Série 6 :
Programmation *C* - Pointeurs 2 - Allocation dynamique

Buts

Le but de cette série d'exercices est de vous permettre de pratiquer les bases de l'allocation dynamique.

Rappel

Avez-vous pris connaissance des **conseils relatifs à ces séries d'exercices** ?

Exercice 1 : tableaux dynamiques (pointeurs, niveau 2)

NOTE : il faut faire cet exercice **sans** recopier le cours, mais par vous-même. Sinon cela n'a aucun intérêt !

Implémentez complètement la structure de donnée «tableau dynamique».
On devra pouvoir : créer, détruire un tableau dynamique, ajouter un élément à la fin, changer la valeur d'un élément à une position donnée et lire la valeur à une position donnée.

Prêtez particulièrement attention à l'intégrité des accès (l'endroit accédé doit être défini) et utilisez des conventions intelligentes pour les valeurs de retour en cas d'accès non conforme.

Exercice 2 : Multiplications de matrices revisitées (pointeurs + typedef, niveau 2 + 3)

Partie 1 : première amélioration : exercice sur les pointeurs

Reprendre l'exercice 6 de la **série 4**, c'est-à-dire copier votre programme `mulmat.c` [ou **le corrigé** si vous n'avez pas fait l'exercice `;-)`] dans un nouveau programme, puis éditez-le pour le modifier.

La principale critique du code de la semaine passée est que les fonctions `lire_matrice` et `multiplication` créeent leur propre `Matrice`, laquelle est **recopiée** en sortie (échange de l'information entre le `return` de la fonction et son appel). Il y a donc à chaque fois 2 `Matrice`s : celle de l'instruction qui fait l'appel et celle de la valeur de retour de la fonction appelée. Cela est coûteux et inutile.

Une solution pour éviter cette duplication des `Matrice`s est d'utiliser les **pointeurs**.

Changez les fonctions `lire_matrice` et `multiplication` de telle sorte qu'elles retournent un pointeur sur une `Matrice` (qu'elles auront elles-mêmes allouée).

Dans le `main`, déclarez trois pointeurs sur des `Matrice`s : `M`, `M1` et `M2`, puis modifiez le programme en conséquence et calculez `M = M1*M2`.

Pensez à libérer la mémoire dès que vous n'en avez plus besoin !

Partie 2 (niveau 3, FACULTATIVE) : seconde amélioration

Supposons maintenant que l'on souhaite effectuer plusieurs fois des produits de matrices. La solution précédente n'est pas très satisfaisante non plus car il faudrait dans ce cas **à chaque fois** libérer la mémoire dans le bloc appelant les fonctions `lire_matrice` et `multiplication`, et **à chaque fois** ces fonctions alloueraient une nouvelle place en mémoire. Perte de temps !

La bonne solution pour éviter à la fois les copie locale (partie 1) et les allocations/libérations de mémoire trop fréquentes est que les fonctions `lire_matrice` et `multiplication` modifient la valeur d'un argument supplémentaire (passé par référence), lequel représente le résultat de la fonction et serait alloué/initialisé par le bloc appelant.

Pour un emploi plus pratique de ces fonctions, celles-ci retournent de plus l'adresse de cet argument modifié. Cela permet d'utiliser ces fonctions elles-mêmes comme argument d'autres fonctions.

Cette solution donnerait les prototypes suivants:

```
Matrice*  lire_matrice(Matrice* lue);
```

et

```
Matrice*  multiplication(Matrice const * M1,
    Matrice const * M2,
    Matrice * resultat);
```

Recopiez à nouveau le programme puis éditez-le selon les prototypes ci-dessus. Testez-le dans le `main` avec les appels suivants :
(**Note :**) *attention*, ici `M1`, `M2`, `M3` et `M4` sont à nouveau des `Matrices` et non plus des pointeurs sur des `Matrices`, d'où le signe «&»).

```
lire_matrice(&M1);
multiplication(&M1, lire_matrice(&M2), &M3);
```

ou

```
multiplication(&M1, addition(&M2, &M3, &temp), &M4);
```

Partie 3 (niveau 2) : troisième amélioration

Transformez le code pour que les matrices soient allouées dynamiquement.

Prevoyez les fonctions supplémentaire nécessaires : initialisation, réallocation, libération.

Exercice 3 : réseau IP (pointeurs + typedef, niveau 2)

Dans le réseau Internet, vous savez sûrement que le routage TCP/IP se fait « de proche en proche », chaque nœud du réseau ne connaissant que ses voisins directs. C'est ce que nous allons modéliser ici.

