Compte rendu 2i102

Language C

Projet «Morpion» Version 2.

Gabriel LEFFAD

SOMMAIRE

Introduction

Le morpion est un jeu de réflexion se pratiquant à deux joueurs au tour par tour et dont le but est de créer le premier un alignement sur une grille. Le jeu se joue généralement avec papier et crayon.

Règle du jeu

Les joueurs inscrivent tour à tour leur symbole sur une grille qui n'a pas de limites ou qui n'a que celles du papier sur lequel on joue. Le premier qui parvient à aligner cinq de ses symboles horizontalement, verticalement ou en diagonale gagne un point.
Le morpion donne un avantage assez important à celui qui commence. Des formes évoluées existent, comme le Gomoku ou la Pente, qui ajoutent à la notion d'alignement une notion de prise. Le renju prévoit des handicaps pour le joueur qui commence, ce qui permet d'équilibrer les chances des deux joueurs. Une partie dure environ une minute.

Nous allons ici permettre a l’utilisateur plus de liberté en lui proposant de choisir le nombre d’alignements et la taille de la grille.

Cahier des charge émis par Mr Foillot

micro-projet
- a rendre le 21/12/2018 + rapport final
    -> v0 v1 (v2 si prête) pour le 22/11/2018

- morpion (non graphique)
    - mode texte
    - scanf coordonnées
    - printf grille

    v0:
    - grille fixe 3x3
    - fin lorsque alignement || grille pleine
    - deux joueurs j1, j2
    - affichage de la grille tour a tour avec scanf en dessous(entrer x..)

    v1:
    - grille dynamique 3 < N < 33 (gerer les erreur user), dim choix user scanf
    - l'utilisateur doit saisir "morpion [dimension]" et le morpion apparaît (use atoi to get the integer out of the string)

    v2:
    - dim 3 <= N <= 33 && 1 < align <= 5
        -> N >= align (align = nb alignement pour gagner, donc supérieur a dimension).

Nous allons ici implémentée la version 2 du morpion.
On démarre le programme avec la commande : ./morpion [taille] [alignement]

```
xxx@xxx:~/cours/atic/c/projet$ ./morpion 3 3
 . . .
 . . .
 . . .
Joueur X, a vous de jouer ! (format : [X,X])
```

Diviser un problème en sous problèmes

Tel que nous l'avons appris en cours de 2i102, il est nécessaire d'aborder un grand problème en le divisant en sous-problèmes, cette démarche sera garante d'une flexibilité de notre code, d'une meilleure stabilité ainsi que d'une plus grande aisance lors du débuggage.

Voici donc les différentes fonctions ainsi que l'implémentation du code, permettant de construire le programme demandé.

Fonction d'affichage de la grille

```
 87 //affichage de la grille
 88 void    ft_print_grid(char **grid, int size)
 89 {
 90     int i;
 91     int j;
 92
 93     i = -1;
 94     j = -1;
 95     while (++i < size)
 96     {
 97         while (++j < size)
 98         {
 99             printf(" %c", grid[i][j]);
100         }
101         printf("\n");
102         j = -1;
103     }
104 }
```

Le choix de la structure de donnée de la grille est un tableau à deux dimensions allouées dynamiquement.

Nous allons donc effectuer une double boucle permettant de parcourir la grille ligne par ligne en affichant chaque élément particulier, ligne par ligne.

La fonction prend aussi en paramètre la taille de la grille afin de ne pas créer d'erreurs de segmentation en sortant de l'espace mémoire allouée.

Contrairement aux chaînes de caractères, les tableaux d'entiers ne possèdent pas de caractères de fin de tableau '\0'.

On initialise les variables à -1 car on incrémente dans la boucle, cette méthode sera utilisée tout au long du développement du programme.

Fonction d'initialisation de la grille avec des caractères '.'

```
106 //initialisation de la grille avec des caracteres '.'.
107 void    ft_init(char **grid, int size)
108 {
109     int i;
110     int j;
111
112     i = -1;
113     j = -1;
114     while (++i < size)
115     {
116         while (++j < size)
117             grid[i][j] = '.';
118         j = -1;
119     }
120 }
```

Le caractère principal notifiant une case vide sera : '.'. Nous parcourons donc la mémoire allouée comme dans la fonction précédente en attribuant la valeur '.' à chaque case mémoire.

