INFO7 - SAÉ L1 info - 2022-2023

Version 2 du 2 mai 2023

1 Organisation du projet

1.1 Calendrier

Le projet va se dérouler dans le cadre d'une Situation d'Apprentissage et d'Évaluation (SAÉ). Pour cela vous aurez des travaux à rendre et des évaluations tout au long du projet.

Durant le projet nous allons vous évaluer, donc vous devez être présent(e) à toutes les séances.
Si vous êtes absent à une séance vous aurez alors -1 à la note finale de contrôle continu, et si lors de cette séance une évaluation était prévue vous aurez en plus 0 à cette évaluation.

Le projet va se diviser en 9 séances de 3h en salle machine de la façon suivante:

- **séance 1 - semaine 13 (27 au 31 mars):** présentation du projet; identification des éléments à représenter et le choix de leur représentation en listant les avantages et inconvénients de chacune. Etude à rendre.
- **séance 2 - semaine 15 (10 au 14 avril):** affichage d'un échiquier et des pièces ; implémentation des fonctions de base ; lecture/écriture de fichiers contenant la représentation d'un état de jeu et affichage. Evaluation du rendu de la séance 1.
- **séance 3 - semaine 15 (10 au 14 avril):** affichage plus évolué (couleurs, pièces, ..) ; identification des masques.
- **séance 4 - semaine 18 (2 au 6 mai):** mouvement d'une pièce : déplacements possibles, prises possibles, affichage des possibilités (masques). Rendu du code permettant l'affichage d'un état de jeu, l'affichage du masque montrant les mouvements possibles du Roi et de la Tour à partir d'états donnés sous représentation FEN dans des fichiers.
- **séance 5 - semaine 18 (2 au 6 mai):** mouvement de toutes les pièces. Évaluation des séances 2 à 4.

- **séance 6 - semaine 19 (9 au 13 mai):** boucle de jeu avec listes des pièces mangées.
- **séance 7 - semaine 19 (9 au 13 mai):** rajout de l'historique, rajout des règles de base. Rendu du code permettant un jeu avec des règles simples.
- **séances 8 et 9 - semaine 19 (9 au 13 mai):** rajout de règles, de stratégies. Évaluation de la boucle de jeu. Rendu final.

1.2 Travail préalable

Avant la première séance (29 mars groupe A et 31 mars groupes B et C), vous devez:

1. comprendre le jeu d'Échecs (voir la présentation dans la partie 2);
2. trouver un autre étudiant de votre groupe pour constituer un binôme. Lors de la séance 1, la liste des binômes sera déterminée et fixe pour tout le projet. Attention nous nous réservons le droit de vous imposer un binôme dans le cas où vous êtes seul(e). Durant le projet nous allons vous évaluer, donc vous devez être présent(e) à toutes les séances.

1.3 Consignes

- Un espace Moodle "info7" a été créé, vous y déposerez les fichiers demandés au fur et à mesure des séances.
- Les rendus se feront par binôme avec le nom que nous vous aurons indiqué lors de la séance 1.
- Vous devrez travailler en binôme (sauf condition exceptionnelle validée par l'enseignant), lors des évaluations orales l'enseignant choisira qui du binôme répond aux questions. Un seul code sera fourni et les 2 membres devront pouvoir expliquer la démarche : si la solution de l'un s'impose alors l'autre doit pouvoir répondre, ce qui signifie que vous devez avoir communiqué.
- Le calendrier de rendu des différentes étapes sera le suivant:
  1. Avant la séance 2 : déposer dans "Evaluation1_Séance 1" le tableau d'analyse + code des 4 fonctions de base (avec 2 représentations);
  2. Avant la séance 5 (2 mai) : déposer dans "Evaluation2_Séances 2-4" le code des 4 fonctions de base + affichages + mouvements possibles du Roi et de la Tour;
  3. Avant la séance 8 : déposer dans "Etape3_groupeXX" les fichiers demandés pour la reproduction des individus;
  4. A la fin de la dernière séance tous les fichiers seront à rendre avec a minima les fonctions demandées.

Remarques de politesse :

- tout code rendu doit se compiler sans faute, une faute de compilation divisera la note par 2 ;
- le code doit être documenté : avant chaque fonction 1 ou 2 lignes doivent décrire la fonction et présenter les noms des variables utilisées notamment dans les paramètres ;
- les commentaires dans les fonctions seront mis avec parcimonie, uniquement lors de code un peu plus compliqué nécessitant une explication.

Critères d'évaluation :

- La qualité du code : correction, concision, précision, duplication du code, etc.
- Qualité d'écriture du code : espacement, choix des noms de variables, pertinence des commentaires, etc.
- Qualité des algorithmes : place mémoire utilisée, complexité (nombre de cases testées, nombre de comparaison, etc.)
- Capacité d'explication du code et écriture de tests incrémentaux

Évaluation :

- note finale :
  - Session1 = 1/2 examen + 1/2 C.C.
  - Session2 = Sup(examen2, (1/2 examen2 + 1/2 C.C.), Session1)

La section 2 présente le sujet du projet et notamment ce que vous devez faire avant la première séance, la section 4 détaille l'architecture du code et les fonctions nécessaires et les sections suivantes présentent le travail à faire et à rendre pour chaque séance.

2 Présentation du sujet : implémentation d'un jeu d'Échecs

Le jeu d'Échecs est un jeu qui se joue à 2 avec un plateau composé d'une grille sur laquelle chaque joueur place et déplace des pièces dans un même but: mettre en **Échec** le Roi (une des pièces) de l'adversaire. Chaque joueur dispose des mêmes pièces au départ et par le jeu des déplacements de ses pièces, il va pouvoir prendre les pièces de l'adversaire afin de l'affaiblir et de gagner. Le joueur qui gagne est celui qui a réussi à mettre le Roi en échec de telle façon que le Roi le reste quel que soit son prochain déplacement, on dira qu'il est **Mat**.

