Rapport de projet
Programmation impérative

Lilian MALLARDEAU

Semestre 1

Table des matières

Repository GitLab

Vous pouvez retrouver si vous le souhaitez mon projet sur le GitLab d'ARISE :

`https://git.iiens.net/mallarde2019/projet-ipi-s1`

Sujet et choix de développement

Le sujet du projet consistait à écrire un interpréteur et un débogueur pour un langage de programmation appelé p2d.

Un programme dans ce langage est défini par une grille de caractères de largeur et de longueur donnée, chaque caractère représentant une action. Un curseur se déplace sur cette grille en fonction des instructions données par le caractère sous le curseur, dans une des huit directions possibles.

En fonction des instructions données par chaque caractère, le programme interagit avec une pile d'entiers (par exemple, dépile deux entiers et empile leur somme, ou duplique le dernier entier de la pile).

Afin de pouvoir les réutiliser pour d'autres projets plus tard, et également par soucis de clareté du code, j'ai choisi de développer des modules pour les piles, les matrices et les curseurs.

Modules développés

Afin de maximiser la portabilité de mes modules, j'ai préféré non pas créer un module de pile d'entiers, ou encore de matrice de caractères, mais j'ai défini dans chaque interface de module un type `stack_elem` ou `matrix_elem`, qui permet de changer le type des éléments si nécessaire.

```
typedef int stack_elem;
typedef char matrix_elem;
```

Pile

Une pile est implémentée comme une liste chainée. Un élément de la pile est donc défini comme une structure, contenant la valeur de l'élément en question, et un pointeur vers l'élément suivant.

La pile elle-même est quant à elle définie comme étant un pointeur vers un maillon, premier élément de la pile.

```
typedef struct item item;
typedef item* stack;
struct item {
    stack_elem head;
    stack tail;
};
```

J'ai également défini quelques fonctions et procédures sur ces piles, notamment `push`, `pop`, `create_stack`, et `is_stack_empty`, afin de faciliter leur utilisation dans le projet.

Matrice

Une matrice est définie comme une structure, contenant deux entiers m et n représentants la taille de la matrice, et un tableau en 2 dimensions de taille $m \times n$ (qui sera alloué à l'aide de malloc dans la fonction de création de matrice).

```
typedef struct matrix matrix;
struct matrix {
    int m, n;
    matrix_elem** mat;
};
```

De même que pour les piles, certaines fonctions ont été implémentées afin de faciliter l'utilisation du module, notamment `create_matrix(int n, int m)`, `copy_matrix(matrix)` et `print_matrix(matrix)`.

Curseur

Le curseur est modélisé par une structure qui contient deux entier x et y et une direction dir.

```
typedef struct cursor cursor;
struct cursor {
    int x, y;
    enum direction dir;
};
```

La direction est définie en tant qu'énumération, ce qui permet facilement de tourner de 45° à gauche ou à droite en enlevant respectivement 1 ou -1 à la valeur actuelle, et en appliquant un modulo 8 au résultat.

```
typedef enum direction direction;
enum direction {up = 0, up_right, right,
   down_right, down, down_left, left, up_left};
```

La fonction `void move_forward(cursor* cur, int n, int x_max, int y_max)` permet d'avancer dans la direction courante de n cases, en prennant en considération la taille de la matrice pour ne pas dépasser (grille torique).

J'ai également défini les procédures `rotate_cw`, `rotate_ccw` et `set_random_dir`, qui permettent respectivement de tourner à droite ou à gauche de 45°, et de chosir une direction au hasard.

Architecture du programme

En s'appuyant sur ces modules, le programme (que ce soit l'interpréteur ou le débogueur) doit d'abord ouvrir le fichier .p2d, récuperer la taille $n \times m$ du programme, créer une matrice de taille $n \times m$ et la remplir avec les caractères lus. J'ai donc décidé de construire un fichier `main.c`, qui, une fois compilé en exécutable `p2d`, reçoit en argument le programme à exécuter, lit les dimensions, crée la matrice et la remplit avec les caractères du programme p2d, puis lance le débogueur si la variable d'environnement `P2D_DEBUG_MODE` est définie, sinon l'interpréteur.

