Programmation Procédurale en Langage C – TD4

Pointeurs : chaînes de caractères, tableaux, vecteurs et matrices

Exercice 1 : Comparaison de chaînes de caractères

1) Que font ces deux programmes ?

```
1  /* VERSION 1 */
2  #include <stdio.h>
3
4  #define MAXLEN 20
5
6  int main () {
7    char mot1[MAXLEN] ;
8    char mot2[MAXLEN] ;
9
10   printf ("Donnez␣un␣mot␣:␣") ;
11   scanf ("%s", mot1) ;
12
13   printf ("Donnez␣un␣mot␣:␣") ;
14   scanf ("%s", mot2) ;
15
16   if (mot1 == mot2)
17     printf ("Ces␣deux␣mots␣sont␣egaux\n") ;
18   else
19     printf ("Ces␣deux␣mots␣sont␣differents\n") ;
20
21   return 0 ;
22 }
```

```
1  /* VERSION 2 */
2  #include <stdio.h>
3  #include <string.h> /* strcmp() */
4
5  #define MAXLEN 20
6
7  int main () {
8    char mot1[MAXLEN] ;
9    char mot2[MAXLEN] ;
10
11   printf ("Donnez␣un␣mot␣:␣") ;
12   scanf ("%s", mot1) ;
13
14   printf ("Donnez␣un␣mot␣:␣") ;
15   scanf ("%s", mot2) ;
16
17   if (!strcmp (mot1, mot2))
18     printf ("Ces␣deux␣mots␣sont␣egaux\n") ;
19   else
20     printf ("Ces␣deux␣mots␣sont␣differents\n") ;
21
22   return 0 ;
23 }
```

2) Ces programmes sont-ils équivalents ? Que fait la fonction `strcmp()` ? *(Regardez partie 10 du cours...)*

3) Combien de comparaisons sont **effectivement** réalisées, lors de l'exécution, à la ligne 16 de la version 1 ? à la ligne 17 de la version 2 ?

4) Pourquoi mettre un `!` devant `strcmp()` ?

Exercice 2 : Allocation mémoire

Pour les besoins d'un programme, il est souhaité une fonction qui alloue dynamiquement de la mémoire pour un tableau de `N` valeurs initialisées avec la valeur `val`. Ci-dessous, deux versions de la fonction `creer_tableau()` implantent ce mécanisme.

```
3  /* VERSION 1                          *
4   * Fonction qui cree un tableau de   *
5   * taille size et le rempli avec val */
6  int *creer_tableau (int size, int val) {
7    int i, res[size] ;
8
9    /* Initialiser le tableau */
10   for (i=0 ; i<size ; i++)
11     res[i] = val ;
12
13   return res ;
14 }
```

```
4  /* VERSION 2                          *
5   * Fonction qui cree un tableau de taille *
6   * size et le rempli avec val. Le tableau *
7   * cree devra etre libere avec free()     */
8  int *creer_tableau (int size, int val) {
9    int i, *res ;
10
11   /* Allocation de memoire pour le tableau */
12   res = malloc (size * sizeof(int)) ;
13   if (res == NULL) {
14     fprintf (stderr, "Error:␣memory␣allocation\n") ;
15     exit (1) ;
16   }
17
18   /* Initialiser le tableau */
19   for (i=0 ; i<size ; i++)
20     res[i] = val ;
21
22   return res ;
23 }
```

1) En quoi ces fonctions sont-elles différentes ?

2) La compilation de la version 1 de la fonction émet un "*warning*". Selon vous, pourquoi ?

3) Quel comportement anormal peut produire l'utilisation de la version 1 ?

