

# Rapport IPI

Gautier POURSIN

6 Janvier 2020

## Introduction

Dans ce projet, nous devons implanter un interpréteur et un débogueur dans un langage de programmation appelé prog2D: cela correspond à une grille de caractères où un curseur se déplace dans la grille. Chaque caractère correspond à une instruction donnée. Le curseur peut se déplacer dans 8 directions possibles et le curseur se déplace toujours dans la même direction, tant qu'un caractère indiquant une autre direction n'est pas rencontré. En parallèle du déplacement du curseur, nous implémenterons une pile d'entier qui s'empile et se dépile en fonction des instructions données. Ensuite, le débogueur permettra à l'utilisateur d'entrer différentes commandes et d'avoir l'état du programme à chaque étape.

Dans un premier temps, nous allons nous focaliser sur les structures de travail choisies, puis sur l'interpréteur.

Ensuite, nous allons nous intéresser au débogueur et quelles ont été les difficultés rencontrées.

## Sommaire

### 1 Les structures choisies

### 2 Interpreteur

#### 2.1 Rentrer le fichier dans une matrice

#### 2.2 Fonctions de déplacement dans la matrice

#### 2.3 Différents cas en fonction des caractères rencontrés

### 3 Débogueur

#### 3.1 Création des différentes instructions

#### 3.2 Pas de solution trouvée pour "entrée"

### 4 Améliorations à faire

### 5 Manuel Utilisateur

## 1 Les structures choisies

```
struct position{  
    int x,y;  
    // x:numero de lignes, y:numero de colonnes  
};  
typedef struct position position;
```

Figure 1: La structure correspondant à la position du curseur

Cette structure est composée de 2 entiers **x** et **y** correspondant respectivement au numéro de ligne et de colonnes où le curseur se situe dans la matrice. Cela permet de suivre l'avancement du curseur. Il sera nommé **position**.

```
//w:nbre de colonne, h=nbredeligne  
struct array2D{  
    int h,w;  
    int **content;  
};  
typedef struct array2D array2D;
```

Figure 2: La structure correspondant à une matrice

Cette structure est composée de 2 entiers **w** et **h** correspondant au nombre de lignes et de colonnes de la matrice. De plus, elle contient un pointeur de pointeur correspondant à des entiers. En effet, la matrice dans laquelle sera placé le fichier est composée de code ASCII de chaque caractère. Il sera nommé **array2D**.

```
struct stack {  
    int *t;  
    int top;  
    int size;  
};  
typedef struct stack stack;
```

Figure 3: La structure correspondant à une pile

Cette structure est composée d'un entier **top**, correspondant au haut de la pile, d'un entier **size** correspondant à la taille de la pile (j'ai choisi de travailler avec des piles qui ont des tailles variables.) et enfin d'un pointeur **t** qui correspond à l'entier pointé par la pile. Il sera nommé **stack**.

```

struct ensemble{
    position curseur;
    array2D tableau;
    int directory;
    int caractere;
    stack s;
};

typedef struct ensemble ensemble;

```

Figure 4: La structure globale nécessaire au bon déroulement du programme

Cette structure me permet de regrouper tous les éléments nécessaires pour avoir un déroulement de l'interpréteur plus simpliste, permettant d'avoir en argument qu'un élément **ensemble**, qui contient une pile, la matrice de caractère, la position du curseur, le caractère sur lequel on se situe et la direction du déplacement. La direction est représentée par un entier, variant entre 0 et 7.

## 2 Interpréteur

### 2.1 Rentrer le fichier dans une matrice

Pour réaliser l'interpréteur, la première chose à faire est de mettre un fichier de caractère dans une matrice. Pour cela, il faut avoir la taille de la matrice nécessaire pour y mettre tout le fichier, sans allouer de mémoire en trop. Il faut donc avoir la taille exacte du fichier. Cette taille (le nombre de lignes et de colonnes) se trouve sur la première ligne du fichier. Il faut donc extraire la première ligne seulement à l'aide de la fonction **fgets**, qui renvoie une chaîne de caractère. C'est pourquoi il est nécessaire de repérer le premier espace dans la chaîne de caractère pour avoir ensuite d'un côté, le nombre de lignes, et de l'autre, le nombre de colonnes. Cela peut être fait avec la fonction **premierespacedanschaîne**. Ensuite, à l'aide d'une fonction d'extraction de chaîne, notée **extractionchaîne** dans mon programme, je peux extraire la première partie de la chaîne (ie la partie de la première ligne avant l'espace) afin d'avoir le nombre de colonnes. La chaîne extraite étant un nombre noté en caractère, il faut le convertir en entier avec la fonction **atoi**, utilisée dans la fonction **conversion**. Avec ces différentes fonctions, j'obtiens donc la taille nécessaire pour ma matrice.

