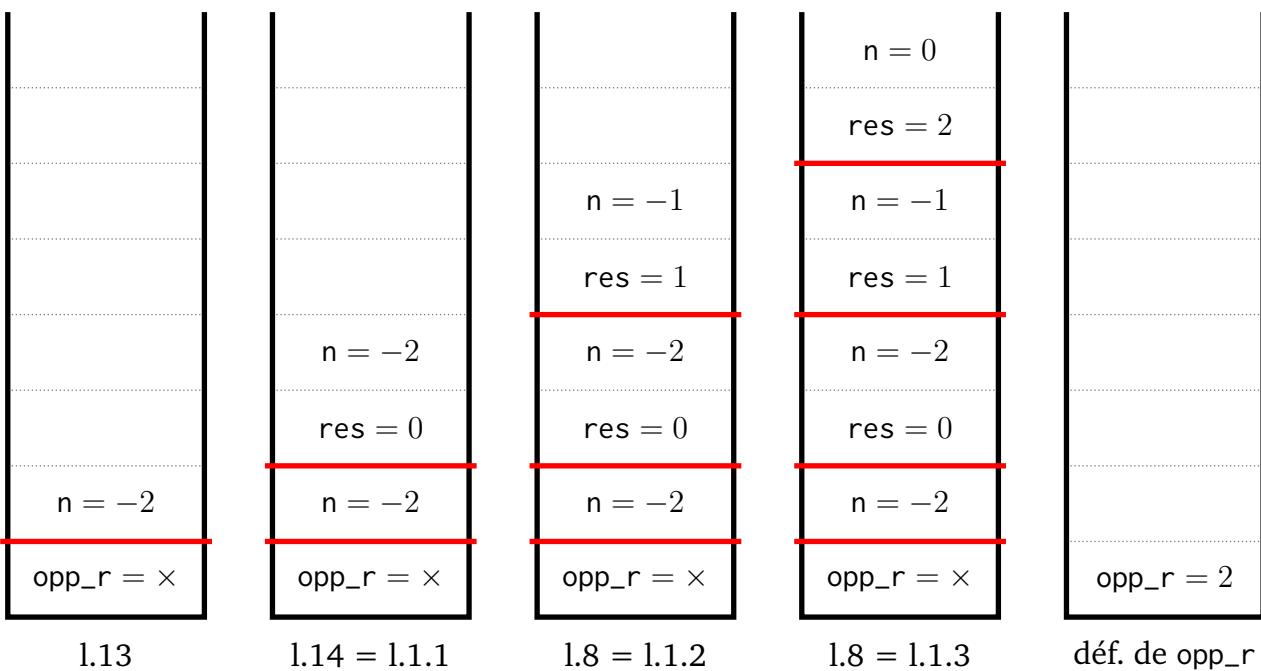
1 int oppose_while(int n){
2     int res = 0;
3     int nn = n;
4     while(nn > 0){
5         nn = nn - 1;
6         res = res - 1;
7     }
8     while(nn < 0){
9         nn = nn + 1;
10        res = res + 1;
11    }
12    return res;
13 }
```

```

1 int aux (int n, int res){
2     if (n == 0){ return res; }
3     else{
4         if(n > 0){
5             return aux(n-1, res-1);
6         }
7         else{
8             return aux(n+1, res+1);
9         }
10    }
11 }
12
13 int oppose_rec(int n){
14     return aux(n,0);
15 }
```

Solution





À retenir Lorsqu'une fonction est appelée, dès le début est réservé sur la pile l'espace nécessaire pour stocker chacun des arguments et chaque des variables locales à la fonction. Ainsi, à chaque appel récursif de l'espace est réservé sur la pile pour copier les valeurs des arguments dans de nouvelles cases, tandis qu'avec une boucle while, les nouvelles valeurs des variables à chaque tour de boucle viennent écraser les anciennes valeurs. Ainsi, en C, on peut faire déborder la pile si on lance trop d'appels récursifs imbriqués. Le même problème peut survenir en OCAML avec des fonctions récursives non terminales.

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 .

```

1 struct poly_s {
2     int n;
3     float* tab; //de taille n
4 };
5 typedef struct poly_s poly1;
6 typedef struct poly_s* poly2;
```

Solution

```

1 poly1 cree_poly_deg2(float a, float b, float c){
2     //cree une occurrence du polynome aX^2+bX+c
3     poly1 res;
4     res.n = 3;
5     res.tab = (float*) malloc (sizeof(float)*3);
6     res.tab[0] = c;
7     res.tab[1] = b;
8     res.tab[2] = a;
9     return res;
10 }
```



```

1 poly2 new_poly_deg2 (float a, float b, float c){
2     //cree une occurrence du polynome aX^2+bX+c
```

