Portefolio de programmation C

Le travail de cette année a été divisée en deux partie, une par semestre. J'ai fait des projets au premier semestre et des TP du fil rouge au second.

Ayant déjà reçu une formation au langage C au cours du DUT, je profite des projets pour me remettre à niveau et répondre à des contraintes plus avancées.

Le deuxième semestre ne laissant que peu de temps pour travailler cette matière j'ai renoncé à cette stratégie pour faire les TP.

Un dépôt Github contient tout les travaux réalisés:

https://github.com/smanicome/Cours/tree/master/CoursC

Table des matières

1 Semestre 1

1.1) GuessWhich

Ce programme est un court jeu consistant à faire deviner un nombre à l'utilisateur.
Le jeu se déroule en trois étapes :

1. Demander la difficulté
   Pour cela il faut tout simplement laisser l'utilisateur entrer un chiffre sur l'entrée standard pour ensuite le récupérer dans le code. Un simple scanf() suffit en prenant soin de bien spécifier de vouloir un entier.
   Contrairement à la façon dont je l'ai premièrement implémenté, on pourrait changer le format pour un entier non signé et vérifier la valeur récupéré.

2. Générer un nombre aléatoire en fonction de la difficulté
   Afin d'éviter d'avoir toujours le même nombre à deviner avec la génération pseudo-aléatoire, je modifie le seed qui est par défaut à 1 pour l'heure au moment de l'exécution avec time(NULL).
   Le résultat de rand est alors rabaissé par le modulo de la difficulté sélectionné.

3. Faire deviner
   Pour faire deviner il suffit de récupérer un nombre de l'entrée standard et le comparer au nombre généré aléatoirement. Grâce à un compteur, l'utilisateur peux voir son nombre d'essais à la fin de la partie.

Le programme n'utilise que des fonctions de bases du C et ne présente aucune difficulté.
Ce projet m'a donc servi pour me remettre à niveau.

1.2) WordCount

Ce programme reprend le principe de la commande wc.
On doit pouvoir afficher le nombre mots, de lignes ou de caractères contenu dans un ou plusieurs fichiers.
Dans ce programme, j'ai fait en sorte de pouvoir exécuter la commande sur plusieurs fichiers en spécifiant une option pour le type de réponse, à défaut d'option le nombre de mots sera affiché.

De nombreuses notions sont abordées:
1. Gérer le manque de paramètre
   Un simple test sur argc, s'il est inférieur à 2 alors il manque le fichier à traiter

2. Ouvrir et fermer des fichier en C
   Une boucle permet d'ouvrir et fermer les fichiers passés en argument. Si le fichier n'existe pas un message d'erreur signale l'utilisateur du problème d'ouverture.
   Le fichier ouvert, en lecture seule, prend la forme d'un flux qui est fermé en fin d'utilisation.

3. Récupérer des options dans la ligne de commande
   En début d'itération on vérifie si l'argument passé est une option, elle sera alors ignoré pour éviter de la lire comme un fichier.
   Lorsque l'on tente de lire un fichier on cherche si une option est spécifié (c, w, l), s'il n'y en a aucune ou si elle n'est pas dans la liste alors l'option par défaut est w.

4. Lire des fichiers
   countChars
      Renvoie simplement la position du curseur en fin de fichier.

   countWords
      La lecture se fait caractère par caractère.
      Compte les mots espacé par de caractères «vides» (espace, tabulation, saut de ligne), un mot est défini par un caractère non «vide» précédé de rien ou d'un caractère «vide».

   countLines
      Compte simplement les caractères '\n'.

5. Segmenter le code
   Le code se devait forcément d'être fragmenté, autrement il aurait été simplement illisible. Les fonctions distinctes en fonction de l'option passé est donc pour moi incontournable.

Ce second projet, légérement plus compliqué que le précédent, m'a permis de me familiariser avec la lecture des fichiers et la manipulation de données en général.

1.3) Tee

Ce programme est une réplique de la commande homonyme. Elle doit permettre de récupérer le texte de l'entrée standard taper par l'utilisateur, pour l'écrire dans la sortie standard et les fichiers passés en paramètre.