Définissez un type de données pouvant représenter un nœud ayant (au moins) une adresse et un tableau de voisins, qui seront d'autres nœuds représentées *à l'aide* du même type de données, mais *sans copie*.

Une adresse est simplement un tableau de 4 entiers positifs de 8 bits chacuns (i.e. compris entre 0 et 255). Vous pouvez pour cela utiliser soit un tableau de 4 `unsigned char`, soit un `uint32_t` de C99 (bibliothèque `<stdint.h>`).

Concevez le type de données précédent pour être le plus dynamique possible (pas de constantes « en dur » dans le code). Tâchez également de rendre votre code le plus robuste possible.

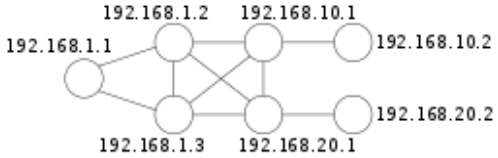
Écrivez ensuite (au moins) les fonctions suivantes :

- `creation()` permettant de créer une nœud en donnant en argument les 4 parties de son adresse (quatre `unsigned char`s) ;

  au début, un nœud n'a pas de voisins ;
- `sont_voisins()` prenant deux nœuds en paramètre et permettant de créer un lien « de voisinage » entre ces deux nœuds ;

  cette relation est symétrique (si A est voisin de B, alors B est voisin de A), il faudra donc la représenter comme telle (mettre le voisin chez chacun des deux nœuds) ;
- `voisins_communs()` prenant deux nœuds en paramètres et retournant le nombre de voisins communs ;
- `affiche` qui prend un nœud en paramètre et affiche son adresse ; elle devra également afficher les adresses de tous ses voisins directs.

Finissez le programme en écrivant dans le `main` le code correspondant à la situation suivante :



Affichez le nœud 192.168.10.1, puis affichez le nombre de voisins communs à 192.168.1.1 et 192.168.20.1, puis le nombre de voisins communs à 192.168.1.2 et 192.168.1.3.

Exemple de déroulement :

```
192.168.10.1 a 4 voisins : 192.168.1.2, 192.168.1.3, 192.168.10.2, 192.168.20.1.
192.168.1.1 et 192.168.20.1 ont 2 voisins communs.
192.168.1.2 et 192.168.1.3 ont 3 voisins communs.
```

Exercice 4 : jeu de « snake » (pointeurs + typedef, niveau 3)

Dans cet exercice, on vous demande d'implémenter le jeu du serpent où le joueur contrôle un serpent en mouvement et veille à ce qu'il ne touche ni les obstacles, ni les bords de l'écran, ni son propre corps. Le serpent grandit à chaque fois qu'il rencontre de la nourriture sur son passage.

Partez du fichier **`snake.c` fourni** qui contient le code pour afficher le jeu à l'écran et interagir avec l'utilisateur. Vous allez devoir y remplir les parties manquante : les structures de données dans et implémenter la logique du jeu.

Vous pouvez compiler le code du jeu soit comme d'habitude, soit si vous êtes sur une machine qui a la bibliothèque **ncurses** (instalée par exemple avec `sudo apt-get install libncurses5-dev`), utilisez alors la commande :

```
gcc -ansi -Wall -pedantic -DUSE_CURSES snake.c -o snake -lncurses
```

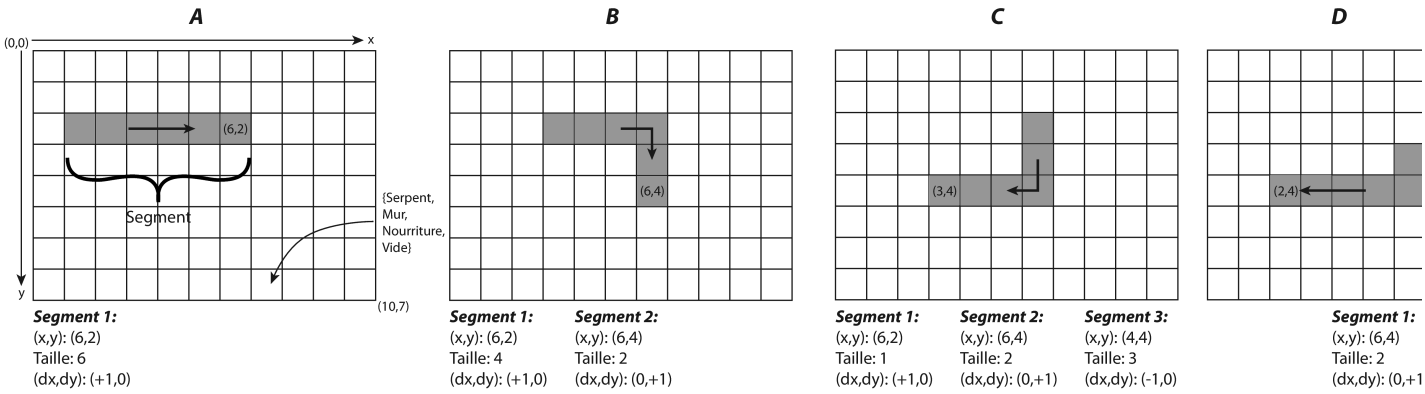
Quand vous aurez tout codé, vous pourrez déplacer le serpent avec les flèches de direction. Le jeu se termine si le serpent touche un mur ou se mord la queue.

