Programmation Avancée

Corrigé Puissance 4

Grégory Bise

Version 1.0, 2020-03-23

Table of Contents

Introduction

Ce document présente une solution pour le devoir Puissance 4. je rappelle qu'en programmation, il n'y a jamais une unique solution, prenez donc ce document comme base de comparaison avec votre solution et pour vous apporter des points d'éclaircissement si vous avez rencontré des difficultés.

Le code source est associé à ce document et je présenterai les points clés du code.

Compilation

Le projet se compile via l'outil CMake et ces quelques lignes de commande :

```
$ mkdir ./build
$ cd build
$ cmake -G "Unix Makefiles"  -DCMAKE_BUILD_TYPE=Debug ../
$ make
```

Revue de code détaillée

J'ai, bien entendu, commencé par lire le sujet, et écrire la fonction *main* en fonction des consignes données : un menu avec trois choix, vérifier le choix de l'utilisateur, lancer le jeu, exécuter des test, tant que l'utilisateur ne choisit pas *Quitter* le programme continu, etc.

main.c

J'ai fait simple en fonction de ce que donne l'énoncé, ainsi on obtient un code *main* très minimaliste, lisible et compréhensible.

```
1  int main(int argc, char *argv[])
2  {
3      bool quit = false;
4
5      do
6      {
7          switch (menu())
8          {
9          case 1:
10             playP4();
11             break;
12
13         case 2:
14             testP4();
15             break;
16
17         default:
18             quit = true;
19             break;
20         }
21     } while (!quit);
22
23     return 0;
24 }
```

La méthode *menu* affiche un choix retourne un entier, représentant le numéro saisi par l'utilisateur.

```
1 - Jouer
2 - Mode test
3 - Quitter
```

J'ai ensuite écris le moteur du jeu Puissance 4 en parallèle des tests unitaires. De cette façon j'ai pu tester progressivement mon développement.

p4.c

Je reviendrai sur les tests dans la section suivante.

J'ai utilisé un préfix pour les variables et méthodes de façon à identifier leur utilité. Par exemple, les variables statiques du jeu sont préfixées par p4_, les méthodes pour l'affichage par dsp_, et les méthodes propres au jeu par game_.

Comme pour la méthode *main* j'ai commencé par écrire la méthode *playP4* puisque c'est le point d'entrée du jeu. Dans cette méthode j'ai transcris en code la logique du jeu énoncé dans le sujet.

```
1  void playP4(void)
2  {
3      int colonne = 0;
4      int ligne = 0;
5      int indice_pion = 0;
6
7      printf("Puissance 4 - v1.0\n");
8
9      game_init();
10
11     while(!game_isFinished())
12     {
13         dsp_board();
14         colonne = dsp_position();
15         /* la fonction retourne un choix entre 1 et x
16          * mais les indices de tableaux sont entre 0 et x */
17         colonne -= 1;
18         ligne = game_push(colonne, p4_pions[indice_pion]);
19         if (ligne >= 0)
20         {
21             if (game_isVictory(ligne, colonne, p4_pions[indice_pion]))
22             {
23                 dsp_board();
24                 printf("Le joueur au pion '%c' gagne :)\n", p4_pions[indice_pion]);
25                 return;
26             }
27
28             if (++indice_pion > 1)
29             {
30                 indice_pion = 0;
31             }
32         }
33     }
34
35     printf("Match nul :/\n");
36 }
```

On commence par initialiser le jeu via la méthode *game_init*. Ce qui a pour effet de :

- remplir le plateau de jeu (tableau à deux dimensions de 6 lignes et 7 colonnes) du caractère '.'.
- initialiser le nombre maximum de poins jouables (*p4_max_pion*). Cette variable sera utilisée pour déterminer s'il n'y a plus de coups possibles, et donc match nul.

Ensuite le jeu va boucler, en alternant les joueurs, jusqu'à ce qu'il y ait une victoire ou match nul. Ce dernier cas est testé par la méthode *game_isFinished* où l'on compare la valeur de *p4_max_pion* avec 0.

```
1 static bool game_isFinished(void)
2 {
3     return (0 == p4_max_pion);
4 }
```

La victoire de l'un des joueurs est déterminée via la méthode *game_isVictory*, c'est sans doute la méthode la plus complexe du jeu où l'on peut rencontrer des difficultés. Il faut se mettre à la place du joueur, à chaque fois que l'on va ajouter un pion dans une colonne, on va immédiatement vérifier s'il y a un alignement. C'est ce que va faire la méthode *game_isVictory*, en lui donnant la position du dernier pion ajouté (ligne, colonne), la méthode va cherche dans différentes directions si d'autres pions du même symboles sont adjacents.

```
1  static bool game_isVictory(int ligne, int colonne, const char pion)
2  {
3      /* diagonale / */
4      if (4 <= (1 + game_countPawns(ligne - 1, colonne + 1, -1, 1, pion) +
   game_countPawns(ligne + 1, colonne - 1, 1, -1, pion)))
5      {
6          return true;
7      }
8      /* horizontale - */
9      if (4 <= (1 + game_countPawns(ligne, colonne -1, 0, -1, pion) +
   game_countPawns(ligne, colonne + 1, 0, 1, pion)))
10     {
11         return true;
12     }
13     /* verticale   | */
14     if (4 <= (1 + game_countPawns(ligne - 1, colonne, -1, 0, pion) +
   game_countPawns(ligne + 1, colonne, 1, 0, pion)))
15     {
16         return true;
17     }
18     /* diagonale   \ */
19     if (4 <= (1 + game_countPawns(ligne - 1, colonne - 1, -1, -1, pion) +
   game_countPawns(ligne + 1, colonne + 1, 1, 1, pion)))
20     {
21         return true;
22     }
23
24     return false;
25 }
```