Fonction de saisie du choix du joueur.

```
122 //saisi du choix choix du joueur.
123 void    ft_get_choice(int *a, int *b, char player, int size)
124 {
125     char    c;
126     while (!(*a < size && *a >= 0 && *b < size && *b >= 0))
127     {
128         printf("Joueur %c, a vous de jouer ! (format : [X,X])\n", player);
129         scanf("%i,%i", a, b);
130         while((c = getchar()) != '\n' && c != EOF);
131     }
132 }
```

A chaque tour, les joueurs fournissent deux coordonnées sur la grille, qui sont deux entiers distincts. De plus, chaque joueur sera modélisé par le caractère 'X' ou 'O' respectivement.
A chaque tour, le caractère alterne, et sera stocké dans la variable 'player'.

Cette fonction prends en argument deux adresses d'entiers, le signe du joueur, ainsi que la taille du tableau afin de vérifier que les coordonnées saisis par l'utilisateur sont bien contenues dans la mémoire allouée.

On affiche l'invitation à jouer au joueur concerné, on récupère les coordonnées à l'aide de la fonction scanf, et on vide le buffer pour éviter les boucle infinie au cas où l'utilisateur entre une string. En effet, si l'utilisateur entre une chaînes de caractères, scanf ne s'exécute plus, donc les variable pointées a et b ne change pas, on se retrouve donc dans une boucle infinie.

En vidant le buffer, scanf peut s'exécuter une nouvelle fois à chaque tour de boucle si besoin.

Fonction insertion du choix du joueur.

```
134 //insertion du choix du joueur
135 void    ft_insert_choice(char player, char **grid, int size)
136 {
137     int     i;
138     int     j;
139 
140     i = -1;
141     j = -1;
142     while (1)
143     {
144         ft_get_choice(&i, &j, player, size);
145         if (grid[i][j] != '.')
146         {
147             printf("\nErreur : case deja pleine ! Jouez une autre case ;)\n");
148             i = -1;
149             j = -1;
150         }
151         else
152         {
153             grid[i][j] = player;
154             break;
155         }
156     }
157 }
```

Ici, on va insérer le choix du joueur dans la case correspondante, sous condition que la case soit vide, sinon on affiche un message d'erreur stipulant que la case est occupée et on rappel au tour suivant la fonction de saisie du choix du joueur. On sort de la boucle lorsque le joueur a saisi des coordonnées d'une case vide, au bon format, et comprises dans la mémoire allouée.

Fonction de vérification de grille pleine.

```
159 //verification de grille pleine.
160 int     ft_check_full(char **grid, int size)
161 {
162     int i;
163     int j;
164
165     i = -1;
166     j = -1;
167     while(++i < size)
168     {
169         while(++j < size)
170             if(grid[i][j] == '.')
171                 return (0);
172         j = -1;
173     }
174     return (1);
175 }
```

Nous allons ici vérifier si la grille est pleine : on parcours la grille en contrôlant si la casse mémoire contient le caractère '.'. Si on trouve le caractère '.', alors la grille n'est pas pleine, sinon, la grille est pleine. Nous choisissons 1 et 0 comme valeur de retour car ces fonctions sont des fonctions test qui seront appelées dans les conditions à venir, signifiant vraie et faux respectivement.

Fonction de test de l’existence d’une combinaison gagnante.

(Fonction particulièrement amusante à construire ).

```
 4 //test l'existance d'une combinaison gagnante.
 5 int     ft_check_win(char **grid, char p, int size, int align)
 6 {
 7     int     i;
 8     int     j;
 9     int     tmp1;
10     int     tmp2;
11     int     cpt;
12
13     i = -1;
14     j = -1;
15     while (++i < size)
16     {
17         while (++j < size)
18         {
19             tmp1 = i;
20             tmp2 = j;
21             cpt = 0;
22             //droite
23             while (tmp1 < size && tmp1 >= 0 && tmp2 < size
24                     && tmp2 >= 0 && grid[tmp1][tmp2] == p)
25             {
26                 tmp2++;
27                 cpt++;
28                 if (cpt == align)
29                 {
30                     printf("\nPlayer (%c) win the game !\n", p);
31                     return (1);
32                 }
33             }
```

...