```

3 poly2 res = malloc (sizeof(struct poly_s));
4 res->n = 3;
5 res->tab = (float*) malloc (sizeof(float)*3);
6 res->tab[0] = c;
7 res->tab[1] = b;
8 res->tab[2] = a;
9 return res;
10 }

```

À retenir L'espace occupé par un objet de type `t*` est le même quel que soit le type `t`, c'est l'espace qu'occupe un pointeur. Pour remplir l'objet de type `t` au bout d'un tel pointeur, il faut avoir réservé l'espace pour un objet de type `t` (par déclaration sur la pile ou par `malloc` dans le tas selon les cas).

À retenir Pour retourner directement^a une struct, on peut déclarer un objet de type cette struct, le remplir et le retourner. En revanche si on veut retourner un pointeur vers une struct qu'on a créée dans le corps de la fonction, celle-ci doit avoir été enregistrée dans le tas et non dans la pile, sans quoi elle est écrasée lorsqu'on sort de la fonction. Il faut donc faire un `malloc`.

a. i.e. pas à travers un pointeur

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 remplace la ligne 3 par la ligne 4 d'une part, et si on décommente la ligne 16 d'autre part.

<pre> 1 char* c1 = "mpsi"; 2 char* c2 = "mpsi"; 3 char c3[5] = "mpsi"; 4 //char c3[4] = "mpsi"; 5 char c4[5]; 6 strcpy(c4,c1); //de string.h 7 char* c5 = (char*) malloc(5*sizeof(char)); 8 strcpy(c5,c1); //de string.h </pre>	<pre> 9 printf("Adresses enregistrées par\n"); 10 printf("c1 : %p / c2: %p\n",c1,c2); 11 printf("c3 : %p / c4: %p\n",c3,c4); 12 printf("c5 : %p\n\n",c5); 13 14 printf("Chaîne pointées par\n"); 15 printf("c1:%s/c2:%s\n",c1,c2); 16 //c2[2]='2'; 17 printf("c3:%s/c4:%s/c5:%s\n",c3,c4,c5); 18 c3[2] = '2'; 19 printf("c3:%s/c4:%s/c5:%s\n",c3,c4,c5); </pre>
--	---

Solution

On obtient par exemple l'affichage ci-contre.

```

1 Adresses enregistrées par
2 c1 : 0x557aaafecf004 / c2: 0x557aaafecf004
3 c3 : 0x7fff115a7753 / c4: 0x7fff115a774e
4 c5 : 0x557ab068b260
5
6 Chaîne pointées par
7 c1:mpsi/c2:mpsi
8 c3:mpsi/c4:mpsi/c5:mpsi
9 c3:mp2i/c4:mpsi/c5:mpsi

```

→ Les chaînes explicitement déclarées aux lignes 1 et 2 sont stockées dès la compilation

- dans un même endroit de la mémoire qu'on ne peut pas modifier. On le constate car les valeurs de c1 et c2, les adresses qu'elles enregistrent sont les mêmes.
- Les chaînes c3 et c4, si elles ne sont pas remplies de la même manière, sont déclarées de la même manière : comme un tableau statique, un tableau dont la taille est une constante littérale, et sont donc enregistrées dans la pile. On peut s'en convaincre en ajoutant la déclaration d'un ou plusieurs entiers et en affichant leurs adresses. On voit que les adresses se ressemblent ce qui confirme l'emplacement sur la pile. En observant les écarts entre les adresses, on pourra remarquer que déclarer un objet **char** c[5] revient à déclarer cinq variables de type **char** qui sont alors empilées consécutivement sur la pile.
 - La chaîne c5 est quant à elle enregistrée dans le tas.

Si on remplace la ligne 3 par la ligne 4, *i.e.* si on oublie de réservé la place pour le caractère de fin, l'affichage de c5 termine par des caractères non maîtrisés et non désirés.

Si on décommente la ligne 16, l'exécution de cette ligne déclenche une erreur de segmentation. En effet la chaîne c1 a été réservée dans une zone mémoire protégée, on ne peut pas la modifier (et heureusement sinon c2 serait modifiée par la même occasion). Cette zone est protégée même s'il y a un seul pointeur vers cette chaîne, par exemple ici même si on supprime c2.

À retenir Les objets de type **char*** définis par une chaîne explicite sont gérés de manière particulière à la compilation. Lorsque plusieurs occurrences de la même chaîne sont déclarées, la chaîne est enregistrée une seule fois et on obtient alors plusieurs pointeurs vers la même zone mémoire. Les pointeurs obtenus peuvent être modifiés (pour pointer vers autre chose) mais la zone mémoire en question ne peut être modifiée (afin de ne pas dégrader la chaîne vers laquelle pointe peut-être un autre pointeur).

À retenir Pour une chaîne de n caractères il faut un tableau de $n + 1$ **char** pour enregistrer aussi le caractère de fin de chaîne '**\0**'.

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.

Solution

```
1 typedef struct cell1_s cell1;
2 struct cell1_s {
3     int val;
4     cell1* next; //adresse de la cellule suivante ou NULL pr la dernière
5 };
6
7 typedef cell1* liste_c1;
8 //adresse de la 1ere cellule ou NULL pr la liste vide
```

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.

Solution

Il ne faut pas seulement un pointeurs enregistrant l'adresse de la première cellule, mais aussi un enregistrant la dernière. C'est pourquoi on définit ici aussi une structure pour les listes.