Les pièces utilisées par les joueurs sont au nombre de 16, avec 6 types différents qui ont chacune une manière différente de se déplacer et de prendre. Les pièces sont:

- Roi (anglais : King) ⇒ 1 pièce
- Reine (anglais : Queen) ⇒ 1 pièce
- Fou (anglais : Bishop) ⇒ 2 pièces
- Tour (anglais : Rook) ⇒ 2 pièces
- Cavalier (anglais : Knight) ⇒ 2 pièces
- Pion (anglais : Pawn) ⇒ 8 pièces

Pour vous familiariser a minima avec **le jeu d'Échecs**, c'est-à-dire comprendre les différents déplacements des pièces, les règles de déplacements et de prises, nous vous demandons de:

1. lire la page Wikipedia consacrée à ce jeu [1], et plus particulièrement la partie "Règles du jeu".
2. vous rendre ensuite sur le site gratuit lichess.org [2] où dans l'ordre vous trouverez:
   - dans l'onglet "Apprendre":
     - les "Bases des échecs" vous apprend à déplacer vos pièces et les fondamentaux. Cette partie est très utile pour la compréhension du jeu;
     - les autres parties sont plus de l'entraînement, si vous souhaitez aller un peu plus loin.
   - dans l'onglet "Outils" la manière d'analyser un échiquier avec la notation FEN que nous utiliserons qui permet de décrire l'échiquier, vous remarquerez que chaque fois que vous déplacez une pièce la représentation de l'échiquier est modifiée. Vous trouverez également les ouvertures classiques.
   - dans l'onglet "Jouer" la possibilité de joueur une partie contre l'ordinateur ou contre un.e ami.e.
   - dans l'onglet "Problèmes" des problèmes classiques (pour les joueurs plus aguerris)

Il est bien entendu que si vous êtes un joueur d'échecs ce travail préalable est inutile (sauf à connaître la notation FEN).

[1] https://fr.wikipedia.org/wiki/Échecs
[2] https://lichess.org/fr

3 Séance 1 : analyse du jeu

3.1 Analyse et discussion

Dans un premier temps vous allez faire une analyse du jeu avec comme objectifs:

- identifier tous les éléments à représenter : pièces, grille, historique, ...;
- identifier les actions, fonctions qu'il faudra mettre en oeuvre : poser une pièce, récupérer une pièce, ...;
- discuter des avantages et inconvénients de chaque représentation possible pour chaque élément en fonction des actions à mettre en oeuvre ensuite.

Le travail à rendre sera sous forme d'un tableau à 2 dimensions où vous mettrez en ligne les éléments à représenter, en colonne la représentation proposée, et dans chaque case les avantages et inconvénients que vous voyez pour cette représenation en fonction des différentes actions à mener ensuite. Le fichier `Analyse_Seance1.*` vous propose un exemple de tableau à rendre auquel vous pourrez ajouter des lignes et des colonnes. Le tableau ainsi rempli sera à rendre dans un fichier au format pdf.

3.2 Fonctions de base

Dans un deuxième temps, pour étayer votre discussion vous choisirez 2 représentations pour le plateau et vous coderez les fonctions basiques qui permettent de:

- créer un plateau vide **empty**,
- récupérer le contenu d'une case **get_square**,
- modifier le contenu d'une case **set_square**,
- initialiser le plateau avec ses pièces **start**,
- déplacer une pièce d'une case de départ dans une case d'arrivée **move_piece**.

Vous écrirez une structure `game` qui contiendra les objets permettant de décrire le statut du jeu à chaque instant, structure qui pourra être enrichie tout au long du projet.

3.3 Critères d'évaluation de cette séance

L'évaluation de cette partie sera faite d'une part sur la partie écrite à rendre:

- la liste des propositions des représentations des pièces, du plateau de jeu, de l'historique;
- l'analyse (avantages/inconvénients) de ces propositions par rapport au coût mémoire, à la complexité des fonctions de base écrites avec les différentes représentations;

- l'explication argumentée du choix de la représentation des pièces, du plateau de jeu et de l'historique.

Ensuite lors de la séance 2, l'enseignant évaluera votre capacité à argumenter vos choix vis à vis des exigences du client (voir ci-après), notamment à travers le code que vous aurez fourni. Dans cette évaluation, l'écriture du code ne sera pas évaluée.

4 Environnement du projet

Vous allez écrire en C++ un programme qui permet de réaliser une partie d'Échecs entre 2 joueurs humains, puis entre un joueur humain et un ordinateur. Les interactions entre l'utilisateur et la machine seront faites à travers une fenêtre (appelée "Terminal") dans laquelle on peut envoyer des commandes à la machine (compilation, exécution, ...) et récupérer des données (affichage dans la fenêtre, sauvegarde dans des fichiers, ..).

Le client exige d'avoir un programme qu'il puisse exécuter dans un Terminal afin de pouvoir passer le nom des fichiers d'E/S directement en ligne.

Le travail va se découper en plusieurs phases et le code va se découper en plusieurs grandes parties avec une partie par fichier, et les fonctions se rapportant à une même partie seront regroupées dans un même fichier comme suit :

- **types.hpp** : les types utilisés pour la représentation des pièces, du plateau de jeu, de l'état du jeu, et de tous les types que vous aurez à rajouter tout au long du projet;
- **board.*** : manipulation des pièces et mise à jour du plateau de jeu: récupérer/modifier le contenu d'une case, déplacements/prises possibles des pièces;
- **view.*** : affichage de l'état du jeu avec des options telles que l'affichage des possibilités de déplacement/prise d'une pièce à l'aide de masques;
- **mask.*** : création de masques de mise en évidence de certaines cases en fonction de la demande comme mettre en évidence l'ensemble des pièces adverses pouvant être prises par une pièce déterminée, ou l'ensemble des déplacements possibles d'une pièce, ...

4.1 Modularité du programme

Dans ce projet vous allez utiliser la modularité des programmes, soit la possibilité de découper votre programme en plusieurs modules (c'est-à-dire plusieurs fichiers). Pour cela, vous allez déclarer les types de vos variables et de vos fonctions dans des fichiers d'*en-tête* qui auront comme extension .hpp. Ceux-ci

seront ensuite indiqués par une instruction `include` dans les fichiers .cpp où les types et les fonctions seront utilisés.