Les fonctions nécessaires à l'interpréteur et au débogueur sont donc réparties respectivement dans les fichiers `interpreter.c` et `debugger.c`, qui sont linkés avec l'exécutable `p2d` lors de la compilation avec `make`.

Afin de pouvoir lancer le débogueur facilement, j'ai écrit un wrapper `debugp2d` en bash, qui définit la variable `P2D_DEBUG_MODE`, et qui lance `p2d` en lui passant les arguments reçus :

```
export P2D_DEBUG_MODE=1
./p2d "$@"
```

Ouverture du fichier et récupération des dimensions

J'ai utilisé un flux libc pour l'ouverture et la lecture du fichier p2d. La première ligne est lue grâce à la fonction `getline` (qui permet d'obtenir un pointeur vers la chaine de caractères correspondante), et on utilise ensuite `sscanf` pour récupérer les dimensions m et n en `int`.

Il a ensuite suffit de créer une matrice aux dimensions $m \times n$, puis de la remplir caractère par caractère à l'aide de 2 boucles `for` imbriquées et de la fonction `getc`, qui lit un caractère et avance le curseur.

Interpréteur

Par soucis de praticité, notamment pour le développement du débogueur, j'ai défini une fonction

```
void one_step(matrix*, cursor*, stack*)
```

qui fait avancer le programme d'une étape, en mettant à jour le curseur, la pile d'entiers, et si nécessaire la matrice. On distingue chaque commande avec un `switch` sur le caractère courant, et on effecue les actions nécessaires sur la pile d'entiers, le curseur et la matrice en fonction du caractère.

On définit ensuite la fonction `void interpreter(matrix prog)`, qui reçoit une matrice correspondant à un programme p2d valide et qui, après

avoir initialisé le curseur et la pile, lance le programme jusqu'à sa terminaison à l'aide d'une boucle `while` :

```
while (prog.mat[cur.y][cur.x] != '@') {
  one_step(&prog, &cur, &prog_stack);
}
```

Problèmes rencontrés et solutions

Problème d'indices

Lors du développement de l'interpréteur, j'ai instinctivement écrit `mat[cur.x][cur.y]` pour accéder au caractère de la matrice en position (x, y). Cependant, étant donnée la façon dont j'ai implémenté le type matrice (c'est-à-dire comme une matrice en mathématiques, m_{ij} donnant l'élément à la $i^{\text{ème}}$ ligne et à la $j^{\text{ième}}$ colone), la valeur `x` du curseur correspond au numéro de colone, et la valeur `y` du curseur correspond au numéro de ligne.

Il m'a alors seulement suffit d'inverser `cur.x` et `cur.y` dans tout le code source de l'interpréteur pour palier à cette erreur.

Problème avec les commandes 'p' et 'g'

J'ai également rencontré un problème avec les commandes liées aux caractères 'p' et 'g', cependant je ne m'en suis rendu compte que tard, lors du test avec `sieve.p2d`. En effet, tous les programmes fournis en exemple fonctionnaint, sauf celui-là qui imprimait une grille infinie de 0. Effectivement, ce programme est le seul à contenir des caractères 'p' et 'g'. J'ai pensé qu'il serait plus pratique de d'abord développer totalement le débogueur, afin de m'en servir pour corriger cette erreur.

J'ai donc remarqué après développement de ce dernier que lors des commandes 'p' ou 'g', les coordonnées x et y du caractère ciblé sont inversées : après inversion de l'ordre de dépilement de x et y dans la pile, le problème était corrigé.

Débogueur

Le sujet nous demandait, après avoir réalisé l'interpréteur, de développer un débogueur pour programmes `.p2d`. Le débogueur demanda plus de réflexion et de conception en amont du développement que l'interpréteur. Celui-ci devait d'abord afficher la matrice du programme courant, entourées de repères et d'indicateurs permettant d'afficher la position actuelle du curseur, puis l'état de la pile (ainsi que, pour chaque entier, le caractère correspondant à son code ASCII entre parenthèses) et enfin les coordonnées du curseur.

