Université de Lille

Projet : Sokoban

m2101
DUT Info
Julien Baste
2020–2021

Consigne : L'objectif de ce projet est de vous faire manipuler tous les concepts du C que vous avez pu voir pendant le semestre. Il est attendu que vous fassiez au moins les exercices 1 à 4. Les exercices 5 et 6 sont volontairement plus compliqués, mais font partie intégrante de la notation. L'exercice 7 est simplement du bonus. Vous devez avoir fait les 4 premiers exercices pour que l'exercice 7 soit considéré.

Dans cet énoncé, je ne vous donnerais aucune contrainte sur la structure de votre projet. À vous de faire des choix organisés. Pour cela je vous conseille de lire la totalité du sujet avant de commencer. Il est attendu que votre code soit commenté. Par là, j'entends que pour chaque fonction il faut minimum une phrase qui explique ce que fait la fonction. Vous placerez ce commentaire au début de la fonction considérée.

Le projet est à faire par groupe de 2. Le projet est à rendre pour le 11 avril 2021 minuit. Vous le déposerez sur le moodle dans l'activité "Projet".

Pour le rendu, vous devrez compresser votre dossier de travail au format `zip` en donnant à votre archive le nom `prenom1.nom1.prenom2.nom2.zip`. Vous ajouterez à cela un petit rapport d'environ une page décrivant les difficultés que vous avez rencontrées et la manière dont vous les avez résolues ou non. Ce rapport est principalement là pour me permettre d'évaluer ce qui ne fonctionne pas dans votre code.
ATTENTION : Ce rapport indiquera aussi la composition de votre groupe en indiquant nom et prénom de chacun.
NOTE : Vous ne rendrez qu'un seul projet par groupe. (Je ne veux pas avoir à corriger deux fois le même projet, il ne faut pas déconner.)

Pour ce projet, vous avez 3 séances de 1h30 qui sont prévues pour vous débloquer et vous permettre d'avancer. Si vous attendez le dernier moment pour vous y mettre, vous ne pourrez donc pas en bénéficier. Indice : ce sont les derniers exercices qui sont les plus compliqués. Il vous est fortement recommandé d'arriver à la deuxième séance avec au moins les exercices 1, 2 et 3 de terminé et à la troisième séance avec l'exercice 4 de terminé et avoir déjà essayé d'avancer dans les exercices 5 et 6.

Sokoban (ou « garde d'entrepôt ») est un jeu vidéo de réflexion inventé au Japon en 1982. Dans ce jeu, le joueur doit ranger des caisses sur des cases cibles. Il peut se déplacer dans les quatre directions, et pousser (mais pas tirer) une seule caisse à la fois. Une fois toutes les caisses rangées, le niveau est réussi. L'idéal est de réussir avec le moins de déplacement possible. [1]

Ce jeu a connu beaucoup de versions et d'adaptation différentes. Dans le cadre de ce projet, vous allez implémenter une version utilisable sur le terminal. Une version en terminal peut être moins intuitive qu'une version graphique que l'on peut trouver et jouer sur internet, mais à l'avantage de pouvoir être implémentée avec les connaissances que vous avez acquis pendant ce semestre. Une version exemple de ce qui est attendu est disponible sur les machines de TP. Pour cela, vous devez effectuer un `source /home/public/m2101/sokoban/exemple` ce qui va créer dans votre espace de travail le dossier `~/m2101/projet_exemple` dans lequel vous trouverez l'exécutable `sokoban` du jeu ainsi qu'un dossier `Niveaux` contenant deux fichiers correspondant à deux niveaux du jeu. Le niveau 0 proposé est un niveau bac à sable et le niveau 1 correspond au premier niveau du jeu original. Comme d'habitude, je vous encourage à tester le jeu pour comprendre comment il fonctionne.

Pour comprendre l'affichage, il faut distinguer les différents caractères utilisés pour représenter les différents éléments du jeu :

- `' '` représente les cases vides
- `'#'` représente les murs (cases infranchissables)
- `'$'` représente une caisse que vous pouvez déplacer
- `'.'` représente une zone de rangement sur laquelle vous devez placer une caisse
- `'*'` représente une zone de rangement sur laquelle une caisse est actuellement placée
- `'@'` représente votre personnage
- `'+'` représente votre personnage lorsqu'il est sur une zone de rangement.

Exercice 1 : Génération du niveau

Pour implémenter ce jeu, vous allez, comme pour le démineur, travailler avec un tableau qui représentera votre terrain. Ce tableau sera en une seule dimension dans la mémoire, mais vous écrirez des fonctions d'interface qui vous permettront de traiter le tableau comme s'il était en deux dimensions. La différence depuis le TP sur le démineur est que vous connaissez désormais les structures. Au lieu de travailler avec des tailles fixées de terrain, le nombre de lignes et le nombre de colonnes seront écrit dans la structure. Cela permettra de gérer de manière dynamique le passage d'un niveau à un autre, niveaux qui ne seront pas forcément de la même taille.

