

# TP 3 : Allocation dynamique de mémoire

Programmation en C (LC4)

Semaine du 11 février 2008

Pendant l'exécution d'un programme, une zone de la mémoire de l'ordinateur contient les données dont le programme a besoin pour s'exécuter, et qui peut être agrandie au cours de l'exécution.

L'allocation dynamique de la mémoire consiste donc à étendre, pendant l'exécution d'un programme, la mémoire qui lui est attribuée. Les principales fonctions d'allocation dynamiques sont :

- `malloc` pour allouer un bloc de mémoire
- `calloc` pour allouer un bloc de mémoire et l'initialiser à 0
- `realloc` pour agrandir la taille d'un bloc de mémoire
- `free` pour libérer un bloc de mémoire

Ces fonctions se trouvent dans la bibliothèque standard `<stdlib.h>`

Les prototypes de ces quatre fonctions sont les suivant :

« `void * malloc (size_t size)` » : alloue `size` octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé.

« `void * calloc (size_t nmemb, size_t size)` » : alloue la mémoire nécessaire pour un tableau de `nmemb` éléments, chacun d'eux représentant `size` octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros.

« `void * realloc (void * ptr, size_t size)` » : modifie la taille du bloc de mémoire pointé par `ptr` pour l'amener à une taille de `size` octets. `realloc()` réalloue une nouvelle zone mémoire et recopie l'ancienne dans la nouvelle sans initialiser le reste.

« `void free (void * ptr)` » : libère l'espace mémoire pointé par `ptr`, qui a été obtenu lors d'un appel antérieur à `malloc()`, `calloc()` ou `realloc()`.

Le type `size_t` est équivalent au type `unsigned long int`.

**Exercice 1** Ecrire les fonctions d'affichage d'un vecteur d'entiers de taille `dimension`, « `void affiche_vecteur(int * vecteur, int dimension)` » et d'une matrice d'entiers de taille « `lignes*colonnes` », « `void affiche_matrice(int ** matrice, int lignes, int colonnes)` ».

## ► Exercice 2

```
void affiche_vecteur(int * vecteur, int dimension)
{
    int i;
    printf("_");
    for(i=0;i<dimension;i++)
        printf("%d_",vecteur[i]);
    printf("\n");
}
```

```
void affiche_matrice(int ** matrice, int lignes,int colonnes)
{
    int i;
    for(i=0;i<lignes;i++)
```

```

    affiche_vecteur(matrice[i],colonnes);
    printf("\n");
}

```

**Exercice 3** Ecrire la fonction qui alloue la mémoire d'un vecteur de taille `dimension`, puis qui l'initialise à la valeur `val`, « `int * alloue_vecteur(int dimension, int val)` ». Ecrire la fonction « `void libere_vecteur(int * vecteur)` », qui libère le vecteur `vecteur`. Afficher ce vecteur pour tester vos fonctions.

► **Exercice 4**

```

int * alloue_vecteur(int dimension, int val)
{
    int i;
    int * vecteur = malloc(dimension*sizeof(int));
    for(i=0;i<dimension;i++)
        vecteur[i]=val;
    return vecteur;
}

void libere_vecteur(int * vecteur)
{
    free(vecteur);
}

```

**Exercice 5** Ecrire la fonction qui alloue la mémoire d'une matrice de taille « `lignes*colonnes` », puis qui l'initialise à la valeur `val`, « `int ** alloue_matrice(int lignes, int colonnes, int val)` ». Ecrire la fonction « `void libere_matrice(int ** matrice, int n)` », qui libère la matrice `matrice`. Afficher cette matrice pour tester vos fonctions.

► **Exercice 6**

```

int ** alloue_matrice (int lignes, int colonnes,int val)
{
    int i,j;
    int** matrice = malloc(lignes*sizeof(int *));
    for (i=0;i<lignes;i++)
        matrice[i]= malloc(colonnes*sizeof(int));
    for (i=0;i<lignes;i++)
        for(j=0;j<colonnes;j++)
            matrice[i][j]=val;
    return matrice;
}

void libere_matrice(int ** matrice,int lignes)
{
    int i;
    for (i=0;i<lignes;i++)
        free((int *)matrice[i]);
}

```

**Exercice 7** Ecrire une fonction qui génère une matrice identité en utilisant une simple boucle « `int ** genere_matrice_identite(int dimension)` ».