Dans le cas de ce TP, nous avons donc la lecture sur l'entrée standard, et l'écriture sur les fichiers en sortie.

Pour gérer ces fichiers, une vérification de l'ouverture en écriture de ces derniers est réalisé, si le test est passé les fichiers sont stockés dans un tableau ré-alloué dynamiquement pour éviter d'utiliser excessivement ou insuffisamment de l'espace mémoire.

Pour terminer le programme, l'utilisateur doit interrompre la récupération de données en entrant CTRL + D, de cette manière les fichiers de sortie peuvent être convenablement fermés.

Ce projet m'a permis de m'avancer sur l'allocation dynamique de la mémoire. Après de multiple tentatives et version, je suis parti d'un programme simple à un fichier de sortie à une version plus avancée et sécurisée capable de supporter plusieurs fichiers de sortie en considérant que ces fichiers existent ou non.

1.4) Fortune

Fortune est un programme purement ludique, il permet d'afficher une citation ou une blague aléatoirement sélectionnée dans un fichier.

Pour réaliser cela l'utilisation d'un fichier indexé a été nécessaire. J'en ai trouvé un sur Internet déjà rempli et délimité.

Le programme commence donc par générer un chiffre aléatoire entre 0 et le nombre de citation. De la même manière que pour le premier projet, il faut modifier les seed.
Jusqu'à ce qu'un index valable est sélectionné, on recherche un index.

Ensuite il faut déplacer le curseur jusqu'à l'index du délimiteur et lire le fichier jusqu'au prochain délimiteur ou la fin du fichier.

Durant ce traitement, il faut s'assurer que l'index récupérer aléatoirement permet de lire une citation, il faut donc qu'une citation suive ce délimiteur.

Ce projet est plus avancées que les précédents dans la mesure ou je dois penser aux multiples contraintes placés sur le fichier ou les index. Pour répondre à cela j'ai donc créer des fonction générique me permettant de séparer le contenu d'un fichier avec les délimiteurs.

Pour pouvoir exploiter au mieux ce code, je l'ai alors séparer en deux fichier, la partie spécifique de fortune et celle du délimiteur.

Suite à tous ces projets de préparation, j'ai décider de me lancer dans un projet plus conséquent.

1.5) Alchemist

Ce projet m'a pris bien plus de temps que tous les autres cumulés.

Pour le réaliser j'ai suivi en grande partie les suggestion contenu dans le sujet. J'ai malgré tout apporté des modifications. Le jeu ne met plus en scène des balles mais des slimes colorés et animés.

Premièrement, je n'utilise pas de liste chaînée pour détecter les objets à fusionné, étant donné qu'un embranchement est possible il faudrait que plusieurs objets puissent en suivre un unique, et donc ne pas supprimer ce dernier avant que les autres ne soient supprimés.
Pour éviter cela j'ai exploité un second tableau qui a servi de calque, comme suggéré dans le sujet.

Pour détecter les objets à fusionner dans le plateau j'utilise une fonction récursive parcourant le tableau de bas en haut et de gauche à droite, elle est rappelée si une fusion à eu lieu.
Cette fonction appelle une autre fonction récursive qui détecte les objets de la même couleur et retourne la longueur du chemin tracé.

Ensuite un second appel récursif similaire au précédent permet de vider le tableau et d'afficher les animations de mouvement. Cette fonction est recréée séparément car il n'est pas possible de connaître la longueur totale du chemin depuis une branche, donc de savoir si la fusion aura lieu.

Deuxièmement, le projet est codé comme l'enseignant l'a pensé. Seules quelques améliorations se sont ajoutés, comme les animations de mouvement et d'attente ou la musique.

Lors du développement j'ai essayé de rendre le code le plus lisible et segmenté possible, j'ai donc pris soin de créer plusieurs fichiers séparés, de commenter les fonctions et d'ajouter un Makefile (celui proposé sur le site de la librairie MLV, légèrement modifié).

Pour placer une contrainte supplémentaire durant le développement, pour avoir un code plus propre, j'ai compilé le programme avec les flags `-Wall -ansi -pedantic`.