Dans ce projet, tous les types des données seront définis dans un fichier **types.hpp** et toutes les fonctions seront d'abord déclarées dans des fichiers dont l'extension sera `.hpp` et ensuite définies dans des fichiers de même nom où seule l'extension sera modifiée en `.cpp`. Cette décomposition va permettre de répartir les fonctions dans différents fichiers en fonction de leur thématique (par exemple les fonctions relatives à l'affichage seront dans un fichier **view.*** alors que celles relatives aux déplacements des pièces seront dans un fichier **move.***) et de ne recompiler que le(s) fichier(s) modifié(s) depuis la dernière compilation, c'est ce qu'on appelle la compilation séparée.

4.1.1 Les fichiers d'en-tête

Les fichiers d'*en-tête* sont des fichiers dont l'extension sera `.hpp` et qui contiennent les en-têtes des fonctions qui seront utilisées ultérieurement dans des fichiers de code `.cpp`. Pour appeler les fonctions définies dans un fichier `.hpp` il suffira d'inclure ce fichier en en-tête du fichier code dans lequel les fonctions seront utilisées. C'est exactement ce que vous faites quand vous incluez une bibliothèque `#include <cmath>` et que vous utilisez la fonction `sqrt`.

En général on définit les entêtes des fonctions dans un fichier d'extension `.hpp`et les fonctions dans un fichier de même nom avec l'extension `.cpp`.

Exemple

Code du fichier calcul.hpp

```
// calcul.hpp : déclaration des fonctions
// Carre d'un nombre
float carre(float);
// x puissance y
double puissance(int, int);
```

Code du fichier calcul.cpp

```
// calcul.cpp : définition des fonctions
// Carre d'un nombre
float carre(float x) {return x*x;}
// x puissance y
double puissance(int x, int y){
    int res = 1;
    for (int i=1; i<=y ; i++)
        res = res*x;
    return res;
}
```

Dans le code du fichier main.cpp

```
// main.cpp : programme principal
#include "calcul.hpp" //inclusion du fichier contenant les entêtes des fonctions,
                      //remarquez les guillemets "" à la place des <>
```

Parfois il arrive que les définitions se fassent directement dans le fichier `.hpp` mais le code risque d'être moins lisible, et surtout oblige à recompiler tout le code à chaque modification.

4.1.2 Compilation séparée : protection d'inclusion multiples d'un même fichier

La programmation modulaire implique automatiquement une compilation séparée. Lorsque des fonctions sont déclarées (ou définies) dans différents fichiers d'en-tête, il arrive souvent que l'inclusion d'un même fichier d'en-tête soit faite plusieurs fois car utilisée dans plusieurs fichiers. Pour éviter cette redéfinition des fonctions, des directives permettent de faire de la compilation conditionnelle.

Il existe plusieurs directives permettant la compilation conditionnelle (`#ifdef`, `#ifndef`,`#if`) afin de compiler les lignes comprises entre une de ces directives et la directive de fin du bloc `#endif` uniquement si la condition est remplie. Pour éviter l'inclusion multiple d'un même fichier d'en-tête, nous allons définir au début de ces fichiers une variable dite **variable de compilation** à l'aide de la directive `#define` et c'est seulement si cette variable `nom_variable` n'est pas encore connue lors la compilation (`#ifndef nom_variable`) que les fonctions seront compilées. Par convention la variable de compilation porte le nom du fichier d'en-tête et est écrite en majuscule (exemple : la variable de compilation du fichier "toto.hpp" sera `TOTO_HPP_`)
Exemple avec le fichier précédent :

```
//calcul.hpp : déclaration des fonctions
#ifndef CALCUL_HPP_  //si la variable de compilation n'est pas
                     //connue (définie)
#define CALCUL_HPP_  //on la définit

float carre(float x);
double puissance(int x, int y);

#endif  //fin des fonctions à compiler si la variable
        //de compilation n'est pas définie
```

Une directive alternative est `#pragma once` qui est une directive préprocesseur implémentée dans la plupart des processeurs. Elle est dépendante des processeurs contrairement aux précédentes. L'utilisation de `#pragma once` est moins sujette à des erreurs de la part du développeur que l'utilisation des directives précédentes. En revanche la duplication d'un fichier sous des noms différents dans des répertoires différents (dans le cas de liens physiques ou symboliques où les données sont au même endroit en mémoire) n'est pas trivial à identifier et `#pragma once` ne joue pas forcément son rôle de détecter les inclusions multiples lorsqu'il existe ce type de fichiers.
Exemple avec le fichier précédent :

```
//calcul.hpp : définition des fonctions
```

```
#pragma once

float carre(float x);
double puissance(int x, int y);
```

4.1.3 Compilation séparée : ligne de commande

Pour compiler un programme simple (un seul fichier source), on utilise la commande suivante :

```
$ g++ nom_prog.cpp -o nom_prog
```

On demande au compilateur g++ de compiler le fichier source `nom_prog.cpp` et de créer le fichier exécutable `nom_prog`.
Rappel : si on ne précise pas -o `nom_prog`, le compilateur créera un exécutable nommé `a.out`.

Pour compiler un programme composé de 2 fichiers sources (`fonc.cpp` et `test_fonc.cpp`), il va falloir d'abord compiler chacun des fichiers séparément, compilation qui va renvoyer un fichier objet pour chaque fichier compilé. Ces fichiers objets vont être ensuite liés avec les fichiers précompilés d'une ou plusieurs bibliothèques grâce à l'éditeur de lien, afin de créer le fichier exécutable. La compilation va donc suivre les étapes suivantes :

```
$ g++ -c fonc.cpp #crée le fichier fonc.o
$ g++ -c test_fonc.cpp #crée le fichier test_fonc.o
$ g++ -o test_fonc fonc.o test_fonc.o #crée le fichier exécutable test_fonc
                                      #à partir des fichiers objets précédents
```

4.1.4 Compilation séparée : Makefile

Pour éviter de compiler chaque fois les différents fichiers modifiés et ensuite de créer l'exécutable, et surtout pour éviter d'oublier de compiler un fichier anciennement modifié, il est d'usage de créer un fichier nommé **Makefile** qui va permettre de rassembler toutes les lignes de commande. Une fois ce fichier créé, la seule commande à exécuter sera alors `make` qui permettra alors de créer l'exécutable dont le nom aura été donné dans le fichier Makefile. Il est possible de spécifier quelle règle du Makefile executer (par example, `maketest_fonc`) , sinon quoi c'est la première règle du fichier qui est choisie.