► **Exercice 8**

```
int ** alloue_matrice_zero(int lignes, int colonnes)
{
    int i;
    int** matrice = calloc(lignes,sizeof(int *));
    for (i=0;i<lignes;i++)
        matrice[i]= calloc(colonnes,sizeof(int));
    return matrice;
}
```

```
int ** genere_matrice_identite(int dimension)
{
    int i;
    int ** identite = alloue_matrice_zero(dimension,dimension);
    for (i=0;i<dimension;i++)
        identite[i][i]=1;
    affiche_matrice(identite,dimension,dimension);
    return identite;
}
```

**Exercice 9** *Le triangle de Pascal :*

En mathématiques, le triangle de Pascal est un arrangement géométrique qui stocke les coefficients du développement de  $(x + y)^i$  qui sont les coefficients binomiaux  $\binom{i}{j}$ . À la lignes  $i$  et colonnes  $j$  ( $0 \leq j \leq i$ ) est placé le coefficient binomial  $\binom{i}{j}$ , dont voici la représentation : (vous trouverez plus de détail sur la page [http://fr.wikipedia.org/wiki/Triangle\\_de\\_Pascal](http://fr.wikipedia.org/wiki/Triangle_de_Pascal))

1										
1	1									
1	2	1								
1	3	3	1							
1	4	6	4	1						
1	5	10	10	5	1					
1	6	15	20	15	6	1				
1	7	21	35	35	21	7	1			
1	8	28	56	70	56	28	8	1		
1	9	36	84	126	126	84	36	9	1	
1	10	45	120	210	252	210	120	45	10	1

- 1) Ecrire la fonction qui alloue la mémoire d'une matrice triangulaire inférieure carrée « `int ** alloue_matrice_pascal(int dimension)` ».
- 2) Ecrire une fonction « `int ** remplit_matrice_pascal(int dimension)` » qui stocke les coefficients binomiaux d'un polynôme de taille  $n$  (matrice de Pascal de taille  $n$ ).
- 3) Ecrire une fonction « `void affiche_matrice_pascal(int dimension)` » qui affiche une matrice de pascal de taille  $n$ .

► **Exercice 10**

```

int ** alloue_matrice_pascal(int dimension)
{
    int i,j;
    int** matrice = malloc(dimension*sizeof(int *));
    for (i=0;i<dimension;i++)
        matrice[i]= malloc(i*sizeof(int));
    return matrice;
}

int ** rempli_matrice_pascal(int dimension)
{
    int i,j;
    int** matrice = alloue_matrice_pascal(dimension);
    for (i=0;i<dimension;i++)
        {
            matrice[i][0]=1;
            matrice[i][i]=1;
        }
    for (i=2;i<dimension;i++)
        for(j=1;j<i;j++)
            {
                matrice[i][j]= matrice[i-1][j-1]+ matrice[i-1][j];
            }

    return matrice;
}

```

```

void affiche_matrice_pascal(int ** matrice, int dimension)
{
    int i;
    for(i=0;i<dimension;i++)
        affiche_vecteur(matrice[i],i+1);
    printf("\n");
}

```

La fonction « `int scanf (const char * format, ...)` » permet de lire les caractères de l'entrée standard. S'il s'agit d'entier, il faut utiliser le format `%d`, puis indiquer à quelle adresse mémoire stocker cet entier. Cette fonction se trouve dans la bibliothèque standard `<stdio.h>`.

Exemple : pour récupérer un caractère de l'entrée standard et le mettre dans la variable `car`

```

char recupere_caractere()
{
    char car;
    scanf("%c",&car);
}

```

**Exercice 11** 1) Ecrire une fonction « `int * recupere_n_entiers(int n)` » qui récupère consécutivement `n` entiers depuis l'entrée standard et les stocke dans un tableau de taille `n`.