Ensuite, l'objectif était de mettre le fichier dans la matrice à l'aide de la fonction **ouverturefichier**. Il faut d'abord allouer la mémoire nécessaire pour la matrice, ce qui est possible grâce à la fonction **create**, prenant en paramètre la taille de la matrice. Il faut ensuite remplir la matrice en mettant un caractère par case. J'ai 2 possibilités qui s'offrent à moi: utiliser la fonction **fgets**, qui permet d'extraire le fichier ligne par ligne puis ensuite de répartir la ligne dans

une ligne de la matrice, ou utiliser la fonction **fgetc**, qui permet d'extraire le fichier caractère par caractère. J'ai essayé dans un premier temps la fonction **fgets**. Cependant, je n'ai pas réussi à repartir la ligne du fichier dans la ligne de la matrice. Cette technique s'est avérée être un échec. C'est pourquoi j'ai opté pour la fonction **fgetc**, avec une matrice intermédiaire comportant le code ASCII des sauts de lignes, indésirables ici puis la matrice finale qui est la matrice intermédiaire sans les sauts de lignes.

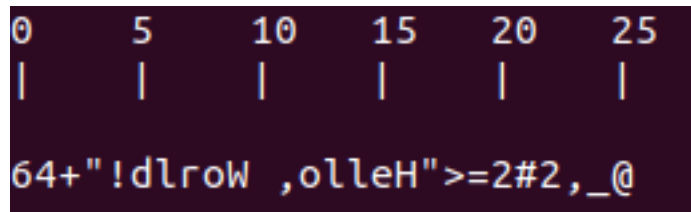
```
array2D fichierdansmatrice=create(conversion(nombredecolonne),conversion(nombrede
edeligne));
// on crée l'espace pour la matrice
array2D tmp=create(conversion(nombredecolonne)+1,conversion(nombredeligne));
//on crée une matrice temporaire

for(int k=0; k<conversion(nombredeligne); k+=1){
    for(int j=0; j<conversion(nombredecolonne)+1; j+=1)
        tmp.content[k][j]=fgetc(fichier);
}

for(int k=0; k<conversion(nombredeligne); k+=1){
    for(int j=0; j<conversion(nombredecolonne); j+=1)
        fichierdansmatrice.content[k][j]=tmp.content[k][j]; //on remplit la matrice
sans les retour à la ligne et sans la première ligne (celle de la taille du fic
hier) puisqu'on l'a extraite avec le fgets. Avec fgetc, on remplit chaque case a
vec un caractère.
}
```

Figure 5: Partie des lignes de code pour rentrer le fichier dans une matrice

Cependant, j'obtiens une matrice d'entiers et non de caractères. La matrice est remplie des valeurs ASCII de chaque caractère, il faudra donc lors de mon switch travailler sur les entiers ASCII et non les caractères. Ma fonction **print-array2D** affiche cependant bien la matrice de caractères, comme attendu dans l'énoncé.



```
0      5      10     15     20     25
|      |      |      |      |      |
64+\"!dlrow ,olleH\">=2#2, _@
```

Figure 6: Affichage du fichier helloworld.txt

## 2.2 Fonctions de déplacement dans la matrice

Pour cette partie, je vais utiliser la structure **ensemble** créée initialement. Il y a 8 déplacements possibles, ces 8 déplacements sont représentés par des entiers, allant de 0 à 7. Voici un exemple de fonction de déplacement, ici le déplacement à droite:

```
ensemble deplacementdroit(ensemble e){
    e.curseur.y=(e.curseur.y+1);
    if (e.curseur.y > e.tableau.w) e.curseur.y = 0;
    e.directory=2;
    return e;
}
```

Figure 7: Déplacement à droite

Le curseur se déplace horizontalement d'une case donc seule la coordonnée selon y varie. Bien sûr, il faut veiller à ne pas être dans une case hors de la matrice. Dans ce cas, on revient à la première colonne. S'il n'y a pas d'indication contraire, on conserve le déplacement à droite donc l'entier correspondant à la direction reste inchangée.

J'ai ensuite créé une fonction **deplacement** qui prend en argument un **ensemble** et renvoie tous les déplacements possibles en fonction de la direction qui est indiquée dans l'**ensemble**.

## 2.3 Différents cas en fonction des caractères rencontrés

Pour cette partie, j'ai créé une unique fonction **differentcas** qui prend en argument un ensemble e et j'ai réalisé un **switch** en fonction des caractères. Dans mon switch, j'ai rentré les codes ASCII des caractères puisque ma matrice est remplie des codes ASCII. J'ai aussi réalisé des fonctions intermédiaires telles que les fonctions **empile**, **depile**, qui permettent d'empiler et dépiler une pile ainsi que les fonctions **quotient** et **reste**, qui donnent le quotient et le reste d'une division euclidienne.

```

ensemble deplacement(ensemble e){
    switch(e.directory){
    case 0:
        e=deplacementhaut(e);
        break;
    case 1:
        e=deplacementdiaghautdroit(e);
        break;
    case 2:
        e=deplacementdroit(e);
        break;
    case 3:
        e=deplacementdiagbasdroit(e);
        break;
    case 4:
        e=deplacementbas(e);
        break;
    case 5:
        e=deplacementdiagbasgauche(e);
        break;
    case 6:
        e=deplacementgauche(e);
        break;
    case 7:
        e=deplacementdiaghautgauche(e);
        break;};
    return e;
}

