Langage C et Programmation Système
TP n° 5 : Enregistrements, pointeurs, allocation

Exercice 1 :

Ecrivez un programme qui affiche les valeurs des différentes variables au terme de chaque instruction dans le tableau ci-dessous. Comprenez les résultats obtenus. (*Indication : pour afficher la valeur d'un pointeur utiliser le formateur* `%p`).

	programme	a	b	c	p1, *p1	p2, *p2
	`int a, b, c, *p1, *p2;`	×	×	×	×	×
1	`a = 1, b = 2, c = 3;`					
2	`p1 = &a, p2 = &c;`					
3	`*p1 = (*p2)++;`					
4	`p1 = p2;`					
5	`p2 = &b;`					
6	`*p1 -= *p2;`					
7	`++*p2;`					
8	`*p1 *= *p2;`					
9	`a = ++*p2 * *p1;`					
10	`p1 = &a;`					
11	`*p2 = *p1 /= *p2;`					

Exercice 2 :

Ecrivez un programme qui contient le code suivant :

```
struct st3 { int a; int b; int c; };
int t[30];
int *p = t;
char *s = (char *) t;
struct st3 *pst3 = (struct st3 *) t;

int i;
for (i = 0; i < 30; i++)
    t[i] = 10 * i;
```

Ajoutez du code pour afficher les valeurs des expressions ci-dessous. Comprenez les résultats obtenus.

1. `*p + 2`
2. `*(p + 2)`
3. `&p + 1`
4. `&t[4] - 3`
5. `t + 3`
6. `&t[7] - p`

7. `p + (*p - 10)`
8. `*(p + *(p + 8) - t[7])`
9. `s[4]`
10. `&(s[4]) - &(s[2])`
11. `pst3[3].b`
12. `((struct st3 *)&t[6]) - pst3`

Exercice 3 : Jeu de la vie

Un automate cellulaire est constitué d'un ensemble de cellules dont l'état évolue en fonction du voisinage. Le plus connu est sans doute le Jeu de la vie, de John Horton Conway. Dans cet automate les cellules évoluent suivant deux règles simples :

- Une cellule morte dont exactement trois voisines sont vivantes naît
- Une cellule vivante entourée de deux ou trois voisines vivantes reste vivante et meurt sinon.

Nous allons écrire notre propre jeu de la vie. L'univers sera fini mais non borné, ainsi la cellule d'un bord sera voisine d'une cellule du bord opposé. L'état sera 1 pour vivante et 0 pour morte.

1. Définissez une structure `struct univ` qui contiendra les champs `char **m` et `int size`.

2. Ecrivez une fonction `struct univ *init_univ(int n)` qui alloue de l'espace pour une matrice carrée de bord `n`, l'initialise aléatoirement et retourne un pointeur sur l'univers.

3. Ecrivez une fonction qui affiche un univers.

4. Ecrivez une fonction `char left(struct univ *u, int i, int j)` qui retourne l'état de la cellule à gauche de celle dont les coordonnées sont passées (attention aux bords).

5. Ecrivez similairement `right`, `up`, `down`, `up_right`, etc. [1]

6. Ecrivez la fonction `void step(struct univ *u)` qui progresse d'un temps. Attention, l'état d'une cellule doit dépendre de l'état des voisines au temps précédent. Si une cellule ne devient vivante que durant la transition en cours, elle est considérée comme morte pour ses voisines durant cette transition.

7. Ecrivez une fonction qui initialise un jeu, lit un nombre d'itérations, exécute le Jeu de la vie en affichant à chaque itération l'univers avant d'attendre 1 seconde et continuer.

Exercice 4 : Allocation mémoire manuelle

Dans cet exercice, nous allons implémenter une méthode d'allocation mémoire alternative à celle proposée par la librairie standard avec `malloc(3)`. Pour cela, nous allons allouer statiquement un grand tableau dont nous distribueront dynamiquement des morceaux.
Nous stockeront les informations sur chaque bloc de mémoire qui a été alloué ou qui est allouable à l'aide d'un en-tête positionné au début du bloc mémoire, composé de deux champs :

- `int free`, un booléen indiquant si le bloc est libre ou occupé,
- `int size`, indiquant la taille du bloc.

1. Remarquez que vous pouvez aussi ajouter un paramètre `direction` et écrire une seule fonction `voisine`.

Initialement, il n'y aura aucun bloc : le tableau statique est vide. Le premier appel à la fonction d'allocation devra donc initialiser le premier bloc du tableau, marqué comme libre et de taille toute la mémoire disponible dans le tableau (moins la taille de l'en-tête). On utilisera une variable statique pour déterminer s'il s'agit du premier appel à la fonction d'allocation.
La mémoire pointée par notre grand tableau source sera à terme organisée ainsi :

libre	taille	zone memoire	libre	taille	zone memoire	libre	...

Lors d'une allocation dynamique, on cherchera un bloc (ou une suite de blocs) libre(s) ayant suffisamment de place, qu'on transformera en un bloc occupé (en ajoutant éventuellement un bloc marqué libre à la suite si la quantité demandée ne correspond pas exactement à la taille totale des blocs libres utilisés).

1. Créez trois fichiers `alloc.c`, `alloc.h` et `main.c`. Les implémentations de fonctions liées à notre allocation de mémoire devront toutes se faire dans le fonction `alloc.c`, et les fonctions utiles à l'utilisateur de notre méthode devront être déclarées dans `alloc.h`.
2. Définissez dans le fichier approprié une constante `N` correspondant à la taille (en octets) du tableau servant de source de mémoire, et définissez ce tableau : `static char memoire[N];`.
3. Définissez la structure de l'en-tête telle que décrite ci-dessus, qu'on appellera `struct memblock`.
4. Écrivez une fonction `void* search_free_blocks(int x)` qui cherche dans les zones mémoires allouées un emplacement libre d'au moins `x` octets dans un ou plusieurs blocs consécutifs. S'il n'y en a pas, cette fonction retournera `NULL`.
5. Écrivez une fonction `void *my_alloc(int x)` qui alloue `x` octets avec notre méthode et renvoie l'adresse de début de la zone du tableau allouée, en utilisant la fonction `search_free_blocks`. S'il n'y a plus assez d'espace disponible, la fonction retournera `NULL`.
6. Écrivez une fonction `void my_free(void *)`, notre version de `free(3)`. Attention au fait que l'adresse connue du programmeur n'est pas l'adresse de début du bloc, mais de la zone mémoire libre (il faut donc retrancher la taille de l'entête).
7. Testez ces fonctions à l'aide d'appels appropriés dans `main.c` (allouez des tableaux d'entiers de plusieurs tailles différentes, désallouez-les, recommencez...). Vous pourrez même essayer de l'utiliser dans l'exercice suivant.
8. BONUS : La fonction `search_free_blocks()` a tendance à fragmenter la mémoire. Écrivez une fonction `void *search_best_free_blocks(int x)` qui cherche la plus petite zone mémoire contenant `x` octets.
9. BONUS : Écrivez `my_realloc()` qui se comporte comme `realloc(3)`. Si à la suite du bloc à réallouer se trouve miraculeusement un ou plusieurs blocs libres que l'on peut récupérer, c'est parfait on les concatène. Sinon il faut allouer un nouveau bloc, copier le contenu existant, et libérer l'ancien bloc.