2) Ecrire une fonction « `int * recupere_entiers(int n,int taille_max)` » qui après avoir alloué un tableau `tab` de taille `n`, récupère consécutivement des entiers depuis l'entrée standard. Lorsque plus de `n` entiers sont récupérés, cette fonction augmente la taille du tableau `tab` de `n` jusqu'à ce que la taille de `tab` soit égale à « `taille_max` ». Pour tester la fonction, on prendra « `n=5` » et « `taille_max=20` »

► **Exercice 12**

```
int* recupere_n_entier(int n)
{
    int i;
    int * tableau = alloue_vecteur(n,0);
    for(i=0;i<n;i++)
        scanf("%d",&tableau[i]);
    return tableau;
}

int * recupere_entiers(int n,int taille_max)
{
    int i;
    int * tableau = alloue_vecteur(n,0);
    for(i=0;i<taille_max;i++)
    {
        if(i == n)
        {
            n+=n;
            tableau= realloc((void *)tableau, sizeof(int) * n);
        }
        scanf("%d",&tableau[i]);
    }
    return tableau;
}
```

**Exercice 13** Ecrire la fonction qui alloue la mémoire d'un tableau 3D de taille « longueur\*largeur\*hauteur », « int \*\*\* alloue\_tableau\_3D(int longueur ,int largeur,int hauteur) ». Ecrire la fonction « void libere\_tableau\_3D(int \*\*\* tableau\_3D) », qui libère le tableau tableau. Après avoir écrit la fonction d'affichage « void affiche\_tableau\_3D(int \*\*\* tableau\_3D, int longueur, int largeur,int hauteur) », affichez ce tableau 3D pour tester vos fonctions.  
Remarque : essayer d'allouer la mémoire de ce tableau 3D de telle façon que la fonction « void libere\_tableau(int \*\*\* tableau\_3D) » n'utilise aucune boucle.

► **Exercice 14**

```
int *** alloue_tableau_3D(int longueur, int largeur, int hauteur)
{
    int *** tableau;
    int i, j;
    tableau = malloc (longueur * sizeof (int **));
    tableau[0] = malloc (longueur * largeur * sizeof (int *));
    tableau[0][0] = malloc (longueur * largeur * hauteur * sizeof (int));
    for (i = 0; i < longueur; i++)
    {
        tableau[i] = tableau[0] + i * largeur;
        for (j = 0; j < largeur; j++)
            tableau[i][j] = tableau[0][0] + i * largeur * hauteur + j * hauteur;
    }
    return tableau;
}

void libere_tableau_3D(int *** tableau)
{
}
```

```

free(tableau[0][0]);
free(tableau[0]);
free(tableau);
}

```

```

void affiche_tableau_3D(int *** tableau,int longueur, int largeur, int hauteur)
{
    int i,j,k;
    for(i=0;i<longueur;i++)
        {
            printf("_");
            for(j=0;j<largeur;j++)
                {
                    printf("_");
                    for(k=0;k<hauteur;k++)
                        {
                            printf("%d_",tableau[i][j][k]);
                        }
                    if(j==largeur-1)
                        printf("_");
                    else
                        printf("\n_");
                }
            printf("\n\n_");
        }
}

```

```

int main(int argc, char ** argv)
{

    int dimension=6;
    int lignes =5;
    int colonnes =4;
    int n=5;
    int taille_max=20;
    int longueur=4;
    int largeur=6;
    int hauteur=3;

    int * vecteur=alloue_vecteur(dimension, 18);
    int ** matrice =alloue_matrice (lignes,colonnes,33);
    int ** matrice_zero= alloue_matrice_zero(lignes,colonnes);
    int ** matrice_identite= genere_matrice_identite(dimension);
    int ** matrice_pascal= rempli_matrice_pascal(dimension);
    int *** tableau =alloue_tableau_3D( longueur, largeur, hauteur);
    int * tableau1= recupere_n_entier(n);
    int * tableau2= recupere_entiers(n, taille_max);

    affiche_vecteur(vecteur,dimension);
    affiche_matrice(matrice,lignes, colonnes);
    affiche_matrice(matrice_zero,lignes, colonnes);
    affiche_matrice(matrice_identite,dimension,dimension);
}

```

```
affiche_matrice_pascal(matrice_pascal,dimension);
affiche_tableau_3D(tableau, longueur, largeur, hauteur);
affiche_vecteur(tableau1,n);
affiche_vecteur(tableau2,taille_max);

libere_vecteur( vecteur);
libere_matrice(matrice,lignes);
libere_matrice(matrice_zero,lignes);
libere_matrice(matrice_identite,dimension);
libere_tableau_3D(tableau);
libere_vecteur(tableau1);
libere_vecteur(tableau2);

return 1;
}
```