Voici un exemple de programme utilisant la fonction `creer_tableau()`, ainsi que des exemples d'exécutions :

```
25 int main () {
26   int *tab, N, i ;
27
28   printf ("Combien␣de␣valeurs␣:␣") ;
29   scanf ("%d", &N) ;
30
31   tab = creer_tableau (N, 0) ;
32
33   /* Afficher le tableau */
34   for (i=0 ; i<N ; i++)
35     printf ("%d␣", tab[i]) ;
36   printf ("\n") ;
37
38
39
40   return 0 ;
41 }
```

Avec la version 1

```
M:\ExemplesC> tableau-v1.exe
Combien de valeurs : 5
0 0 0 0 0
M:\ExemplesC> tableau-v1.exe
Combien de valeurs : 10
0 0 0 0 0 0 0 0 0 0
M:\ExemplesC> tableau-v1.exe
Combien de valeurs : 30
0 0 0 0 0 0 0 0 0 805306368 0 0 0
 0 0 -1209836560 -1208810272 0 0
0 0 0 0 0 0 0 0 0 0
M:\ExemplesC>
```

Avec la version 2

```
M:\ExemplesC> tableau-v2.exe
Combien de valeurs : 10
0 0 0 0 0 0 0 0 0 0
M:\ExemplesC> tableau-v2.exe
Combien de valeurs : 100
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0
M:\ExemplesC>
```

4) Comment expliquer les affichages produits par le programme lors de l'utilisation de la version 1 de `creer_tableau()` ?

5) Quel appel manque-t-il à la ligne 38 du programme pour une utilisation correcte de la version 2 de `creer_tableau()` ?

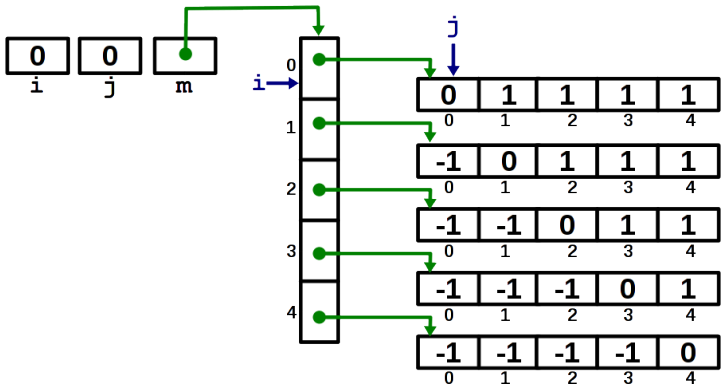
Exercice 3 : Parcours de matrices

En C, une matrice est un tableau de tableaux (ou donc un pointeur vers un tableau de pointeurs). Comme pour les tableaux, il est possible de créer une matrice de manière "en dur", *i.e.* alloué à la compilation. Par exemple, pour une matrice de taille N×M :

```
int mat[N][M] ;
```

Le programme, incomplet, ci-dessous procède à l'affichage d'une matrice carré. Complétez ce programme pour initialiser la matrice de manière à obtenir l'affichage ci-dessous.

```
1  #include <stdio.h>
2
3  int main () {
4    int i, j ;
5    int m[5][5] ;
6
7    /* Initialiser les valeurs de la matrice */
8
9
10
11
12
13
14
15
16
17
18
19   /* Affichage de la matrice */
20   for (i=0 ; i<5 ; i++) {
21     for (j=0 ; j<5 ; j++)
22       printf ("%2d␣", m[i][j]) ;
23     printf ("\n") ;
24   }
25
26   return 0 ;
27 }
```




```
M:\ExemplesC> matrice.exe
 0  1  1  1  1
-1  0  1  1  1
-1 -1  0  1  1
-1 -1 -1  0  1
-1 -1 -1 -1  0
M:\ExemplesC>
```

Exercice 4 : Le module Matrice (avec une petite couche de génie logiciel...)