Chaque règle Makefile se compose d'un nom, d'une liste optionnelle d'ingrédients, et d'une liste de commandes à effectuer. Le nom de la règle est généralement le nom du fichier généré par la règle, et les ingrédients correspondent soient à des noms de fichiers, soit à d'autres règles. Les commandes d'une règle ne sont exécutées que si et seulement si le fichier à générer n'existe pas ou qu'un des ingrédient est plus récent que celui-ci. Ainsi, seul le code des fichiers modifiés depuis la dernière compilation sont recompilés.

La création du fichier Makefile demande de bien comprendre la compilation. Aussi des outils ont été proposés pour créer ce fichier sans connaissances particulières uniquement savoir quels fichiers sont utilisés dans le projet.

Exemple de Makefile pour l'exemple précédent:

```
default: test_fonc # regle par defaut
fonc.o: fonc.cpp # regle de creation du fichier fonc.o
    g++ -c fonc.cpp
test_fonc.o: test_fonc.cpp
    g++ -c test_fonc.cpp
test_fonc: fonc.o test_fonc.o  # regle de l'executable en fonction des .o
    g++ -o test_fonc fonc.o test_fonc.o
```

On pourra utiliser la commande `make test_fonc` qui mettra en oeuvre les différentes règles suivant les fichiers modifiés depuis la dernière compilation.

Dans la suite, les fichiers Makefile seront automatiquement générés par l'outil CMake.

4.1.5 Compilation séparée : utilisation de CMake pour créer Makefile

Pour compiler un projet et créer son exécutable à partir de plusieurs fichiers, nous pouvons utiliser `CMake`qui un outil open source permettant de créer le script de compilation à l'aide d'un fichier de configuration générique : appelé **CMakeLists.txt**. Ce fichier est indépendant du système (linux, windows, etc.).

Le fichier CMakeLists.txt doit décrire votre projet. Il est composé d'appels à des fonctions permettant de lire les noms des fichiers, librairies, des variables nécessaires à l'écriture du Makefile. Les fonctions que vous pourrez utiliser sont :

- `cmake_minimum_required()` : permet de préciser une version de cmake à utiliser (la plus récente actuellement est la 3.26, mais faites attention vous n'avez peut-être pas installé la dernière version de cmake),
- `project()` : permet de donner le nom du projet,
- `add_library()` : permet de générer une librairie à partir d'une liste de fichiers,
- `target_include_directories()` et `target_link_libraries()` : permettent de donner les dépendances (répertoires contenant du code, librairies spécifiques) à ajouter aux fichiers,
- `add_executable()` : permet de générer le nom de l'exécutable à partir de la liste des fichiers sources,
- `set()` : permet de définir des variables.

En reprenant l'exemple précédant du Makefile, le CMakeLists.txt suivant permet de :

- d'enregistrer dans une variable **SRC** la liste de tous les fichiers sources (*i.e.* ayant comme extension .cpp),
- d'enregistrer dans une variable **HEADER** la liste de tous les fichiers d'entête (*i.e.* ayant comme extension .hpp),
- de définir notre projet **projet**,
- de générer l'exécutable dont le nom est `test_fonc`.

Le fichier résultat sera le suivant:

```
# CMakeLists.txt
# le dièse commence une ligne de commentaire
project(projet) # nom du projet

set(SRC fonc.cpp test_fonc.cpp)
set(HEADER test_fonc.hpp)

add_executable(test_fonc ${SRC} ${HEADER})
```

Une fois que le fichier CMakeLists.txt est défini, il suffit d'exécuter la commande `cmake nom_repertoire_source`. `nom_repertoire_source` contient aussi le fichier CMakeLists.txt. La commande produit alors le fichier de compilation `Makefile` qui permettra ainsi de générer l'exécutable. Lorsqu'un fichier sera modifié, il suffira de relancer la commande `make` et l'exécutable. La commande `cmake` devra être relancée uniquement lors de la modification de CMakeLists.txt (lors de l'ajout ou la suppression de fichiers sources).

La commande `cmake` va créer plusieurs fichiers qu'il est souvent préférable de mettre dans un répertoire, ce qui permet de détruire d'un coup tous les fichiers si nécessaire (notamment lors de modifications de CMakeLists.txt).

Première étape : créer le Makefile:

1. créer un répertoire nommé `build` par convention
   `mkdir build`
2. déplacer dans le répertoire
   `cd build`
3. exécuter à partir de ce répertoire la commande `cmake` sur le répertoire source
   `cmake ..`

Deuxième étape : compiler et exécuter:

1. sous le répertoire `build` compiler les fichiers sources
   `make`
2. sous le même répertoire, lancer l'exécutable
   `./nom_executable paramètres`

Pour utiliser CMake vous devez bien entendu avoir téléchargé l'application. Vous pouvez le faire en suivant le lien : `https://cmake.org/download/`

4.2 Entrées-Sorties

Vous allez écrire la fonction **main** de manière à pouvoir réaliser les entrées-sorties à l'aide de fichiers ou de valeurs.

La compilation de votre fichier se fera dans un premier temps avec la ligne suivante (ou en écrivant le fichier CMakeList.txt équivalent) :

```
g++ -W -Wall main.cpp -o main
```

Si vous souhaitez exécuter votre programme en passant plusieurs paramètres en entrée/sortie vous appellerez la ligne de commande suivante :

```
./main argument1 argument2 ... argumentn
```

Les paramètres de cette dernière ligne de commande dans le programme seront pris en compte dans le programme via les arguments possibles de la fonction main.

4.2.1 Arguments de la fonction `main`

La fonction main accepte 2 arguments qui sont :

- **argc** : de type entier (`int`), qui renvoie le nombre de paramètres effectivement passés au programme. Ce nombre est toujours ≥ 1, puisque le nom de l'exécutable est lui-même un paramètre effectif.
- **argv** : de type tableau de chaînes de caractères (`char* argv[]`), où `argv[0]` contient donc le nom de l'exécutable, `argv[1]` contient donc la chaîne de caractères "argument1", et `argv[i]` contient le ième paramètre passé au programme.