Le débogueur attend ensuite des commandes de l'utilisateur, parmis :

- **step** : avance d'une étape dans le programme
- **step n** : avance de n étapes dans le programme, ou jusqu'au prochain breakpoint
- **run** : exécute le programme jusqu'à la fin, c'est-à-dire jusqu'à renconter le caractère '@' (ou jusqu'à renconter un breakpoint)
- **restart** : retourne à l'état initial, avant exécution
- **quit** : quitte le débogueur
- **prec** : recule d'une étape
- **prec n** : recule de n étapes
- **breakpoint x y** : ajoute un breakpoint aux coordonnées (x, y)
- **removebp x y** : supprime le breakpoint aux coordonnées (x, y) s'il existe

De plus, si l'utilisateur entre une ligne vide (ou seulement des espaces), le débogueur doit répéter la dernière commande.

Enregistrement de l'historique des étapes

Afin de pouvoir revenir aux étapes précedentes (commandes `prec`, `prec n` ou `restart`), il est nécessaire d'enregistrer un historique de chaque étape. J'ai donc pour cela défini une structure dédiée :

```
typedef struct prog_step prog_step;
struct prog_step {
    matrix mat;
    cursor cur;
    stack stack;
};
```

Un élément `prog_step` représente une étape du programme, avec l'état de la matrice, du curseur et de la pile au moment de l'enregistrement.

Ainsi, un historique complet des étapes du programme est incrémenté dans un tableau de `struct prog_step` défini ainsi :

```
prog_step* saved_steps = (prog_step*)
   calloc(HISTORY_BUFFER_SIZE, sizeof(prog_step));
```

La constante `HISTORY_BUFFER_SIZE` est définie en début de fichier à l'aide d'un `#define` : il s'agit de la taille originale du tableau lors de sa création.

Pour facilier l'enregistrement d'une étape, j'ai défini la fonction `save_step` suivante :

```
void save_step(int index, matrix mat, cursor cur,
    stack prog_stack, prog_step* saved_steps[],
    size_t* size);
```

À l'aide des fonctions `copy_matrix` et `copy_stack` des modules `matrix` et `stack`, la fonction `save_step` crée un nouvel élement `prog_step`, stocke une copie de la matrice, du curseur et de la pile actuels, et ajoute cet élément au tableau `saved_steps` en position `index`. Si la taille du tableau `*size` est inférieure à `index`, `save_step` agrandit le tableau `saved_steps` de `HISTORY_BUFFER_SIZE` cases supplémentaires, en lui réallouant la mémoire nécessaire à l'aide de `realloc` :

```
while (*size <= index) {
    *saved_steps = realloc(*saved_steps, (*size +
        HISTORY_BUFFER_SIZE) * sizeof(struct
        prog_step));
    if (*saved_steps != NULL) {
      *size += HISTORY_BUFFER_SIZE;
    } else {
      fprintf(stderr, "Error while reallocating
          program history array. Exiting.");
      exit(2);
    }
  }
```

J'ai également défini une fonction `restore`, qui prend en argument un pointeur vers la matrice du programme, le curseur actuel, la pile actuelle, le tableau `saved_steps`, ainsi que l'index de l'étape à restorer, et qui copie la matrice, le curseur et la pile sauvegardés à la place de ceux actuels, afin de revenir à l'étape souhaitée.

Gestion des breakpoints

Un breakpoint n'étant rien d'autre qu'un point, je les ai défini comme une structure contenant deux entiers x et y :

```
typedef struct breakpoint breakpoint;
struct breakpoint {
    int x, y;
};
```

De même que pour l'historique, j'ai déclaré au début du débogueur un tableau

```
breakpoint* breakpoints = (breakpoint*)
    calloc(BREAKPOINTS_BUFFER_SIZE,
    sizeof(breakpoint));
```

destiné à stocker les breakpoints définis par l'utilisateur. La taille initiale `BREAKPOINTS_BUFFER_SIZE` est définie en tant que constante en début de fichier avec un `#define`.

La fonction

```
add_breakpoint(int x, int y, breakpoint* breakpoints[],
    size_t* n_breakpoints, size_t* buffer_size);
```

permet d'ajouter le breakpoint (x, y) au tableau `breakpoints` s'il n'y est pas déjà présent, en réallouant la mémoire disponible pour le tableau si nécessaire, de la même façon que dans la fonction `save_step`.

