TD2: matrices, struct, entrée/sortie, compilation

Exercice 1 – Allocation de tableau dynamique à 2 dimensions

Une matrice d'entiers se déclare statiquement ainsi : `int mat[5][6];`. La syntaxe indique assez clairement qu'une matrice d'entiers est en fait un tableau de tableaux d'entiers en langage C. Il est nécessaire de bien comprendre que :

- `mat` est de type `int**` (pointeur sur pointeur sur entier),
- `mat[1]` est de type `int*` (pointeur sur entier),
- `mat[1][3]` est de type `int` (entier).

Q 1.1 Écrivez une fonction qui permet d'allouer dynamiquement une matrice de taille (n, m), n et m étant passés en argument de la fonction. Cette fonction retourne la matrice créée.

Q 1.2 Proposer une fonction qui alloue dynamiquement une matrice, et qui retourne `1` si l'allocation s'est bien passée (assez de mémoire) et `0` sinon.

Exercice 2 – Matrice triangulaire et struct

Q 2.1 Expliquer comment allouer statiquement une matrice triangulaire, puis expliquer pourquoi il est plus intéressant de le faire dynamiquement.

Rappel : une matrice m est triangulaire supérieure si $m[i][j] = 0$ pour tout $i > j$, et triangulaire inférieure si $m[i][j] = 0$ pour tout $i < j$.

Q 2.2 Utiliser un `struct` pour représenter complètement une structure de matrice triangulaire (inférieure ou supérieure).

Q 2.3 Écrire une fonction permettant d'allouer dynamiquement une telle matrice.

Q 2.4 Écrire une fonction d'affichage d'une telle matrice, avec des zéros en dessous ou au dessus de la diagonale selon le type de matrice.

Exercice 3 – Lecture et écriture dans un fichier

On souhaite pouvoir écrire dans des fichiers le contenu de répertoires téléphoniques. Pour ce faire, notre programme définit et manipule le type suivant :

```
1 typedef struct personne{
2     char* nom;
3     long tel;
4 } Personne;
```

Un répertoire téléphonique peut être représenté par un tableau de Personne, ou encore un tableau de pointeurs sur Personne. De manière générale, on préfère utiliser des tableaux de pointeurs sur `struct` pour occuper une plus petite zone mémoire contiguë (plutôt que de faire un tableau de `struct` avec des

`struct` gourmands en espace mémoire). Dans cet exercice, on peut se permettre d'utiliser un tableau de Personne, comme une Personne est seulement composé d'un pointeur et d'un entier.

```
typedef struct repertoire{
   Personne* tab;
   int taille;
} Repertoire;
```

Q 3.1 Écrire une fonction qui commence par créer un fichier dont le nom est passé en argument, puis écrit dans ce fichier le contenu d'un répertoire téléphonique, lui aussi passé en argument. Chaque ligne du fichier correspond à une personne, et doit être de la forme `nom:tel`.

Q 3.2 À présent, on souhaite pouvoir écrire un répertoire téléphonique à la fin d'un fichier existant. Indiquer comment modifier la fonction précédente pour y parvenir.

Q 3.3 Écrire une fonction de lecture qui prend un nom de fichier en entrée, et qui retourne un répertoire téléphonique stocké dans le fichier.

Exercice 4 – Compilation séparée

Séparer un programme en plusieurs fichiers est une bonne idée quand le code prend de l'ampleur. La création de bibliothèques de fonctions accroît fortement la maintenabilité d'un code C et permet de penser à la ré-utilisabilité de code écrit.

Considérons un projet composé de trois fichiers `A.c`, `B.c` et `prog.c`, tels que `A.c` et `B.c` sont des librairies de fonctions utilisées par `prog.c`. On peut demander à compiler l'ensemble dans un seul programme (de nom `prog`) par la commande suivante :

```
gcc A.c B.c prog.c -o prog
```

Pour que la compilation se passe bien, il est nécessaire d'inclure dans `prog.c` au moins une spécification (une signature) des fonctions que `A.c` et `B.c` apportent au projet. Pour ce faire, on utilise généralement des fichiers *header* (extension ".h") contenant toutes les spécifications nécessaires à l'utilisation de la bibliothèque de fonctions. Les fichiers *headers* pourront ensuite être inclus dans chaque fichier `*.c` qui voudra utiliser les fonctions définies dans ces bibliothèques. Dans notre cas, nous allons donc inclure les fichiers `A.h` et `B.h` dans le fichier `prog.c`.

Par ailleurs, devoir compiler `A.c` et `B.c` à chaque fois que l'on fait une modification mineure dans le fichier `prog.c` n'est pas forcément utile. En effet, il s'avère que la compilation peut se décomposer en deux étapes bien différentes :

1. **La compilation** (proprement dite) : on passe d'un fichier `.c` à un fichier `.o`. Le fichier source est en fait compilé en langage machine (fichier objet). Mais le code est 'relogeable' : on ne connaît pas l'adresse des fonctions ni des variables. Plus exactement, les adresses sont encore 'translatables', données dans un mémoire virtuelle.
2. **L'édition de lien** : les mémoires virtuelles des différents fichiers objets sont unifiées et toutes les adresses sont résolues de manière univoque.

C'est uniquement cette dernière étape qui doit être faite d'un seul tenant, avec l'ensemble des fichiers-objets, résultats de compilations qui, elles, peuvent être séparées. Dans notre cas, nous pouvons donc exécuter les commandes suivantes :

```
gcc -c A.c
gcc -c B.c
gcc -c prog.c
gcc A.o B.o prog.o -o prog
```

puis exécuter seulement les deux dernières commandes en cas de modifications de `prog.c`.

Pour ne pas avoir à retenir quels fichiers ont été modifiés depuis la dernière compilation du projet, on peut utiliser l'outil `make` qui permet d'automatiser cette tâche à partir d'un `Makefile`. Un fichier `Makefile` suit un format permettant de déclarer les différentes dépendances entre fichiers d'un même projet, afin de trouver le nombre minimum d'opérations (compilations ou édition de lien) nécessaires afin de produire l'exécutable final. La commande `make` utilise le fichier `Makefile` pour effectuer ces opérations nécessaires.

Q 4.1 Reprenons nos trois fichiers `A.c`, `B.c` et `prog.c` et supposons que :
- seule une fonction `int f(int)` est définie dans le fichier source `A`,
- seule une fonction `void g(int)` est définie dans le fichier source `B` et qu'elle utilise la fonction `int f(int)`,
- le fichier `prog.c` contient seulement une fonction `main(void)` appelant `int f(int)` et `void g(int)`.

Proposer une organisation, le squelette de chaque fichier et le Makefile associé.