Le prototype de la fonction main est donc :

```
int main (int argc, char * argv[]);
```

Dans notre exemple précédent :

```
./main fichier 10
```

`argc` $==$ 3 et `argv[0]` contient "./main", `argv[1]` contient "fichier" et `argv[2]` contient "10" (pour récupérer "10" comme entier il faudra convertir la chaîne de caractères en entier avec la fonction `stoi(argv[2])`.

4.2.2 Lecture d'un fichier en C++

La classe `fstream` permet de lire ou écrire un fichier à partir d'un flux. Plus particulièrement, la classe `ifstream` permet de gérer des opérations d'entrée sur des fichiers :

1. Pour ouvrir le fichier de données dont le nom est contenu dans la variable `nom_fic` :

   ```
   char[256] nom_fic = "toto";
   ifstream fic(nom_fic);
   ```

   Dans ce cas, le fichier **toto** est ouvert par un flux nommé **fic** de la même façon que le flux d'entrée standard "cin" que vous connaissez déjà. La récupération des données contenues dans le fichier est aussi simple que lire les données saisies au clavier.

2. Pour lire les données contenues dans un fichier, en fonction du type de données récupérées le code devra être adapté mais ressemblera aux quelques lignes suivantes lorsque le fichier contient des entiers :

   ```
   if (fic) {
     int tab[100]; // lecture des données de type entier
                   // enregistrées dans un tableau
     int i = 0;
     while (!fic.eof() && i < 100) { // tant que la fin
                           // du fichier n'est pas lue et
                           // que le tableau n'est pas plein
         fic >> tab[i]; // lire chaque donnée comme un entier
                        // à enregistrer dans tab[i]
         i++;
     }
   }
   else {
     cout << "ERREUR : impossible d'ouvrir le fichier" << endl;
   }
   ```

3. Pour fermer le fichier lorsque la lecture a été effectuée :

   ```
   close(fic);
   ```

4.2.3 Écriture dans un fichier en C++

De la même manière que la lecture d'un fichier s'effectue à l'aide d'un flux d'entrée, l'écriture (d'un fichier texte) s'effectue à l'aide d'un flux de sortie en utilisant la classe ofstream qui gère les opérations de sortie sur des fichiers :

1. Pour ouvrir un fichier de données :

```
char[256] nom_fic = "toto";
ofstream fic(nom_fic);
```

   Dans ce cas, le fichier **toto** est ouvert par un flux nommé **fic** de la même façon que le flux de sortie standard "cout". L'écriture des données dans le fichier s'effectue aussi simplement que l'affichage à l'écran.

2. Le code suivant illustre l'écriture de données (ici des entiers) dans un fichier texte :

```
if (fic) {
  int i = 0;
  while (i < 100) { // enregistrement des valeurs d'un tableau tab
                    // de 100 éléments dans le fichier
      fic << tab[i] << " "; // écriture de l'élément i
                            // syntaxe similaire à l'affichage
      i++;
  }
}
else {
  cout << "ERREUR : impossible d'ouvrir le fichier en sortie" << endl;
}
```

3. Pour fermer un fichier lorsque l'écriture est terminée :

```
close(fic);
```

Les exemples précédents décrivent deux opérations de base, mais il existe de nombreuses opérations permettant de lire à partir d'une position, de lire un caractère à la fois, etc. et de même pour les opérations d'écriture.

5 Séance 2 : Structuration du projet et affichage basique

5.1 Structuration du projet

But : Le but de cette section est de structurer correctement votre programme et de mettre en place l'environnement de développement.

Expression du besoin :

Besoin de séparation de code. Un programme correctement structuré consiste à séparer les différentes déclarations et définitions de types et fonctions en plusieurs fichiers selon des thématiques. Une thématique développe une partie des fonctionnalités de l'application. Par exemple, toutes les définitions de fonctions qui concernent le plateau (board) devront être regroupées ensemble dans un même fichier `board.cpp`. Pour des questions de clarté et de lisibilité d'une application, les déclarations de type peuvent être regroupées dans un même fichier au lieu d'être dispersées dans plusieurs fichiers. De plus pour faciliter la lecture des nombreux types, c++ permet de définir des alias de type en utilisant le mot clé `using`. Par exemple :

```
using nom_alias = type_alias;
```

Le nom de l'alias `nom_alias` remplace le type `type_alias`. Dans l'exemple ci-dessous, "MyTab" représente un tableau de 10 entiers :

```
using MyTab = int[10];
MyTab b;  // au lieu de int b[10];
for (int i = 0; i < 10; i++)
  b[i] = 0;
```

Besoin de l'environnement de travail et tests. Nous souhaitons aussi que l'environnement de travail pour compiler et exécuter votre programme utilise les outils cmake/make.

Il est aussi nécessaire dans un bon environnement de développement de tester chacune des fonctions écrites.

Attendus : L'ordre des attendus n'est pas important et vous pouvez faire les attendus dans l'ordre que vous souhaitez.

A2.1.a Définir le fichier *board.cpp* qui regroupe les fonctions précédentes de la section 3.2.

A2.1.b Déclarer les entêtes des fonctions dans le fichier `board.hpp`.

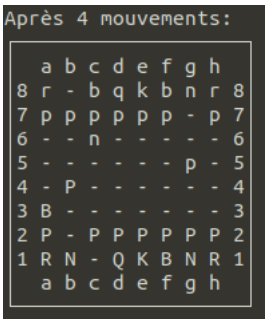
A2.1.c Définir des structures utilisées dans un fichier `types.hpp`.

A2.1.d Écrire un programme principal dans le fichier `main.cpp` pour tester les fonctions écrites.

5.2 Affichage basique

But : Le but de cette section est de réaliser des affichages en mode texte avec une mise en forme simple.