Nous souhaitons écrire un paquetage de manipulation de matrices de réels. Pour ce faire, le type structuré sera le suivant :

```
typedef struct matrice {
  int n, m ;
  int **mat ;
} *matrice ;
```

À titre d'exemple, voici un paquetage de manipulation de **vecteurs** de réels :

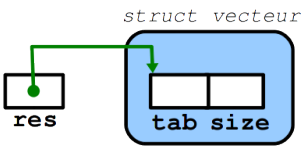
```
4  #ifndef __VECTEUR_H
5  #define __VECTEUR_H
6
7  /* Structure de donnee */
8  typedef struct vecteur {
9    int size ;
10   float *tab ;
11 } *vecteur ;
12
13 /* Creation / Destruction */
14 vecteur vecteur_creer (int N) ;
15 void    vecteur_detruire (vecteur v) ;
16
17 /* Acces a la structure */
18 float vecteur_get_val (vecteur v, int i) ;
19 void  vecteur_set_val (vecteur v, int i, float val) ;
20 int   vecteur_taille (vecteur v) ;
21
22 /* Fonctions de manipulation */
23 void  vecteur_remplir (vecteur v, float val) ;
24 void  vecteur_afficher (vecteur v) ;
25 void  vecteur_copier (vecteur dst, vecteur src) ;
26 vecteur vecteur_dupliquer (vecteur v) ;
27 void  vecteur_scalaire (vecteur v, float k) ;
28 float vecteur_somme (vecteur v) ;
29 char  vecteur_egal (vecteur v1, vecteur v2) ;
30 void  vecteur_addition (vecteur dst,
31                         vecteur src1,
32                         vecteur src2) ;
33 float vecteur_produit_scalaire (vecteur v1,
34                                 vecteur v2) ;
35
36 #endif /* __VECTEUR_H */
```

```
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <assert.h>
7  #include "vecteur.h"
8
9  vecteur vecteur_creer (int N) {
10   vecteur res ;
11
12   /* Allocation de la structure */
13   res = malloc (sizeof(struct vecteur)) ;
14   assert (res != NULL) ;
15
16   /* Allocaton du tableau */
17   res->size = N ;
18   res->tab = malloc (N * sizeof(float)) ;
19   assert (res->tab != NULL) ;
20
21   return res ;
22 }
23
24 void vecteur_detruire (vecteur v) {
25   /* Liberer le tableau */
26   free (v->tab) ;
27   /* Liberer la structure */
28   free (v) ;
29 }
30
31 float vecteur_get_val (vecteur v, int i) {
32   return v->tab[i] ;
33 }
34
35 void vecteur_set_val (vecteur v, int i,
36                       float val) {
37   v->tab[i] = val ;
38 }
39
40 int vecteur_taille (vecteur v) {
41   return v->size ;
42 }
```

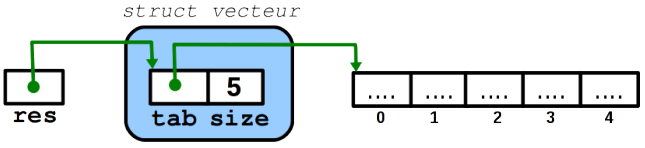
Construction de la structure de donnée dans la fonction `vecteur_creer()` dans `vecteur.c` : *(exemple avec `N` valant `5`)*

ligne 10 :

ligne 13 :



ligne 18 :