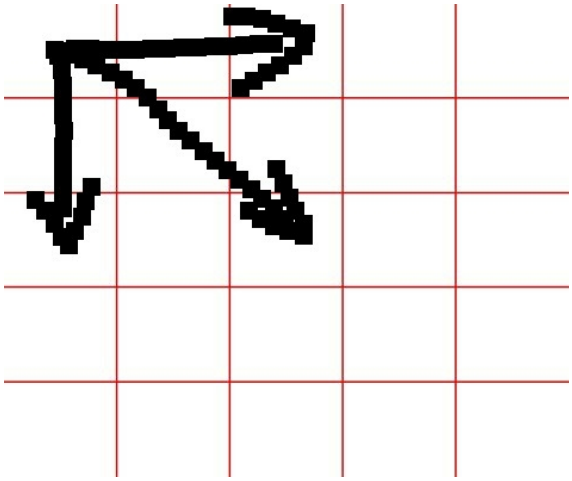
La principale problématique : comment parcourir la grille pour trouver les alignements gagnant sans en rater aucun ?

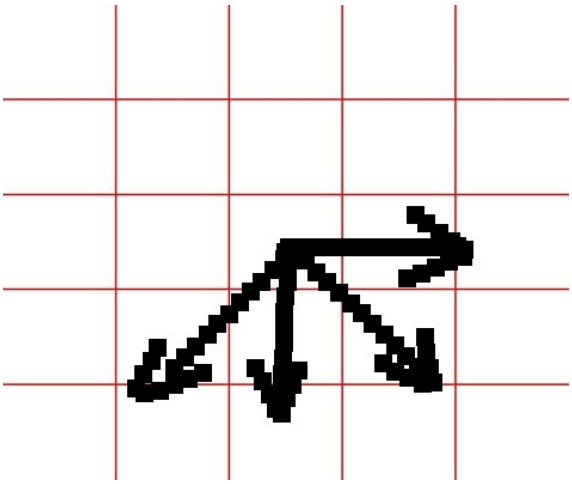
Réfléchissons à l’angle de vue que l’on va adopter, en réutilisant le principe de décomposition en sous-problèmes.

On a une grille. Donc, on a des lignes, des colonnes, et des diagonales. Donc on a des directions : haut, bas, gauche, droite, diagonale haut_gauche, diagonale haut_droit, diagonale bas_gauche, diagonale bas_droit.

On pourrait parcourir toutes ces direction en cherchant une combinaison gagnante, cependant, on sait que le sens de la combinaison n’est pas pris en compte pour gagner. Ainsi, on peut se contenter de parcourir les 4 axes dans une unique direction. On obtient donc 4 sens de parcours, au lieu de 8.

Sachant que nous partons du coin supérieur gauche de la grille, on aura : droite, bas, diagonale bas_droit, diagonale bas_gauche.





Ainsi, pour chaque case on effectue un parcours non redondant en ayant vérifié toutes les possibles combinaisons gagnantes lorsque l'on a fini de parcourir la grille.

On a donc une double boucle qui déplace ce que l'on peut appeler le curseur de vérification sur la grille.

Pour chaque position on vérifie les 4 axes en comptant les répétitions continues du caractère 'X' ou 'O' selon la case où le curseur se trouve, à l'aide d'un compteur.

Ces parcours se manifestent dans le code par : une boucle par axe, l'incrémentation des coordonnées dans les boucles, et une réinitialisation après chaque vérification d'axe à la position du curseur afin de vérifier l'axe suivant en partant de la case du curseur.

On arrête de vérifier un axe lorsque les coordonnées sorte de la mémoire allouée.

On arrête la fonction en retournant 1 lorsque le compteur atteint le nombre d'alignements choisi lors du lancement du programme.

Le main

Condition d'exécution.
D'après le cahier des charges : 3 ≤ dimension ≤ 33 et 1 < alignement ≤ 5.

On met donc ces conditions en tant que condition d'exécution du programme.