Le serpent se déplace horizontalement ou verticalement dans un rectangle représenté par un tableau bidimensionnel dont l'origine correspond au coin supérieur gauche de l'écran. L'exemple de la figure ci-dessous montre un tableau de taille 11 par 8. Chaque case du tableau peut contenir une partie du serpent, un mur, de la nourriture, ou être vide.

Un serpent est représenté par une liste de segments. Lorsqu'il se déplace uniquement en ligne droite, le serpent n'a qu'un seul segment. A chaque fois que le serpent change de direction, un nouveau segment est créé. Plus le serpent est long, plus il peut se tortiller et plus il peut y avoir de segments.

Représenter le serpent par une liste de segments permet d'optimiser l'affichage. Lorsque le serpent se déplace, il suffit de redessiner la queue et la tête, le reste du corps ne bougeant pas visuellement.

Illustrons par un exemple en décrivant les étapes A, B, C et D de la figure suivante :



A. Le serpent avance en ligne droite vers la droite (direction (1,0)). La tête du segment se trouve aux coordonnées (6,2) et a une longueur de 6 cases.

B. Le serpent a changé de direction (le joueur a appuyé sur la flèche du bas) et un nouveau segment a été créé. Pendant que le serpent avance, la taille du segment de tête augmente et celle du segment de queue diminue.

C. Après que le serpent a avancé de deux cases, le joueur appuie sur la flèche gauche. Comme lors de l'étape précédente, un nouveau segment est créé, portant leur nombre à trois. Après que le serpent a avancé de trois cases, les segments 1, 2 et 3 ont respectivement une longueur de 1, 2 et 3.

D. Le serpent avance d'une unité, augmentant de 1 la taille de la tête et réduisant de 1 celle de la queue. Le segment de queue a désormais une taille nulle et est supprimé de la liste des segments. Le plus vieux segment restant devient le nouveau segment de queue.

4.1 - Types de données

Dans le fichier **`snake.c` fourni**, définissez les types de données pour représenter (voir ci-desosus) la direction du mouvement d'un segment, un segment, un serpent, le contenu d'une case du tableau, ainsi que le jeu lui-même. Veuillez respecter scrupuleusement le nom des types de structures, des membres et des fonctions indiquées car certaines d'entre elles sont utilisées par le reste du programme déjà écrit.

4.1.1. - Direction du mouvement

Définissez une structure de type `direction_t` ayant deux entiers *dx* et *dy*.
Nous prendrons comme convention que ces entiers valent -1, 0 ou 1, et qu'un seul à la fois peut être non nul (un serpent ne peut pas se déplacer en diagonale).

4.1.2 - Segment

Définissez une structure de type `segment_t` regroupant les éléments suivants :

- les coordonnées *x* et *y* du segment ;
- la longueur du segment ;
- la direction du segment ;
- un pointeur sur le segment précédent.

4.1.3 - Serpent

Un serpent est simplement une liste chaînée de segments. Le premier élément de la liste est la queue, le dernier est la tête. Définissez une structure de type `snake_t` contenant un pointeur sur la queue et un pointeur sur la tête.

4.1.4 - Cases du jeu

Le serpent évolue sur un tableau pouvant contenir du vide, des murs, de la nourriture, ou un morceau du serpent lui-même. Définissez un type énuméré `map_cell_t` avec les éléments `EMPTY`, `WALL`, `FOOD` et `SNAKE`, dans cet ordre.

4.1.5 - Définition du jeu

Définissez une structure de type `game_t` qui regroupe les éléments suivants :

- un serpent ;
- la largeur `width` du tableau ;
- la hauteur `height` du tableau ;
- un tableau *unidimensionnel* `map` d'éléments `map_cell_t` de taille quelconque (i.e. non connue à la compilation).