Q1. Définissez une structure `niveau_t` qui contient comme attribut `terrain`, qui sera un pointeur vers la première case d'un tableau de `char` de taille qui sera défini plus tard, `nb_colonnes` qui contiendra le nombre de colonnes du terrain et `nb_lignes` qui contiendra le nombre de lignes du terrain.

Q2. Écrivez une fonction `niveau_t* nouveau_niveau(int nb_colonnes, int nb_lignes);` qui vous génère un nouveau niveau où `nb_colonnes` et `nb_lignes` sont correctement initialisés et où `terrain` cible un espace mémoire correspondant à un tableau de la bonne taille et qui retourne un pointeur vers la structure générée.

Q3. Écrivez une fonction `void place_sur_terrain (niveau_t* niveau, int colonne, int ligne, char car);` qui place le caractère `car` dans le tableau `terrain` qui est un attribut du niveau `niveau`. Le caractère sera placé à la ligne `ligne` et la colonne `colonne`. Rappel : Vous avez déjà fait cela pour le TP du démineur.

Q4. Écrivez une fonction `char lecture_du_terrain (niveau_t* niveau, int colonne, int ligne);` qui lit le caractère du tableau `terrain`, qui est un attribut du niveau `niveau`, qui est placé à la ligne `ligne` et la colonne `colonne`. Rappel : Vous avez déjà fait cela pour le TP du démineur.

Q5. Les fonctions `place_sur_terrain` et `lecture_du_terrain` sont vos fonctions d'interfaçage entre votre vision du terrain en deux dimensions et celle de la mémoire en une seule dimension. Désormais vous accéderez au tableau `terrain` de votre structure `niveau_t` uniquement au travers de ces fonctions.

- Écrivez une fonction `void initialise_le_terrain (niveau_t* niveau);` qui initialise toutes les cases du terrain du niveau `niveau` avec un mur (c'est-à-dire le caractère `'#'`).
- Écrivez une fonction `void affichage_niveau (niveau_t* niveau);` qui affiche votre terrain à l'écran.

Q6. Testez vos fonctions. Normalement l'application de

- `nouveau_niveau(10,4);`, de
- `initialise_le_terrain(niveau)` et de
- `affichage_niveau(niveau);` devrait vous produire :

```
##########
##########
##########
##########
```

1. C'est quand même vachement pratique wikipedia.

Q7. Maintenant que vous savez générer un niveau vide, il est temps de charger en mémoire des niveaux complets. Pour ce projet, tous les niveaux seront stockés sous forme de fichier dans un dossier `Niveaux`. Ces fichiers contenant les niveaux seront tous construits de la manière suivante :

— un entier représentant le nombre de colonnes du niveau
— un espace
— un entier représentant le nombre de lignes du niveau
— un retour à la ligne
— le niveau en lui-même composé des caractères précédemment présentés.

Vous trouverez deux exemples de fichier niveau dans `~/m2101/projet_exemple/Niveaux`. Les noms des fichiers sont de la forme `niveau_nb` où `nb` est remplacé par le numéro du niveau.

Écrivez une fonction `niveau_t* lecture_du_niveau(int quel_niveau);` qui va lire le fichier niveau numéro `quel_niveau` et qui va le charger en mémoire dans une structure niveau qu'il aura créé pour l'occasion. Testez en affichant le niveau chargé. Attention si `niveau_0` contient pile le bon nombre de caractères pour chaque ligne, `niveau_1` contient des lignes qui ne sont pas complétés jusqu'à 19 colonnes et dans ce cas il faut compléter en ajoutant des espaces. De même, certaines lignes contiennent des espaces supplémentaires en fin de ligne, dans ce cas il faut ignorer les espaces en trop. Conseil : utilisez la fonction `scanf` pour lire les nombres et la fonction `getc` pour lire dans le terrain.

Contenu de `niveau_0` qui est un niveau bac à sable créé pour vos tests :

```
10 7
##########
#        #
#  $   @ #
#    #   #
#  .     #
#        #
##########
```

Contenu de `niveau_1` qui est le premier niveau de la version originale du jeu :

```
19 11
    #####
    #   #
    #$  #
  ###  $##
  #  $ $ #
### # ## #   ######
#   # ## #####  ..#
# $  $         ..#
##### ### #@##  ..#
    #     ########
    #######
```

Q8. Biensur vous avez écrit une fonction `void liberation_du_niveau (niveau_t* niveau);` qui libère correctement la mémoire occupée par une structure de type `niveau_t`.

Exercice 2 : Déplacement de votre personnage

Dans cette partie, vous allez vous intéresser à déplacer votre personnage.