Plateau initial



Après 4 mouvements

Figure 1: Exemples d'affichage basique du plateau initial

Expression du besoin :

Nous souhaitons un affichage dans la console d'un plateau de jeu avec les pièces sous la forme donnée par la figure 1. Il est à noter la numérotation des colonnes avec les lettres a, b, c, ..., h, et la numérotation des lignes de 1 à 8. Les pièces doivent être représentées par les lettres suivantes :

- r ou R : rook (tour);
- n ou N : knight (cavalier);
- b ou B : bishop (fou);
- q ou Q : queen (dame);
- k ou K : king (roi);
- p ou P : pawn (pion).

Les pièces noires sont en minuscules et les pièces blanches en majuscules.

Attendus :

A2.2.a Créer un fichier `view.cpp` qui regroupe les fonctions définies dans cette section qui concerne l'affichage.

A2.2.b Définir la fonction `print_square` qui affiche une case.

A2.2.c Définir la fonction `print_board` qui affiche le contenu d'un plateau (échiquier et pièces) passé en paramètre de la fonction.

A2.2.d À l'aide des fonctions décrites dans la séance 1, définir un test dans un programme principale qui consiste à créer un plateau avec des pièces, puis à faire bouger 2 pièces noires et deux pièces blanches et à afficher les déplacements.

5.3 Notation FEN

But : Le but de cette section est d'écrire et lire dans des fichiers contenant des plateaux en notation FEN.

Expression du besoin :

Nous souhaitons un second affichage selon la notation Forsyth-Edwards[3] (FEN). Elle consiste à représenter les emplacements des pièces sur le plateau sous forme d'une chaîne de caractères où les lettres représentent les pièces et les chiffres les nombres de cases consécutives sans pièce. Chaque ligne du plateau sera séparée par un "/". Sous cette notation, le plateau de jeu initial (plateau de gauche de la figure 1) est donc représenté par la chaîne de caractères :

"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR"

La notation FEN est compacte et nous souhaitons enregistrer et lire l'état d'un plateau en notation FEN vers et depuis un fichier.

Attendus :

A2.3.a Définir une fonction **write_FEN** qui écrit l'état du plateau sous format FEN dans un fichier. Le plateau et le fichier sont des paramètres de la fonction.

A2.3.b Définir une fonction **read_FEN** qui lit l'état du plateau sous format FEN depuis un fichier. Le plateau et le fichier sont des paramètres de la fonction.

A2.3.c Définir un exécutable (code avec programme principale) `test_read_FEN` dont le paramètre est le nom fichier à lire en notation FEN et qui affiche le plateau correspondant.

A2.3.d Définir un exécutable (code avec programme principale) `test_write_FEN` dont le paramètre est le nom fichier à écrire en notation FEN et qui écrit un plateau.

À Noter que 2 fichiers `FEN1.txt` et `FEN2.txt` en notation FEN sont disponibles sur Moodle pour exemple.

6 Séance 3 : affichage avancé

6.1 Affichage en couleur

But : Le but est d'améliorer le première affichage simple en dessinant les pièces et d'ajouter des couleurs.

[3] https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation

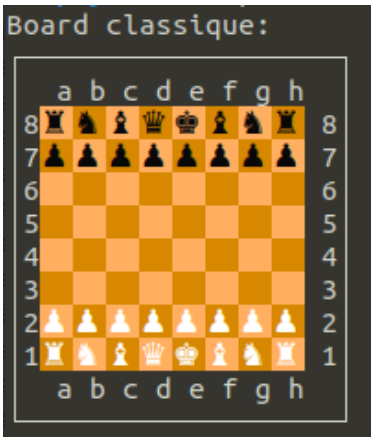


Figure 2: Exemples d'affichages des pièces avec images (codage Unicode) et couleur.

Expression du besoin :

Nous souhaitons obtenir un affichage comme dans la figure 3. Les pièces sont représentées par des images au dessin classique des pièces à la place des symboles FEN (de couleur blanche et noire) et le fond du plateau est aussi de couleur.

Pour réaliser cet affichage des pièces, il existe le codage Unicode des caractères représentant les pièces du Jeu d'échec disponible à cette url : `https://en.wikipedia.org/wiki/Chess\_symbols\_in\_Unicode`.

Pour réaliser l'affichage en couleur, les 256 couleurs du terminal sont utilisables. En effet, les couleurs du fond et du texte peuvent être modifiées à l'aide du caractère `ESC` suivi du caractère `[` et d'une certaine séquence qui permet de modifier les attributs d'affichage de la fenêtre. Le code ascii de `ESC` étant 27 (en décimal) ou 1b (en hexadécimal), la séquence `\x1b[a;b;cm` où a, b et c sont des nombres écrits en décimal, et le caractère `m` signifie que les paramètres SGR (Select Graphic Rendition) [4] sont utilisés :

- `\x1b[am` :
  - a=0 : remet les attributs dans l'état initial;
  - $30 \leq a \leq 37$: met la couleur du **texte** à la couleur "a-30";
  - $40 \leq a \leq 47$: met la couleur du **fond** à la couleur "a-40";
- `\x1b[38;5;nm` : met la couleur du texte du terminal à la couleur n compris entre 0 et 255;
- `\x1b[48;5;nm` : met le fond du terminal à la couleur n compris entre 0 et 255;

[4] pour plus de détails sur les couleurs voir https://conemu.github.io/en/AnsiEscapeCodes.html

Le choix des couleurs du plateau (couleur de fond) n'est pas imposé, mais il faut veiller à ce que l'alternance des cases noires et blanches soit visible. Notez que sur un plateau bien présenté la case a1 est une case noire, et que les deux reines commencent sur des cases de leurs couleurs.

Attendus : Il s'agit de réécrire la fonction `print_square` en ajoutant des fonctionnalités supplémentaires.

A3.1.a En partant de la fonction déjà écrite `print_square`, définir la fonction `print_square_color` qui doit réaliser l'affichage demandé lorsque les fonctions `set_background` et `set_foreground` sont définies.

A3.1.b Définir la fonction `set_background` qui affiche le fond d'une case.

A3.1.c Définir la fonction `set_foreground` qui affiche la pièce.

6.2 Construction de masques

But : Enregistrer et afficher des informations supplémentaires au plateau de jeu relatives aux déplacements de pièces.