```

Figure 8: Ensemble des déplacements

### 3. Débogueur

#### 3.1 Création des différentes instructions

Pour cette partie, j'ai réutilisé le programme de l'interpréteur, en modifiant simplement mon **main**. C'est à dire qu'au lieu d'avoir un **while** qui avance automatiquement, je demande une commande pour avancer de **n étapes**, pour **quitter** le programme, pour **recommencer** le programme, etc. Voici par exemple la partie du programme pour l'instruction **step**. La technique utilisée consiste à comparer la chaîne de caractère rentrée par l'utilisateur et celles qui sont dans mon **main**: chacune procédant à des actions différentes.

```

while (e.caractere !=64){
    printf("rentrer une instruction:\n");
    scanf("%s",c);
    if (strcmp(c,"step")==0){
        e.caractere=e.tableau.content[e.curseur.x][e.curseur.y];
        e=differentcas(e);
        e=deplacement(e);
        printf("%d\n",e.directory);
        printf("x=%d;y=%d\n",e.curseur.x,e.curseur.y);
        printf("%c\n",e.caractere);
        printstack(e.s);
    }
}
rentrer une instruction parmi celles ci : step, run, quit, restart, stepn.
step
0      5      10     15
|      |      |      |
|      |      |      |

&gt;=1-=v v * _; .v
^      _;>$=^ [
        @,*< 5
          2 =
          \

2
x=0;y=2
=
5 ()|5 ()|

```

Figure 9: Fonction pour avancer d'une étape

### 3.2 Pas de solution trouvé pour "entrée"

Concernant l'instruction **entrée**, je n'ai pas réussi à trouver une solution pour réaliser cette instruction. En effet, ma technique consiste à comparer une chaîne de caractère avec une autre. Cependant, je n'ai pas réussi à traduire la fonction **entrée** en chaîne de caractère. Je pensais utiliser son code ASCII correspondant, mais cela fut un échec.



## 4. Améliorations à faire

Dans ce projet, je pense que j'alloue beaucoup de mémoire en excès. par exemple, pour rentrer le fichier dans la matrice, j'utilise une matrice temporaire ce qui est à mon avis évitable. De plus, je remplis la matrice avec des entiers ASCII, alors que la fonction **fgets** la remplirait de caractères. J'ai donc mal utilisé cette fonction. Enfin, ma technique pour le débogueur n'est pas optimale puisque elle s'avère compliqué, je fais tout dans le **main**. J'aurais du faire une fonction à part qui n'utilise pas la fonction **strcmp** et ainsi je pourrais peut être avoir toutes les instructions souhaitées. La dernière amélioration à faire concerne l'affichage. En effet, je n'ai pas réussi à afficher les coordonnées sur le coté de la matrice ni les flèches indiquant le position du curseur. Pour cela, il faudrait augmenter la taille de ma matrice de 2 lignes et 2 colonnes supplémentaires pour y inclure les indications voulues.

## 5. Manuel utilisateur

### Fonctionnement de l'interpréteur

Pour lancer l'interpréteur, il faut que le texte que l'on souhaite lire soit situé dans le même répertoire que le programme **interpreteur.c**. Si on exécute le programme sans aucun argument, le programme renverra un message d'erreur. Par exemple, voici comment lancer le programme fact.txt:

```
gautierpoursin@gautierpoursin:~/Documents/Programmation_Imperative/projet_final$ gcc -Wall interpreteur.c
gautierpoursin@gautierpoursin:~/Documents/Programmation_Imperative/projet_final$ ./a.out fact.txt
```

Figure 10: Commande pour exécuter fact.txt

Si l'utilisateur lance un programme qui ne se situe pas dans le même dossier, alors le programme renverra une **Segmentation Fault**.

### Fonctionnement du débogueur

De la même manière que pour l'interpréteur, il faut que le fichier que l'on souhaite déboguer soit dans le même répertoire que le débogueur, nommé **debogueur.c**. Une fois le débogueur lancé, l'utilisateur aura le choix parmi 5 instructions : **step**, **stepn**, **run**, **restart**, **quit**. En dehors de ces 5 instructions, rien ne se passera. Pour l'instruction **stepn**, une fois cette instruction rentrée, le programme demandera à l'utilisateur de combien il souhaite avancer dans son programme. Ainsi, rentrer l'instruction **step 5 ne fonctionne pas!**

```
gautierpoursin@gautierpoursin:~/Documents/Programmation_Imperative/projet_final$ gcc -Wall debogueur.c
gautierpoursin@gautierpoursin:~/Documents/Programmation_Imperative/projet_final$ ./a.out fact.txt
0   5   10  15
|   |   |   |
|   |   |   |

& >= 1 - = v v * _ ; . v
^      _ ; > $ = ^ [
      @ , * <      5
          2 =
          \

x=5 y=15
rentrer une instruction parmi celles ci : step, run, quit, restart, stepn.
stepn
combien d'étapes souhaitez vous faire ?

```

Figure 11: Commande pour lancer le débogueur et faire l'instruction step n