Q1. Pour pouvoir déplacer votre personnage, vous devez en premier lieu lire l'entrée fournie par l'utilisateur. Pour ce projet il vous est proposé de jouer avec les touches `zqsd` (`z` pour haut, `q` pour gauche, `s` pour bas et `d` pour droite).

Écrivez une fonction `char entree_du_joueur (void);` qui lit les caractères entrés par le joueur jusqu'à ce qu'il tombe sur une lettre `z`,`q`, `s` ou `d` puis retourne cette lettre.

Q2. Pour savoir où se situe votre personnage, il y a plusieurs manières de faire et toutes seront acceptées. Dans ce sujet et afin de vous guider, il vous est proposé de mémoriser spécifiquement la position du joueur à l'aide d'une structure `point_t` qui aura pour attributs `colonne` et `ligne`. De plus la position de votre personnage sera enregistrée dans votre structure `niveau_t` pour que cela fasse un tout.

Définissez une structure `point_t`, ajoutez un attribut `perso` de type `point_t*` à la structure `niveau_t` et modifiez vos fonctions pour que `perso` soit correctement initialisé et pris en compte.

Q3. Écrivez une fonction `void deplacement (niveau_t* niveau, char direction);` qui prend en paramètre l'état du jeu `niveau` et une direction et déplace votre personnage, si c'est possible, dans la direction donnée par `direction`.

Votre personnage peut se déplacer si la case d'arrivée :

— est vide (ou une zone de rangement)
— est une caisse et que la case suivante est vide (ou une zone de rangement). Dans ce cas là votre personnage doit pousser correctement la caisse à la case d'après.

Dans tous les autres cas, votre personnage ne peut pas se déplacer.

Attention : votre personnage doit être représenté par `'@'` lorsqu'il est sur une case vide et par `'+'` lorsqu'il est sur une zone de rangement. Vérifiez que cela se passe correctement. De même pour les caisses.

Note : Vous êtes fortement invité à écrire des fonctions intermédiaires. Je vous laisse libre déterminer les fonctions dont vous aurez besoin (c'est un projet quand même :-P). Pour vous donner une idée j'ai utilisé 5 fonctions annexes dans mon implémentation du jeu.

Exercice 3 : Fin du jeu

Q1. Maintenant que vous savez charger un niveau et vous déplacer dedans, il ne vous reste plus qu'à vérifier à quel moment le jeu est terminé.

Écrivez une fonction `int niveau_termine (niveau_t* niveau);` qui retourne si vous avez gagné ou non. Il est rappelé que le jeu se termine si toutes les caisses sont sur des zones de rangement. Vous pouvez par exemple tester qu'il n'y ait plus de `'$'` sur le terrain.

Q2. Vous pouvez désormais avoir un jeu fonctionnel qui vous informe lorsque vous avez gagné.

Q3. En plus de savoir que vous avez réussi à terminer le niveau, vous voulez aussi savoir en combien de déplacement vous avez réussi à finir le niveau. Pour cela vous ajouterez un nouvel attribut entier `nb_de_pas` à votre structure `niveau_t` que vous initialiserez et mettrez à jour correctement pour pouvoir afficher ce score à la fin.

Q4. Générez une boucle de jeu qui :

- Demande au joueur à quel niveau il veut jouer.
- Charge le niveau en question.
- Laisse le joueur jouer jusqu'à ce qu'il gagne.
- Affiche le nombre de pas total fait.
- Recommence.

Exercice 4 : Sauvegarde du score

Dans cette partie, vous allez générer des fichiers nommés `score_nb` où `nb` sera remplacé par le numéro du niveau. Ce fichier contiendra simplement un seul entier correspondant au meilleur score (en nombre de pas) obtenu pour le niveau numéro `nb`.

Q1. Écrivez une fonction `int lecture_du_score (int quel_niveau);` qui prend un entier correspondant au numéro du niveau et lit dans le fichier score correspondant l'entier correspondant au meilleur score. La fonction retourne ce dernier. Attention : Si vous essayez de lire un fichier qui n'existe pas, vous risquez une petite erreur.

Q2. Écrivez une fonction `void ecriture_du_score (int quel_niveau, int score);` qui prend un entier correspondant au numéro du niveau et un entier correspondant à un score et qui écrit ce score dans le fichier score du niveau correspondant.

Q3. Adaptez votre boucle de jeu pour qu'à la fin le meilleur score soit affiché et s'il a été amélioré alors le fichier score correspondant doit être mis à jour.

Exercice 5 : Plusieurs meilleurs scores

Dans cette partie, vous allez sauvegarder plusieurs meilleurs scores. Les fichiers utilisés pour enregistrer ces scores seront nommés `score_multi_nb` où `nb` sera remplacé par le numéro du niveau.