4.2 - Fonctions

Dans cette partie, vous allez implémenter une série de fonctions pour le moteur du jeu. Vous devez écrire ces fonctions dans la deuxième partie du fichier `snake.c` fourni.

Pour chacune de ces fonctions, veillez à vérifier la validité des paramètres passés et si nécessaire retourner une erreur adéquate. Toutes les fonctions qui retournent un entier doivent retourner une valeur non nulle en cas d'erreur, et zéro en cas de succès.

4.2.1 - Debugging de serpents

Ecrivez la fonction (de prototype :)

```
void snake_info(const snake_t* snake);
```

qui parcourt la liste des segments et affiche le contenu de chaque segment sur une ligne séparée. Vous pouvez définir le format et le contenu de l'affichage librement.

Cette fonction est appelée à chaque déplacement du serpent pour faciliter le déboguage (vous pouvez aussi l'utiliser ailleurs si ça vous est utile).

4.2.2 - Destruction de serpents

Écrivez la fonction

```
void snake_erase_tail(snake_t* snake);
```

qui supprime le segment de queue du serpent (i.e. l'enlève de la liste et libère sa mémoire).

Écrivez la fonction

```
void snake_destroy(snake_t* snake);
```

qui efface tous les segments du serpent. Réutilisez la fonction `snake_erase_tail`.

4.2.3 - Déplacement du serpent

Écrivez la fonction

```
int snake_add_segment(snake_t* snake, direction_t direction);
```

qui crée un segment, l'attache en tête du serpent et l'initialise avec la direction spécifiée. Un algorithme possible est :

1. allouer de la mémoire pour le segment ;
2. initialiser la direction du segment ;
3. rajouter le segment à la liste :
    - si c'est le premier segment, initialiser sa taille à 1 et initialiser la queue du serpent ;
    - sinon, initialiser sa taille à 0 et définir ses coordonnées (x,y) comme la somme du vecteur direction et des coordonnées du segment précédent (= la tête) ;
4. Remettre la tête à jour.

Écrivez la fonction

```
int snake_move(snake_t* snake, direction_t direction);
```

qui avance le serpent d'une case dans la direction indiquée. Un algorithme possible est :

1. Si la direction indiquée est la même que celle stockée dans le segment de tête du serpent, ajouter la direction aux coordonnées (x,y) de la tête, sinon créer un nouveau segment.
2. S'il y a plus d'un segment (i.e. la tête est différente de la queue), incrémenter la longueur de la tête et décrémenter la taille de la queue.
3. Si la longueur de la queue est nulle, détruire le segment correspondant.

2.2.4 - Mise à jour du tableau de jeu

Écrivez la fonction

```
map_cell_t* cell(game_t* game, unsigned int x, unsigned int y);
```

qui retourne (l'adresse de) la case de coordonnées (`x`, `y`) dans le tableau `map` du jeu `game`.

La convention de représentation unidimensionnelle dans ce tableau est que la la case de coordonnées (`x`, `y`) est stockée à l'indice « `x` plus `y` fois la largeur ».

Écrivez ensuite la fonction

```
int game_update(game_t* game, direction_t direction);
```

qui met à jour l'état du tableau de jeu. Un algorithme possible est :

1. Calculer les coordonnées actuelles (x_q,y_q) de l'extrémité de la queue : x_q = x - (n-1) d_x, y_q = y - (n-1) d_y, où x et y sont les coordonnées du segment de queue, n sa taille et (d_x, d_y) sa direction.
2. Faire avancer le serpent dans la direction passée en paramètres.
3. Si la tête du serpent rencontre un mur (`WALL`) ou si le serpent se rentre dedans (`SNAKE`), retourner une erreur.

   Sinon, si la tête rencontre de la nourriture (`FOOD`), incrémenter la taille de la queue.

   Sinon, vider (`EMPTY`) la case occupée par l'extrémité de la queue.
4. Mettre à jour (`SNAKE`) la case occupée par la tête.

4.2.5 - Initialisation du jeu

Finalement, écrivez la fonction

```
int game_init_snake(game_t* game, unsigned int orig_x, unsigned int orig_y);
```

qui crée un serpent avec un seul segment de longueur 1 et dont les coordonnées de départ sont `orig_x` et `orig_y`, et met à jour (`SNAKE`) la case correspondante.

Vous n'avez pas besoin de vous occuper du reste de la structure `game_t` car elle est initialisée dans le fichier `snake.c` qui vous est fourni.

Dernière mise à jour : $Date: 2012-03-05 09:43:12 $ ($Revision: 1.16 $)