Suite du fichier `vecteur.c` :

```
43
44 void vecteur_remplir (vecteur v, float val) {
45   int i ;
46   for (i=0 ; i < v->size ; i++)
47     v->tab[i] = val ;
48 }
49
50 void vecteur_afficher (vecteur v) {
51   int i ;
52   for (i=0 ; i < v->size ; i++)
53     printf ("%5.2f ", v->tab[i]) ;
54   printf ("\n") ;
55 }
56
57 void vecteur_copier (vecteur dst, vecteur src) {
58   int i ;
59   assert (dst->size == src->size) ;
60   for (i=0 ; i < dst->size ; i++)
61     dst->tab[i] = src->tab[i] ;
62 }
63
64 vecteur vecteur_dupliquer (vecteur v) {
65   int i ;
66   vecteur res = vecteur_creer (v->size) ;
67   for (i=0 ; i < v->size ; i++)
68     res->tab[i] = v->tab[i] ;
69   return res ;
70 }
71
72 void vecteur_scalaire (vecteur v, float k) {
73   int i ;
74   for (i=0 ; i < v->size ; i++)
75     v->tab[i] *= k ;
76 }
77
78 float vecteur_somme (vecteur v) {
79   int i ;
80   float sum = 0. ;
81   for (i=0 ; i < v->size ; i++)
82     sum += v->tab[i];
83   return sum ;
84 }
85
86 char vecteur_egal (vecteur v1, vecteur v2) {
87   int i ;
88   char egaux = (v1->size == v2->size) ;
89   for (i=0 ; i < v1->size && egaux ; i++) {
90     if (v1->tab[i] != v2->tab[i])
91       egaux = 0 ;
92   }
93   return egaux ;
94 }
95
96 void vecteur_addition (vecteur dst, vecteur src1,
97                        vecteur src2) {
98   int i ;
99   assert (dst->size == src1->size
100          && dst->size == src2->size) ;
101  for (i=0 ; i < dst->size ; i++)
102    dst->tab[i] = src1->tab[i] + src2->tab[i] ;
103 }
104
105 float vecteur_produit_scalaire (vecteur v1,
106                                 vecteur v2) {
107   int i ;
108   float res = 0. ;
109   assert (v1->size == v2->size) ;
110   for (i=0 ; i < v1->size ; i++)
111     res += v1->tab[i] * v2->tab[i] ;
112   return res ;
113 }
```

Le fichier *interface* du module **Matrice** sera :

```
4  #ifndef __MATRICE_H
5  #define __MATRICE_H
6
7  /* Structure de donnee */
8  typedef struct matrice {
9    int n, m ;
10   float **mat ;
11 } *matrice ;
12
13 /* Creation / Destruction */
14 matrice matrice_creer (int N, int M) ;
15 void    matrice_detruire (matrice T) ;
16
17 /* Acces a la structure */
18 float matrice_get_val (matrice T, int i, int j) ;
19 void  matrice_set_val (matrice T, int i, int j,
20                        float val) ;
21 int   matrice_lignes (matrice T) ;
22 int   matrice_colonnes (matrice T) ;
23
24 /* Fonctions de manipulation */
25 void  matrice_remplir (matrice T, float val) ;
26 void  matrice_afficher (matrice T) ;
27 void  matrice_copier (matrice dst, matrice src) ;
28 matrice matrice_dupliquer (matrice T) ;
29 void  matrice_scalaire (matrice T, float k) ;
30 char  matrice_egal (matrice T1, matrice T2) ;
31 void  matrice_addition (matrice C, matrice A,
32                        matrice B) ;
33 void  matrice_produit (matrice C, matrice A,
34                        matrice B) ;
35 #endif /* __MATRICE_H */
```

Et le début du fichier *implantation* sera :

```
9  matrice matrice_creer (int N, int M) {
10   int i ;
11   matrice res ;
12
13   /* Allocation de la structure */
14   res = malloc (sizeof(struct matrice)) ;
15   assert (res != NULL) ;
16
17   res->n = N ;
18   res->m = M ;
19
20   /* Allocaton du premier tableau */
21   res->mat = malloc (N * sizeof(float *)) ;
22   assert (res->mat != NULL) ;
23
24   /* Allocaton des tableaux de valeurs */
25   for (i=0 ; i<N ; i++) {
26     res->mat[i] = malloc (M * sizeof(float)) ;
27     assert (res->mat != NULL) ;
28   }
29   return res ;
30 }
31
32 void matrice_detruire (matrice T) {
33   int i ;
34
35   /* Liberer les tableaux de valeurs */
36   for (i=0 ; i < T->n ; i++)
37     free (T->mat[i]) ;
38
39   /* Liberer le premier tableau */
40   free (T->mat) ;
41
42   /* Liberer la structure */
43   free (T) ;
44 }
```