Je considère ce projet comme un excellent support pour se former à la programmation C.
Il reste cependant quelques améliorations possibles, comme générer les animations dans des threads, laisser à l'utilisateur la possibilté de créer une playlist, afficher une image de fond (simple, mais je n'ai aucune idée pour l'image)…

Ce projet est hébergé sur Github à ce lien.

2 Semestre 2

2.1) Exercices avec malloc

Pour mieux cerner les bonnes habitudes de l'allocation dynamique de mémoire je me suis lancé dans ce TP. J'ai déjà fait de l'allocation dynamique mais j'ai jugé qu'il aurait été judicieux d'apprendre son bon fonctionnement.

1. Allouer un tableau

Un premier exercice plutôt simple qui permet de se familiariser avec l'allocation et la libération de la mémoire.

Pour allouer ce tableau je récupère d'abord la taille du tableau sur l'entrée standard, puis la fonction get_array alloue un tableau d'entier de taille n et le rempli de 0 à n-1.

Le remplir permet de vérifier que l'allocation du tableau à bien réservé n emplacement mémoire permettant de stocker des entiers.

Enfin on libère la mémoire allouée avec free_array.

Là nous n'avons qu'un seul fichier, mais dans le cas où nous en avons plusieurs il est primordial de libérer la mémoire au même endroit où elle a été allouée.

2. Allouer un tableau à deux dimensions

Le principe est similaire au premier exercice car il suffit d'allouer un tableau de taille m (tab_m) dans chaque case du tableau de taille n (tab_n).

La différence se situe donc dans le type de données que l'on souhaite stocker dans tab_n, qui sont alors des int* et non des int.

L'autre différence se situe dans la libération de la mémoire, je pensais à tort que free serait récursif, mais il faut prendre soin de libérer soi-même toute la mémoire allouée.

3. Copier le tableau d'argument argv

Nous connaissons la taille de argv grâce à argc, ainsi il n'y a aucun problème pour la taille du tableau qui stockera les arguments.

Le problème qui se pose concerne la taille des arguments en question. Pour cela on récupère leurs taille avec strlen et on sert du résultat + 1, pour le caractère de fin \0, pour allouer chaque tableau de char.

En somme ce TP m'a permis de mieux comprendre le fonctionnement de malloc, notamment la libération de mémoire.

2.2) Tris multicritères sur listes chaînées de personnes

Ce TP est essentiel puisqu'il m'a initié aux fonction pointés.

Au moment où j'ai réalisé ce TP nous avions déjà vu les listes chaînés et les arbres donc tout ce qui concerne les structures y compris l'allocation dynamique et la libération de mémoire est acquis.

Dans un premier temps il faut lire le fichier liste_nom.txt et allouer des cellules pour chaque ligne.

Il faut donc un buffer pour stocker les valeurs lues et des tableaux seront ensuite alloués avec la bonne taille, de la même manière que le TP précédant.

Ensuite une fois qu'on a récupéré toutes les informations d'une ligne, on peut allouer la cellule avec les informations nécessaires.

Les cellules créer seront alors mis les un à la suite des autres sous forme de liste chaînée.

C'est donc maintenant que la deuxième partie commence, il faut que la liste soit triée.

J'ai alors créé les fonctions age_order et name_order qui permettent de comparer deux cellules respectivement en fonction de l'age et du nom.

Ce sont ces fonctions qui seront passés en paramètre à la fonction ordered_insertion, qui parcourra la liste jusqu'à placer la nouvelle cellule de telle sorte qu'elle réponde au critère de tri.

Ce TP est bien plus compliqué que les précédant car il demande la connaissance des structures, des liste chaînées, des tris, de la lecture de fichier et enfin des fonctions pointées.

En raison de cette plus grande diversité dans les notions abordés je considère ce TP comme l'un des plus intéressant.

2.3) Taquin

Ce TP s'oriente plus comme un projet, ce qui le rend d'autant plus intéressant car il laisse la possibilité de concevoir soi-même là logique à adopté.

Cependant le projet de base, sans les améliorations, reste assez simple à réaliser.

Étant donné que j'ai fait le jeu Alchemist, j'ai déjà utilisé la librairie graphique de l'université. Je n'ai donc rien a jouté de ce côté.