Ainsi, dans la méthode *game_isVictory*, on va appeler différentes méthodes pour compter le nombre de pions adjacents de même symbol, à partir de la dernière position jouée. En prenant pour exemple la grille ci-après, la dernière position jouée est ligne 3, colonne 4, pion 'O'.

```
  +---+---+---+---+---+---+---+
1 | . | . | . | . | . | . | . |
  +---+---+---+---+---+---+---+
2 | . | . | . | . | . | . | . |
  +---+---+---+---+---+---+---+
3 | . | . | . | O | . | . | . |
  +---+---+---+---+---+---+---+
4 | . | X | O | X | . | . | . |
  +---+---+---+---+---+---+---+
5 | . | O | X | O | . | . | . |
  +---+---+---+---+---+---+---+
6 | O | X | O | X | . | . | . |
  +---+---+---+---+---+---+---+
    1   2   3   4   5   6   7
```

La méthode va rechercher dans les quatres directions possibles depuis cette position. La recherche utilise une méthode parcourant la grille dans une direction donné via les variables *pasLigne* et *pasColonne*.

```
1 static int game_countPawns(int ligne, int colonne, int pasLigne, int pasColonne,
  char pion)
2 {
3     if (ligne < 0 || ligne >= P4_MAX_LIGNE)
4     {
5         return 0;
6     }
7
8     if (colonne < 0 || colonne >= P4_MAX_COLS)
9     {
10        return 0;
11    }
12
13    if (pion != p4_board[ligne][colonne])
14    {
15        return 0;
16    }
17
18    return 1 + game_countPawns(ligne + pasLigne, colonne + pasColonne, pasLigne,
  pasColonne, pion);
19 }
```

La méthode *game_countPawns* utilise la récursivité.

Tests

Comme annoncé au début du corrigé, j'ai écrit les tests en même tant que j'ai développé le jeu. Ceci m'a permis de tester progressivement ma solution. Quand mes tests sont passés, j'ai pris un peu de recul sur la solution et j'ai amélioré certains traitements, et le style du code. Après chaque changement, je lançais les tests à nouveau pour m'assurer que le fonctionnel n'était pas impacté. J'ai, grâce à cette méthode, pu rapidement corriger des régressions ou non-conformités dues à des erreurs.

J'ai opté pour une méthode simple afin d'écrire mes tests, en utilisant une macro pour répéter le code redondant.

```
#define TEST(func, text, counter) { \
    printf("[TEST] %s\t", text); \
    if (!func()) \
    { \
        counter++; \
        printf("FAIL\n"); \
    } \
    else \
    { \
        printf("PASS\n"); \
    } \
}
```

L'avantage est que c'est portable est que ça ne rend pas incompréhensible le code qui l'utilise.

```
int fails = 0;

TEST(test_game_init, "Initialisation du jeu...", fails);
TEST(test_game_max_pions, "Initialisation du nombre de pions...", fails);
TEST(test_game_push, "En dehors des limites...", fails);
...
printf("%d test(s) en échec\n", fails);
```

Les tests sont dans le même fichier que les méthodes du jeu. Les méthodes étant statiques, donc limitées au fichier où elles sont déclarées, elles ne peuvent être appelées en dehors du fichier pour les tests. C'est un inconvénient des méthodes statiques.

Si on souhaite réellement sortir les tests dans un autre fichier, on peut le faire de différentes manières, plus ou moins élégantes.

Inclure le fichier source dans le fichier source de test

De cette manière le source complet est inclu dans le fichier source où l'on va écrire nos tests, ainsi, les méthodes statiques seront également dans le fichier test. Je déconseille cette méthode pouvant générer des problèmes de définitions multiples.

Utiliser une variable de pré-compilation

On pourrait définir une macro pour remplacer le mot clé *static*, puis utiliser cette macro pour déclarer les méthodes. Via une variable de pré-compilation on définit la macro de deux manières, pour que l'on puisse générer un exécutable de test et l'exécutable standard.

```
#if UNIT_TESTS
    #define my_static
#else
    #define my_static static
#endif
...
my_static bool game_isVictory(int ligne, int colonne, const char pion);
...
```

En compilant avec la définition *UNIT_TESTS*, les méthodes seront publiques, sinon elles seront statiques. C'est la manière la plus élégante que je connais, mais on génèrera deux exécutables (ou deux librairies).

Coverage

J'ai également vérifié, grâce au code coverage, la couverture de passage dans le code via mes tests. J'ai pu, en voyant les lignes non couvertes, améliorer mes tests ou en ajouter.

Pour générer la couverture de code, j'ai exécuter les commandes suivante :

```
1 $ gcov build/CMakeFiles/Puissance4.dir/src/*
2 $ lcov -c -o coverage.info -d .
3 $ genhtml -o report coverage.info
```

Le résultat et disponible dans le répertoire *report*, fichier *index.html*.

Toutes les lignes ne sont pas couvertes en lançant les tests, à savoir, les méthodes d'affichage, la méthode de lancement du jeu.