Expression du besoin :

Afin de simplifier le traitement des informations liées aux déplacements de pièces ou à l'état du jeu, des informations sont enregistrées sous forme de masque. Un masque enregistre une information numérique pour chaque case du plateau. Les informations représentent par exemple les possibilités de déplacements, de prises d'une pièce, voire de plusieurs pièces.

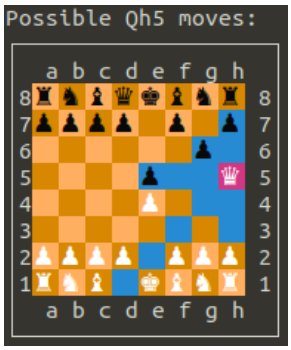
Les masques doivent servir à visualiser ces informations en les représentant à l'aide d'une couleur de fond. Par exemple dans la figure 3a, la couleur bleue met en évidence les déplacements de la reine blanche située en h5. Ces déplacements ayant été au préalable identifiés et sauvegardés dans un masque. Dans l'autre exemple de la figure 3b, le masque permet d'identifier par la couleur rouge les cases (et donc les pièces) qui peuvent être attaquées (dans ce cas-ci les pièces pouvant être prises par une pièce blanche). Le codage des valeurs numériques est de votre choix. La valeur 0 doit représenter le fond "classique" d'une case.

Attendus :

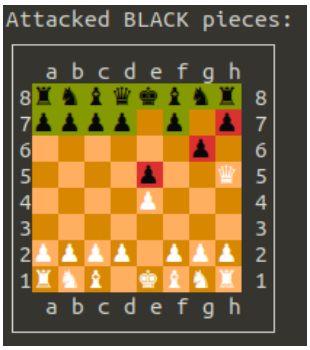
A3.2.a Définir une représentation du masque à déclarer dans le fichier `types.hpp`.

A3.2.b En suivant le modèle des fonctions de manipulation du plateau décrites dans `board.*`, définir dans un fichier source `mask.*` des fonctions permettant de manipuler des masques :

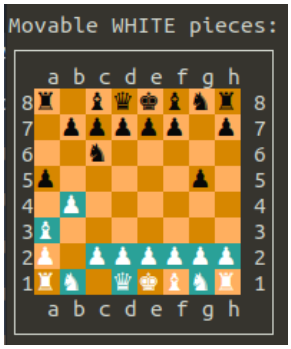
- `empty_mask` pour créer un masque sans information,
- `clear_mask` pour réinitialiser un masque aux valeurs 0,
- `get_mask` pour lire le contenu d'une case du masque,



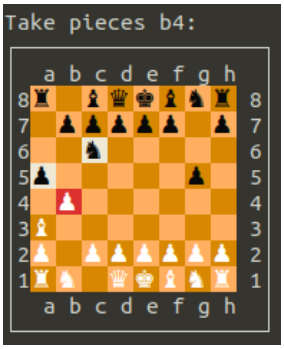
(a) en bleu, les déplacements possibles de la reine blanche.



(b) en rouge, les pièces noires attaquables, en vert non attaquables.



(c) en vert de bleu, les pièces blanches pouvant se déplacer.



(d) en bleu clair, les pièces noires pouvant attaquer le pion blanc en b4.

Figure 3: Exemples d'affichages de la mise en évidence de déplacements ou de prises

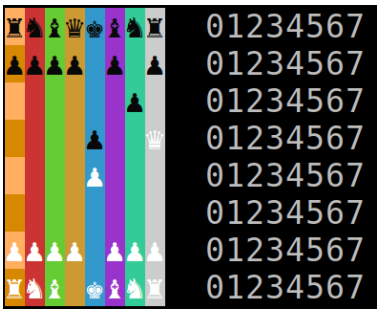


Figure 4: Exemple d'affichage du plateau avec les informations du masque donné à droite.

– `set_mask` pour modifier le contenu d'une case du masque.

A3.2.c Reprendre les fonctions décrites précédemment pour l'affichage (voir section 5.2), dupliquez-les (en utilisant le principe de surcharge expliqué ci-dessous) afin d'afficher le plateau et les informations contenues dans un masque.

A3.2.d Pour tester le nouvel affichage de masque, créer un masque contenant les valeurs données par la Figure 4 et écrire un exécutable pour afficher le plateau correspondant (la position des pièces importera peu).

La notion de *surcharge* d'une fonction existante permet d'utiliser le même nom d'une fonction déjà existante mais dont les paramètres sont différents (nombre et/ou type des paramètres). Les fonctions se différencient par le nom et aussi par la nature des arguments.

7 Séance 4: mouvements et masques

But : Créer de masques relatifs aux déplacements de pièces, en particulier le roi dans cette section.

Expression du besoin :
Une pièce peut se déplacer sur une case vide, ou sur une case contenant une pièce de la couleur adverse. Pour cela, un masque de déplacement est calculé. Chaque cases dans lesquelles la pièce peut se déplacer doit contenir une même valeur.

Techniquement, le masque de déplacement de la pièce en position (i, j) doit être calculé par la fonction `highlight_possible_moves`. Cette fonction doit faire appel à des sous fonctions spécialisées pour chaque pièce. En effet, le masque correspondant à chaque pièce doit être calculé à l'aide d'une fonction `highlight_possible_moves_piece` où `piece` sera remplacée par le nom de la pièce considérée. La fonction calcule les positions possibles (libre ou occupée

par une pièce adverse) en fonction de la position de la pièce (voir figure 3a pour la visualisation du déplacement d'une reine en h5). Au fur et à mesure de leur création, chaque fonction doit être testée à l'aide de la fonction d'affichage des masques de la section précédente.

Par souci de simplicité dans un premier temps, seules sont considérées les règles de déplacement basique des pièces. Par exemple, la prise en passant ou la mise en échec de son propre roi sont introduites dans la section **??**.

Attendus :

Lorsque vous aurez écrit les 3 premières fonctions (mouvement du roi et des tours) avec les tests correspondants, vous déposerez tout votre code (sections précédentes et 3 dernières fonctions) sur Moodle dans "Evaluation2_Séances2-4" **avant le 9 mai 10h**.