Le fichier aura typiquement la forme suivant (mais vous êtes libre de lui proposer une autre forme si vous le désirez, typiquement en écrivant au début du fichier le nombre de lignes de score comme dans les fichiers niveaux) :

```
7 baste
13 beaufils
15 secq
```

Dans cette parie il est à vous de proposer une manière de structurer vos données en mémoire. Une manière de faire et d'avoir une structure dans laquelle vous allez stocker une ligne du fichier score/nom puis un tableau de pointeur vers ces structures, tableau de taille assez grande pour stocker toutes les lignes. Ceci n'est qu'une proposition et vous êtes libre d'utiliser la méthode qui vous semble la plus pertinente.

Q1. Écrivez une fonction `char* nom_du_joueur (void);` qui demande au joueur d'entrer son nom/pseudo et qui retourne une chaine de caractère (donc un tableau de `char`) contenant ce nom. On pourra typiquement forcer ce nom à faire au plus 8 caractères.

Q2. En fin de partie, demandez au joueur d'entrer son nom et ajoutez son score dans le fichier des scores du niveau correspondant.

Q3. Faites en sorte que ce fichier de score soit trié par ordre croissant du score.

Q4. Faites en sorte qu'un même pseudo n'apparaisse qu'une seule fois, les moins bons scores de cette personne devant être effacés.

Q5. Faites en sorte que si le score obtenu en fin de jeu n'est pas au moins assez bon pour être dans le top 5, le jeu ne demande même pas le nom au joueur. Gardez au plus 5 meilleurs scores dans le fichier de sauvegarde.

Exercice 6 : Historique de partie

Lorsque vous jouez, parfois vous vous trompez. Dans ces cas-là, il est très pratique de pouvoir revenir en arrière. Dans cette partie, vous allez implémenter une structure de donnée nommée une pile (ou LIFO). Vous avez déjà vu cette structure de donnée en POO/COO. Dans cette structure de donnée, vous avez un tableau, prévu assez grand, et vous avez un indicateur de combien d'éléments sont dans cette pile. Vous pouvez alors ajouter un élément dans cette pile. Cet élément se placera dans le tableau après les éléments déjà enregistrés dans la pile. Le compteur de taille de pile augmentera en conséquence. Vous pouvez aussi retirer un élément de cette pile. Cet élément sera retiré de la pile et retourné par la fonction. De plus, le compteur de taille de pile diminuera en conséquence.

L'état de votre jeu à un moment donné est stocké dans une variable (disons `niveau`) de type `niveau_t`. Pour pouvoir conserver l'historique, ce que vous devez faire est de créer à chaque fois que vous vous déplacez une copie de l'état du jeu (donc de niveau) avant votre déplacement. C'est cette copie que vous allez placer dans la pile.

Q1. Écrivez une fonction `niveau_t* copie_du_niveau (niveau_t* niveau);` qui prend en argument un pointeur vers une structure de type `niveau_t` et retourne un pointeur vers une nouvelle structure `niveau_t` qui est une copie de la première. Attention à bien faire une copie complète, en particulier pour l'attribut `terrain`.

Q2. Créez une structure `historique_t` pour représenter votre pile. Comme toujours pour les structures, créez une fonction pour créer cette structure, une pour l'initialiser, une pour l'afficher et une pour libérer la mémoire une fois qu'elle ne sert plus.

Attention : dans votre structure vous devez avoir un attribut qui est un tableau de pointeurs vers des éléments de type `niveau_t`. Son type sera donc `niveau_t** tableau`, l'une des étoiles pour dire que c'est un tableau, et l'autre pour dire que le tableau contient des pointeurs. Je vous conseille fortement de faire un dessin de votre mémoire pour ne pas vous mélanger.

Q3. Écrivez une fonction `void sauvegarde_un_coup (historique_t* hist, niveau_t* niveau);` qui vous permet d'enregistrer un coup dans votre historique. N'oubliez pas de mettre à jour le compteur de taille de pile.

Q4. Écrivez une fonction `niveau_t* coup_precedent (historique_t* hist);` qui vous retourne le coup le plus récent présent dans la pile. N'oubliez pas de mettre à jour le compteur de taille de pile. Attention : Si la pile est vide, la fonction retournera le pointeur `NULL` (qui est un nom pour désigner `0`).

Q5. Modifier votre boucle de jeu pour que lorsque vous utilisez la touche `'a'`, vous annuliez le dernier coup. Attention, cette modification peut impliquer de modifier d'autres fonctions, voire même de rajouter un argument à certaines fonctions. Ce n'est pas un problème.

Exercice 7 : Extension (bonus)

Dans cette partie, vous êtes libre de me proposer d'autres améliorations qui vous semblent pertinantes. (Comprenez que j'ai plus d'idées, mais vous en avez surement plein d'autres à ajouter.)