TP 3 : Allocation dynamique de mémoire

Programmation en C (LC4)

Semaine du 12 février 2007

Pendant l'exécution d'un programme, outre la zone de mémoire où sont stockées les variables automatiques, une autre zone de la mémoire pouvant contenir des données existe et sa taille peut être ajustée au cours de l'exécution.
L'allocation dynamique de la mémoire consiste, pendant l'exécution d'un programme, à modifier la taille de cette zone de mémoire pour y placer de nouvelles données ou bien pour libérer l'espace occupé par des données devenues inutiles. Les 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 modifier la taille d'un bloc de mémoire
– `free()` pour libérer un bloc de mémoire

Ces fonctions appartiennent à la bibliothèque standard (et sont déclarées dans l'en-tête `<stdlib.h>`).

Les prototypes de ces quatres fonctions sont les suivants :

`void *malloc(size_t size)` : alloue `size` octets, et renvoie un pointeur sur le début du bloc de mémoire alloué. 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 occupant `size` octets, et renvoie un pointeur vers le début du bloc de mémoire alloué. Ce bloc est rempli 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 (éventuellement) un nouveau bloc de mémoire et recopie (éventuellement) l'ancien dans le nouveau (si la nouvelle taille est plus grande que l'ancienne, le reste du bloc mémoire n'est pas initialisé).
`void free(void *ptr)` : libère le bloc de mémoire pointé par `ptr`, qui a été obtenu lors d'un appel antérieur à `malloc()`, `calloc()` ou `realloc()`.
`size_t` est un type entier portable pour stocker la taille d'un tableau en mémoire.

Exercice 1 Écrire une fonction `void affiche_vecteur(int *vecteur, int dimension)` qui affiche un vecteur d'entiers de taille `dimension` et une fonction `void affiche_matrice(int **matrice, int lignes, int colonnes)` qui affiche une matrice d'entiers de taille `lignes` × `colonnes`,

Exercice 2 Écrire une fonction `int *alloue_vecteur(int dimension, int val)` qui alloue la mémoire nécessaire à un vecteur de taille `dimension`, puis qui initialise tous ses éléments à la valeur `val`. Écrire une fonction `void libere_vecteur(int *vecteur)` qui libère le vecteur `vecteur`. Afficher un vecteur pour tester vos fonctions.

Exercice 3 Écrire une fonction `int **alloue_matrice(int lignes, int colonnes, int val)` qui alloue la mémoire nécessaire à une matrice de taille `lignes` × `colonnes`, puis qui initialise ses éléments à la valeur `val`. Écrire une fonction `void libere_matrice(int **matrice, int lignes)` qui libère la matrice `matrice`. Afficher une matrice pour tester vos fonctions.

Exercice 4 Écrire une fonction qui renvoie une matrice identité en utilisant une seule boucle `int **genere_matrice_identite(int dimension)`.

Exercice 5 *Triangle de Pascal*

Le triangle de Pascal est une présentation géométrique (cf. figure) des coefficients du développement de $(x+y)^i$ qui sont les coefficients binomiaux $\binom{i}{j}$. À la ligne i et colonne j $(0 \leqslant j \leqslant i)$ est placé le coefficient binomial $\binom{i}{j}$.

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) Écrire une fonction qui alloue la mémoire nécessaire à une matrice triangulaire inférieure carrée `int **alloue_matrice_pascal(int dimension)`.
2) Écrire une fonction `int **remplit_matrice_pascal(int dimension)` qui renvoie une matrice de Pascal de taille `dimension`.
3) Écrire une fonction `void affiche_matrice_pascal(int dimension)` qui affiche une matrice de Pascal de taille n.

La fonction `int scanf(const char *format, ...)` de la bibliothèque standard (déclarée dans l'en-tête `<stdio.h>`) permet d'interpréter une suite de caractères tapée au clavier. Si cette suite de caractères représente un entier, on doit utiliser le format `%d` (comme avec la fonction `printf()`), et indiquer à quelle adresse stocker cet entier. Ainsi, pour récupérer un entier entré au clavier et le placer dans la variable `i`, on écrira :

```
int i;
scanf("%d", &i);
```

Exercice 6

1) Écrire une fonction `int *recupere_n_entiers(int n)` qui récupère `n` entiers entrés au clavier et les stocke dans un tableau de taille `n`.
2) Écrire une fonction `int *recupere_entiers(int n, int taille_max)` qui après avoir alloué un tableau `tab` de taille n, récupère des entiers entrés au clavier. 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 supérieure ou égale à `taille_max`. Pour tester la fonction, on pourra prendre, par exemple, `n = 5` et `taille_max = 20`.

Exercice 7 Écrire une fonction `int ***alloue_tableau_3D(int longueur, int largeur, int hauteur)` qui alloue la mémoire nécessaire à un tableau 3D de taille `longueur` × `largeur` × `hauteur`. Écrire une fonction `void libere_tableau_3D(int ***tableau_3D)` qui libère le tableau `tableau`. Après avoir écrit une fonction d'affichage `void affiche_tableau_3D(int ***tableau_3D, int longueur, int largeur, int hauteur)`, affichez un tableau 3D pour tester vos fonctions.
Remarque : essayez d'allouer la mémoire de ce tableau 3D de telle façon que la fonction `void libere_tableau(int ***tableau_3D)` n'utilise aucune boucle.