```
1 typedef struct cell2_s cell2;
2 struct cell2_s {
3     int val;
4     cell2* next; //adresse de la cellule suivante ou NULL pr la dernière
5     cell2* prev; //adresse de la cellule précédente ou NULL pr la 1ère
6 };
7
8 struct liste_c2_s {
9     cell2* debut; //adresse de la 1ere cellule ou NULL pr la liste vide
10    cell2* fin;
11 };
12 typedef struct liste_c2_s liste_c2;
13
```

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.

Solution

```
1 typedef struct cell3_s cell3;
2 struct cell3_s {
3     int entier;
4     char* chaine;
5     cell3* next; //adresse de la cellule suivante ou NULL pr la dernière
6 };
7
8 typedef cell3* liste_c3;
```

```
9 | //adresse de la 1ere cellule ou NULL pr la liste vide
```

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.

Solution

Il faut penser à stocker dans chaque cellule un entier supplémentaire pour la taille du tableau.

```
1 | typedef struct cell4_s cell4;
2 | struct cell4_s {
3 |     int n;
4 |     int* tab; //tableau de taille n
5 |     cell4* next; //adresse de la cellule suivante ou NULL pr la dernière
6 | };
7 |
8 | typedef cell4* liste_c4;
9 | //adresse de la 1ere cellule ou NULL pr la liste vide
```

3 Listes d'entiers simplement chaînées

Dans les questions suivantes on travaille avec le type ci-dessous.

```
1 | typedef struct s_cell cell;
2 | struct s_cell {
3 |     int val ;
4 |     cell* next ; /* adresse de la cell. suivante ou NULL en fin de liste */
5 | };
6 |
7 | /* adresse de la 1ere cellule ou NULL pr la liste vide */
8 | typedef cell* liste_c;
```

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 paramètre.

Solution

```
1 | cell* create_cell(int elem){
2 |     //retourne une liste réduite à une cellule contenant elem
3 |     //!!! attention allocation de mémoire dynamique --> penser au free !!
4 |     cell* res = (cell*) malloc(sizeof(cell));
5 |     res->val = elem;
6 |     res->next = NULL;
7 |     return res;
8 | }
```

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.

Solution

```
1 void affiche_liste(liste_c l){  
2     //affiche la liste l  
3     if(l == NULL){  
4         printf("La liste est vide.\n----\n");  
5     }  
6     else{  
7         cell* p = l; // p joue le rôle de curseur  
8         while(p != NULL){  
9             printf("%d --",p->val);  
10            p = p->next;  
11        }  
12        printf("\n----\n");  
13    }  
14}
```

R. 5-13 Définir une fonction qui prend en paramètre 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**.

Solution

```
1 liste_c cree_liste_premiers_entiers(int n) {  
2     //hyp : n >= 0  
3     //crée puis retourne une liste des n premiers entiers  
4     //!!! attention allocation de mémoire dynamique --> penser au free !!  
5     if (n == 0){return NULL;}  
6     liste_c res = create_cell(0);  
7     cell * p = res;  
8     for(int i = 1; i < n; i++){  
9         p->next = create_cell(i);  
10        p = p->next;  
11    }  
12    return res;  
13}
```

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

Solution

```
1 void free_liste_c(liste_c l){  
2     //libère la mémoire occupée par les cells de l  
3     cell* p = l;  
4     while(p != NULL){  
5         cell* temp = p->next;  
6         free(p);  
7         p = temp;  
8     }  
9 }
```

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

Solution

```
1 liste_c create_from_tab_alt(int* tab, int lg){  
2     // hyp : tab est de taille lg  
3     // retourne une liste chaînée contenant les éléments de tab  
4     // !! attention allocation dynamique de mémoire -> penser au free !!  
5     cell* res = NULL;  
6     for (int i = lg-1; i >= 0 ; i --) {  
7         cell* n = create_cell(tab[i]);  
8         n->next = res;  
9         res = n;  
10    }  
11    return res;  
12 }
```

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**.

Solution

On définit d'abord la fonction qui renvoie la longueur d'une liste afin de pouvoir réserver la bonne taille d'espace mémoire.