J'ai divisé le code en trois fichiers:

- taquin.c pour initialisé le plateau et lancer la partie
- move.c pour gérer les mouvements de tuiles
  - move() : bouge la tuile vide en fonction des arguments passés, vertical et horizontal
  - random_move() : tire un nombre aléatoire modulo 4, pour les quatre directions, puis procède au mouvement si celui-ci est possible en appelant move()
  - shuffle() : appelle 120 fois random_move()
- play.c pour la boucle de jeu et l'affichage graphique
  - print_board() : permet de vérifier sur la console que l'affichage graphique reflète bien l'état du plateau
  - play() : ouvre l'image à reconstituer et lance la boucle de jeu
  - display_image_board() : affiche l'état du plateau dans la fenêtre
  - draw_image_tile() : affiche la fraction de l'image défini dans le bloc à la position du bloc dans le plateau
  - display_board() : affiche juste de chiffre dans la fenêtre au lieu d'une image
  - won() : vérifie si chaque bloc est à sa place, qui est donc la condition de victoire
  - user_action() : boucle jusqu'à ce que le joueur clique sur une tuile valide puis procède au mouvement de la tuile vide.

J'ai repris la structure suggéré dans le sujet en ajoutant dans Plateau les champs empty_i et empty_j permettant de localiser directement la tuile vide. Cela m'évite d'avoir à parcourir le tableau à chaque fois que je souhaite bouger la tuile.

Le jeu se joue à la souris, il faut cliquer sur une tuile adjacente à la tuile vide pour pouvoir échanger leurs emplacement.

Ce projet est intéressant car sa structure simple permet de s'accommoder au développement d'un projet en C sans majeur difficulté.

J'ai fait ce TP car j'avais reçu ce même sujet en guise de projet en DUT et qu'il ne fonctionnait pas totalement. C'était l'occasion de revenir sur un échec et je constate que ce qu'il me manquait à l'époque était juste de l'expérience, puisque c'était le premier TP de la formation en fin de premier semestre.

J'avais prévu d'aller plus en laissant le choix au joueur de jouer dans une banque d'image, ou en chiffre avec la possibilité de modifier la difficulté, mais je n'ai pas eu le temps de le faire.

La fonction display_board() est la seule fonction terminée qui aurai due être ajouté que j'ai pu laisser sans «casser» le programme.

2.4) Table de hashage

Ce TP consiste à créer une table de hashage dans laquelle on stockera chaque mot d'un fichier une seul fois. Lorsque l'on teste cette table, on se rend compte que la recherche d'un mot est bien plus rapide qu'avec un simple parcours de tableau ou de liste.

25353 different words found in Germinal

real 0m0,197s
user 0m0,175s
sys 0m0,013s

Pour le stockage dans une unique liste, mon ordinateur n'étant pas très puissant, et le parcours étant assez long je n'ai pas de chiffre à indiquer pour comparer. Ceci dit, le fait que je n'en vois pas le bout suffit à dire que c'est très long.

Je n'ai rien inventé dans ce TP, il suffisait de suivre le sujet, sachant qu'on a déjà vu les structures, listes, etc. tout ce qu'il reste à faire est faire la table.
Autrement dit allouer une table statique de taille M contenant des Node**, soit une liste de nœud. Ces listes sont les sceaux qui contiendront tous les mots qui répondront à un même critère.
Ce critère est le hash de ce mot modulo le nombre de sceaux, de cette manière il suffit de récupérer le sceaux directement avec le hash.

Ensuite il faut vérifier que le mot n'est pas déjà présent dans le sceaux, s'il n'y est pas on l'ajoute sinon on ne fait rien.

Puisqu'on divise la liste en sceaux, il est bien plus simple de trouver un mot qu'en parcourant une unique liste, et cette différence sera d'autant plus importante à mesure que le nombre de sceaux augmente.

L'utilisation d'une table de hashage est une méthode très efficace lorsque l'on doit implémenter une structure similaire à un dictionnaire, autrement dit stocker une quantité importante de données pour ensuite y chercher un élément spécifique.