A4.1.a Définir la fonction `highlight_possible_moves` que vous enrichirez au fur et à mesure de l'écriture des sous-fonctions.

A4.1.b Définir la fonction `highlight_possible_moves_king` et le programme exécutable de test correspondant.

A4.1.c Définir la fonction `highlight_possible_moves_rook` et le programme exécutable de test correspondant.

8 Séance 5: mouvements et masques complets

8.1 Finalisation des mouvements

But : Créer de masques relatifs aux déplacements de toutes les pièces.

Expression du besoin :

Vous reprendrez les besoins de la section 7. Ici nous attendons que toutes les pièces au déplacements simples soient prises en compte.

Attendus :

A5.1.a Définir la fonction `highlight_possible_moves_bishop` et le programme exécutable de test correspondant.

A5.1.b Définir la fonction `highlight_possible_moves_queen` et le programme exécutable de test correspondant.

A5.1.c Définir la fonction `highlight_possible_moves_knight` et le programme exécutable de test correspondant.

A5.1.d Définir la fonction `highlight_possible_moves_pawn` et le programme exécutable de test correspondant. Attention aux règles concernant cette dernière pièce pour laquelle le déplacement est différent selon :

– sa position : s'il est sur sa ligne de départ il peut avancer d'une ou deux cases au maximum, et sinon il ne peut avancer que d'une case au maximum;
– s'il y a prise : il ne se déplace plus de la même façon.

8.2 Informations pour le coups à jouer

But : Créer des masques informatifs pour l'utilisateur.

Expression du besoin :
Nous souhaitons faire apparaître en couleur à l'aide de masques plusieurs informations à l'utilisateur :

(i) Toutes les pièces qui peuvent se déplacer au prochain coup (voir figure 3c).

(ii) Toutes les pièces qui peuvent être prises au prochain coup (voir figure 3b).

(iii) Toutes les pièces dangereuses à une pièce, c'est-à-dire celles qui peuvent prendre une pièce spécifiée du joueur (voir figure 3d).

Attendus :

A5.2.a Définir la fonction `highlight_movable_pieces` répondant au besoin (i) et le programme exécutable de test correspondant.

A5.2.b Définir la fonction `highlight_attacked_pieces` répondant au besoin (ii) et le programme exécutable de test correspondant.

A5.2.c Définir la fonction `highlight_take_pieces` répondant au besoin (iii) et le programme exécutable de test correspondant.

8.3 Menu de visualisation

But : Sélectionner et afficher les différentes informations disponibles.

Expression du besoin :
Avant de jouer un coup, un joueur peut avoir besoin de visualiser le déplacement de ses pièces, les pièces adverses qui peuvent être prises, ou encore ses propres pièces qui peuvent être prises et ainsi lui éviter de mauvaises surprises.

Nous attendons dans cette section de pouvoir afficher les différentes informations disponibles sur le jeu en cours.

Attendus :

A5.3.a Définir la fonction `mask_choices_menu` qui permet d'afficher les différents masques disponibles.

A5.3.b Définir la fonction `mask_choices` qui permet à un joueur de visualiser autant de masques qu'il le souhaite.

9 Séance 6 : boucle de jeu

9.1 Boucle de jeu : quelque soit la nature du joueur

But : Mettre en place les éléments d'un tour de jeu.

Expression du besoin :

Nous souhaitons mettre en place le code qui permet d'organiser un tour de jeu, c'est-à-dire qu'un joueur humain puisse saisir un coup après avoir pris connaissances d'informations, ou qu'un joueur automatique choisissent aléatoirement un coup.

L'état du jeu doit être contenu dans une structure de type `game`. Toutes les fonctions écrites dans cette section devront mettre à jour ce type.

Attendus :

A6.1.a Créer deux nouveaux fichiers `game.hpp` et `game.cpp` dans lesquels pour définir les fonctions écrites dans cette séance.

A6.1.b Définir la fonction `one_run` qui permet de faire jouer un coup que le joueur soit un humain ou un ordinateur. Cette fonction sera décomposée en 2 sous-fonctions `one_run_human` et `one_run_computer`.

A6.1.c Définir la structure des pièces prises, et ajouter à la structure `game` la (ou les) liste(s) des pièces prises.

9.2 Boucle de jeu : joueur humain

But : Écrire une fonction permettant à un humain de jouer un coup.

Expression du besoin :

Nous souhaitons que lorsqu'un joueur humain joue, il puisse visualiser (voir section 8.3) les informations sur le coup à venir. Ensuite, un joueur doit pouvoir saisir le coup (exemple : a3 vers b4) qu'il désire jouer. Une fois le coup saisi, un test doit vérifier la valider du coup proposé et mettre à jour les données nécessaires s'il est valide.

Attendus :

A6.2.a Définir la fonction `one_run_human` et le programme exécutable de test correspondant qui met en oeuvre l'ensemble des opérations pour qu'un joueur humain puisse jouer.

A6.2.b Définir la fonction `choose_mouvement_human` et le programme exécutable de test correspondant qui permet de saisir le mouvement d'un joueur.

A6.2.c Définir la fonction `test_run` qui est appelée dans la fonction précédente et qui renvoie `true` si le coup est possible et `false` sinon.

9.3 Boucle de jeu : joueur ordinateur

But : Écrire une fonction permettant à un ordinateur de jouer un coup aléatoire.

Expression du besoin :

Nous souhaitons que l'ordinateur effectue un coup valide aléatoirement. Il y a plusieurs façons de générer un coup aléatoirement. Dans une version simple, une pièce est choisie aléatoirement parmi toutes celles disponibles. Si un coup valide est possible à partir de celle-ci, une destination aléatoire est alors choisie, sinon une autre pièce est choisie aléatoirement.

Attendus :

A6.3.a Définir la fonction `one_run_computer` et le programme exécutable de test correspondant qui met en oeuvre l'ensemble des opérations pour qu'un joueur ordinateur puisse jouer.

A6.3.b Définir la fonction `choose_mouvement_computer` et le programme exécutable de test correspondant qui sélectionne le coup de l'ordinateur.