```
1 int nb_elem(liste_c l){  
2     //hyp : l n'est pas NULL  
3     //retourne le nombre d'élém de l  
4     cell* p = l;  
5     int res = 0;  
6     while(p != NULL){  
7         res = res + 1;  
8         p = p->next;  
9     }  
10    return res;  
11 }
```

Puis la création du tableau à partir d'une liste

```
1 int* tab_from_liste(liste_c l){  
2     //hyp : l n'est pas NULL  
3     // retourne un tableau contenant les éléments de l  
4     // !! attention allocation dynamique de mémoire -> penser au free !!  
5     int n = nb_elem(l);  
6     int* tab = malloc(sizeof(int)*n);  
7     cell* p = l;  
8     for(int i = 0; i < n; i++){  
9         tab[i] = p->val;  
10        p = p->next;  
11    }  
12    return tab;  
13 }
```

R. 5-17 Définir une fonction qui ajoute un à un les éléments d'une liste 11 dans une autre liste 12. Les listes 11 et 12 ne doivent pas être dégradées par cette procédure. On renverra la nouvelle liste,

dans laquelle les éléments de 11 apparaissent avant ceux de 12, en ordre inverse. Par exemple pour 11 la liste 1/2/3 et 12 la liste 4/5/6 le résultat attendu est la liste 3/2/1/4/5/6.

Solution

```
1 liste_c rev_append(liste_c l1, liste_c l2){  
2     //ajoute un à un les éléments de l1 en tête de l2 et retourne  
3     cell* p1 = l1; //curseur dans l1  
4     cell* res = l2; //tête de l2 avec les éléments ajoutés  
5     while(p1 != NULL){  
6         cell* new = create_cell(p1->val);  
7         new->next = res;  
8         res = new;  
9         p1 = p1->next;  
10    }  
11    return res;  
12 }
```

R. 5-18 Définir une fonction qui calcule le miroir d'une liste 11 sans la dégrader.

Solution

```
1 liste_c miroir(liste_c l1){  
2     //crée le miroir de l1 sans dégrader l1  
3     return rev_append(l1, NULL);  
4 }
```

R. 5-19 Définir une fonction qui crée une copie d'une liste 11 sans la dégrader.

Solution

```
1 liste_c copie(liste_c l1){  
2     //crée une copie de l1 sans dégrader l1  
3     liste_c temp = miroir(l1);  
4     liste_c res = miroir(temp);  
5     free_liste_c(temp);  
6     return res;  
7 }
```

R. 5-20 Définir une fonction qui prend en paramètre 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.

Solution

```
1 void tronque_liste(liste_c* pl1, int k){  
2     //hyp : 0 <= k <= la longueur de la liste *pl1  
3     //supprime les éléments de *pl1 au delà du k-ième  
4     int cpt = 0;  
5     liste_c* p = pl1;  
6     //inv : à l'adresse p est enregistrée l'adresse de la (cpt+1)-ième cell.  
7     while(cpt < k){  
8         p = &((*p)->next);  
9         cpt ++;
```

```

10 } //ici cpt =k
11 cell* to_delete = *p; //adresse de la cellule à supprimer
12 *p = NULL; //on rompt le lien vers le k+1ème élém
13 while(to_delete != NULL){
14     cell* temp = to_delete -> next;
15     //printf("....on supprime l'elem %d \n",to_delete->val);
16     free(to_delete);
17     to_delete = temp;
18 }
19 }
```

R. 5-21 Définir une fonction qui prend en paramètres 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.

Solution

```

1 void tronque_liste_robuste(liste_c* pl1,int k){
2     // hyp : 0<= k
3     // supprime les éléments de *pl1 au delà du k-ième
4     int cpt = 0;
5     liste_c* p = pl1;
6     //inv : à l'adresse p est enregistrée l'adresse de la (cpt+1)-ième cell.
7     while( (cpt < k) && ( *p != NULL ) ){
8         p = &((*p)->next);
9         cpt++;
10    } //ici cpt =k
11    if(*p != NULL){
12        cell* to_delete = *p; //adresse de la cellule à supprimer
13        *p = NULL; //on rompt le lien vers le k+1ème élém
14        while(to_delete != NULL){
15            cell* temp = to_delete -> next;
16            printf("....on supprime l'elem %d \n",to_delete->val);
17            free(to_delete);
18            to_delete = temp;
19        }
20    }
21 }
```

R. 5-22 Définir une fonction qui prend en paramètre 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 usage de la pile d'appels.

Solution

```

1 liste_c aux(liste_c todo, liste_c res){
2     //renverse les liens de la liste todo, met à la suite res
3     //et renvoie la tête de cette nouvelle liste
4     if(todo == NULL){return res;}
5     else{
6         cell* reste = todo->next;
```

```
7     todo->next = res;
8     return aux(reste, todo);
9 }
10}
11
12void miroir_en_place (liste_c* pl1){
13     *pl1 = aux(*pl1, NULL);
14 }
```