J'ai aussi défini des fonctions `remove_breakpoint` et `get_breakpoint` permettant de rechercher ou de supprimer des breakpoints.

Enfin, le débogueur devant à chaque étape vérifier si le curseur se trouve sur un breakpoint (lors de `run`, `step n` ou `prec n`), j'ai défini également la fonction `check_breakpoint`, qui prend en paramètres `x`, `y`, le tableau `breakpoints` ainsi que le nombre de breakpoints, et qui renvoit 1 si le breakpoint (x, y) existe dans le tableau, et 0 sinon.

Gestion de l'affichage à la console

Afin d'actualiser l'état du programme à l'écran à chaque étape, j'ai défini la fonction `print_screen` suivante :

```
void print_screen(matrix prog, cursor cur,
    stack prog_stack);
```

Cette dernière reçoit la matrice, le curseur et la pile actuels, et imprime l'état actuel du programme sous la forme souhaitée, avec des repères et indicateurs pour afficher la position actuelle du curseur.

En utilisant les séquences d'échappement, j'ai pu afficher les indicateurs '>' et 'v' en couleur, et afficher le caractère courant en couleurs inversées, afin de le faire ressortir et le rendre plus visible.

Pour nettoyer l'écran entre chaque itération avant d'appeler la fonction `print_screen`, j'ai créé une fonction `clear_screen`, définie ainsi :

```
void clear_screen() {
    system("@cls||clear");
}
```

Bien qu'effectivement très peu propre, cette astuce fonctionne sur Windows, Linux et macOS : sur Windows, la commande `cls` est exécutée, avec le `'@'` en début de commande qui permet de ne pas afficher la commande avant de

l'exécuter. Sur Linux et macOS, la commande `@cls` n'existe pas, l'expression `@cls` renvoit alors une erreur, et la commande `clear` est exécutée grace au OU logique `||`.

Récupération de l'entrée utilisateur et de la commande

Après avoir défini tous les types et toutes les fonctions nécessaires, le débogueur en lui-même reçoit la matrice du programme à déboguer, doit afficher l'état initial du programme, et récupérer la commande de l'utilisateur. On doit aussi stocker la commande précedente, si l'utilisateur entre une ligne vide.

Pour ce faire, j'ai défini deux buffers `char userinput[256]` et `char last_command[256]`, et utilisé la fonction `fgets` pour récupérer l'entrée de l'utilisateur et la stocker dans `userinput` :

```
fgets(userinput, 256, stdin);
```

L'extraction des variables nécessaires de la chaine entrée par l'utilisateur fut une difficulté : comment savoir, par exemple, si l'utilisateur a entré la commande `step` ou `step n` ? Après de longs moments de recherche sur le web et dans le manuel Linux, je suis tombé un peu par hasard, dans le manuel de `scanf`, sur la valeur de retour de `sscanf` : la fonction retourne le nombre de variables correctement assignées. J'ai donc utilisé la fonction `sscanf`, qui s'utilise de la même façon que `scanf`, et qui permet d'extraire les variables voulues d'une chaine de caractères. J'ai défini en début de programme les variables tampons `char command[256]`, `int a, b` et `int nbr_args`. J'utilise ensuite la fonction `sscanf` de la manière suivante :

```
nbr_args = sscanf(userinput, "%s %d %d", command,
    &a, &b);
```

Ainsi, si l'utilisateur entre une commande de la forme `step n`, `nbr_args` vaudra 2, le buffer `command` contiendra la chaine `"step"`, la valeur de `n` sera stockée dans `a` et `b` restera inchangée. Sachant que lorsque `sscanf` n'arrive à assigner aucune variable avec succès (par exemple, lorsque l'utilisateur entre une ligne vide, ou seulement des espaces) sa valeur de retour vaut -1, je peux considérer l'ancienne commande à la place lorsque c'est nécessaire :

```
if (nbr_args == -1) {
    nbr_args = sscanf(last_command, "%s %d %d",
        command, &a, &b);
    strcpy(userinput, last_command);
}
```