Pour ce faire, on transforme les chaînes de caractères du tableau argv en utilisant la fonction atoi (ascii to integer), afin d'effectuer les tests d'acceptation d'exécution du programme.

Si les conditions du cahier des charges ne sont pas respectées, un message d'erreur s'affiche, afin d'informer l'utilisateur des valeurs à insérer et du format attendu.

```
188     if (argc > 3 || argc <= 2 || atoi(argv[1]) < 3 || atoi(argv[1]) > 33
189                   || atoi(argv[2]) <= 1 || atoi(argv[2]) > 5
190                           || atoi(argv[1]) < atoi(argv[2]))
191     {
192         printf("\nMerci de saisir deux arguments correctes [[3 > taille > 33] [1 <
    alignement < 5]] : alignement <= taille\n\n");
193         return (1);
194     }
```

Récupération de l'alignement et de la taille et allocation mémoire.

Maintenant que le test de cohérence des données est effectué, on récupère les valeurs de la taille et de l'alignement via la fonction atoi.

```
195     //recuperation de la dimension
196     size = atoi(argv[1]);
197     //recuperation de l'alignement gagnant
198     align = atoi(argv[2]);
199     //allocation memoire de taille size*size
200     if (!(grille = (char **)malloc(size*sizeof(char *))))
201     {
202         printf("\nErreur allocation memoire.\n");
203         return (1);
204     }
205     while (++i < size)
206         if (!(grille[i] = (char *)malloc(size*sizeof(char))))
207         {
208             printf("\nErreur allocation memoire.\n");
209             return (1);
210         }
211     //initiation de la grille et affichage
212     ft_init(grille, size);
213     ft_print_grid(grille, size);
```

Ensuite, on alloue dynamiquement la mémoire en utilisant l'appel système malloc, sur chaque dimension pour une allocation mémoire de size*size case.

On effectue le malloc dans une condition de structure de contrôle afin d'arrêter le programme en cas d'échec de l'allocation mémoire via l'affichage d'un message d'erreur, et le retour de la valeur 1 au système.

Nous pouvons ensuite faire appel à la fonction d'initialisation de la grille, et à la fonction d'affichage de la grille.

Le tour à tour et la vérification de grille pleine

```
214     //tour a tour
215     while (!ft_check_win(grille, 'X', size, align)
216             && !ft_check_win(grille, 'O', size, align)
217             && !ft_check_full(grille, size))
218     {
219         cpt++;
220         if (cpt % 2 != 0)
221             player = 'X';
222         else
223             player = 'O';
224         ft_insert_choice(player, grille, size);
225         ft_print_grid(grille, size);
226     }
227     if (ft_check_full(grille, size))
228         printf("\nGrille pleine.\n");
```

Le jeu peu commencer  !

Via un boucle, on effectue le tour à tour, en contrôlant à chaque tour l'existence d'un alignement gagnant, ou d'une grille pleine.

A chaque tour, la variable « player » prend la valeur 'O' ou 'X', selon le résultat du modulo 2 d'un compteur incrémenté à chaque tour. Si le modulo 2 est égale à 0, alors le nombre est pair. D'un tour à l'autre le résultat passe naturellement de paire à impair, et la valeur « player » suit ce mouvement.

Ensuite, on gère le cas d'arrêt du tour à tour en cas de grille pleine, en affichant un message informatif.

Enfin, le programme s'arrête en retournant la valeur 0 au système, preuve d'une exécution réussie.

Difficultés rencontrées

Nous avons été confronté à deux principales difficultés : le comportement de la fonction « scanf » lors de la saisie d'une chaîne de caractères, et l'implémentation de la fonction de repérage de combinaison gagnante.
Ces deux difficultés ont été résolues par décomposition du problème principal.

Conclusion

Nous avons pu utiliser différentes fonctionnalités du langages C ainsi que certaines méthodes d'allocation mémoire étant plus proche du système, ce, en résolvant un problème complexe par décomposition en sous-problème.

Nous avons pu nous rendre compte de l'importance du travail en amont sur l'approche du problème. L'angle d'attaque ainsi que le plan de résolution sont fondamentaux.

Le code est un outils permettant de résoudre un problème, en revanche c'est au programmeur de trouver la démarche en fonction des moyens techniques dont il dispose.