Je peux ensuite faire des tests avec `strcmp` sur la chaine `command` et exécuter les actions nécessaires :

```
if (strcmp(command, "step") == 0) {
    if (nbr_args == 2 && a >= 0) {
        /* On avance de a etapes */
    } else {
        /* On avance d'une seule etape */
    }
} else if (strcmp(command, "run") == 0) {
    /* ... */
} else if (strcmp(command, "restart") == 0) {
    /* ... */
} else if /* ... */
    /* ... */
```

Exécution des commandes

Ayant maintenant les paramètres nécessaires aux commandes dans les variables `a` et `b`, et en ayant recours aux fonctions définies plus haut, il est plus aisé d'exécuter les actions correspondants à chaque commande.

Pour les commandes `step` et `step n`, j'ai défini une fonction `next`, qui prend tous les arguments nécessaires (pointeur vers la matrice, le curseur, la pile et l'index de l'étape actuelle, et le tableau `saved_steps` ainsi que sa taille), qui utilise la fonction `next_step` définie dans l'interpréteur pour avancer d'une étape, qui enregistre cette étape avec `save_step`, puis qui incrémente `index`. Il suffit donc d'appeler cette fonction une fois pour avancer d'une étape. Pour la commande `step n`, on appelle `next` n fois dans une boucle `for`, mais il faut vérifier à chaque fois que le curseur ne se trouve pas sur un breakpoint :

```
for (int i = 0; i < a; i++) {
  next(&index, &prog_mat, &cur, &prog_stack,
      &saved_steps, &history_buffer_size);
  if (check_breakpoint(cur.x, cur.y, breakpoints,
      n_breakpoints))
    break;
}
```

Pour revenir à l'étape précédente, j'ai défini, de même que la fonction `next`, une fonction `prev` qui prend les mêmes arguments que `next` plus un pointeur `index` vers l'indice de l'étape actuelle, et qui restore l'étape précédente à l'aide de la fonction `restore` définie plus haut, si l'index actuel est supérieur à 1 (afin d'éviter une erreur de segmentation). Dans ce cas, la fonction `prev` décrémente aussi la variable `*index`.

L'exécution des commandes `prec` et `prec n` s'effecue exactement de la même façon, sauf qu'on appelle la fonction `prev` à la place de la fonction `next`.

La commande `run` est implémentée à l'aide d'une boucle `do ... while` et d'un appel à `next` :

```
do {
    next(&index, &prog_mat, &cur, &prog_stack, &saved_steps,
        &history_buffer_size);
} while (!check_breakpoint(cur.x, cur.y, breakpoints,
    n_breakpoints) && prog_mat.mat[cur.y][cur.x] != '@');
```

Les commandes relatives aux breakpoints appellent seulement la fonction `add_breakpoint` ou `remove_breakpoint` définie plus haut, avec les paramètres nécessaires.

Quant aux commandes `restart` et `quit`, il suffit respectivement d'appeler la fonction `restore` avec l'index 0 et de réinitialiser la variable `index` à 0, ou d'utiliser `exit(0)`.

Limites du programme

Une des limites évidentes de ce programme vient du fait que le type `int` ne peut gérer des entiers infiniments grands : il sera donc impossible d'exécuter un programme .p2d correctement si au moins une de ses dimensions dépasse la valeur de la constante `INT_MAX` (dépend des implémentations, souvent 2147483647).

De même, le débogueur enregistrant chaque étape du programme dans un tableau, et incrémentant l'indice correspondant à l'étape actuelle, l'historique sera progessivement effacé pour enregistrer les étapes suivantes si un programme a plus de `INT_MAX` étapes.

Bien que ce soit peu probable, il n'est pas non plus possible d'ajouter plus de `INT_MAX` breakpoints différents, le `INT_MAX+1`$^{\text{ième}}$ écrasant le premier.

Enfin, les buffers destinés à recevoir les commandes utilisateur ayant été définis avec une taille donnée (256 caractères), le débogueur ne peut gérer une entrée utilisateur de plus de 256 caractères (bien que ce soit peu probable).