Seance 2:

Complétion du code de jeu.

(durée max: 2h)

Mot clé const et pointeurs:

En respectant la méthode de **programmation par contrat**, implémentez les autres fonctions de jeu.

➔ **Implémentez** `jeu_recupere_piece`

L'agorithme de cette fonction est proche de jeu_existe_piece.

Cependant elle retourne cette fois un **pointeur** vers la pièce courante.

Le pointeur retourné est un pointeur **non constant**. C'est à dire que l'objet pointé peut être modifié une fois retourné. Cela permet par exemple d'invalider une pièce ou de modifier ses coordonnées.

➔ **Implémentez** `jeu_recupere_piece_information`

Notez que cette fonction est quasi-similaire à jeu_recupere_piece.

L'unique différence consiste à passer en argument un **pointeur constant** vers un jeu, et l'on récupère un **pointeur constant** vers une pièce.

Le pointeur constant passé en argument permet de **certifier** que cette fonction **ne va pas modifier** l'état du jeu de pièces. De même, le retour de pointeur constant permet d'assurer que la valeur de la pièce restera **inchangée**.

Ce type de pointeur permet d'accéder aux valeurs (coordonnées, type), tout en certifiant que celles-ci ne seront pas modifiées.

Le mot clé **`const`** permet d'aider à la lisibilité du code en séparant les informations qui ne seront que lues, des informations qui vont être potentiellement modifiées.

➔ **Testez** les lignes de tests suivantes dans le main.

```
#include "jeu.h"
#include "constructeur_piece.h"

void initialisation_jeu(jeu *mon_jeu)
{
    piece *pointeur_piece_courante=&(mon_jeu->ensemble_de_piece[0]);
    constructeur_piece_tour(pointeur_piece_courante,1,5);
}

int main()
{
    jeu mon_jeu;
    initialisation_jeu(&mon_jeu);

    piece *piece_pointee_1=jeu_recupere_piece(&mon_jeu,1,3);
    piece *piece_pointee_2=jeu_recupere_piece(&mon_jeu,1,5);

    if(piece_pointee_1==NULL)
        printf("En (1,3), la piece retournee vaut NULL.\n");
    printf("La piece pointee en (%d,%d) est une %s.\n",
           piece_pointee_2->coord_x,
           piece_pointee_2->coord_y,
           nom_type_de_piece(piece_pointee_2->type));

    piece_pointee_2->type=fou;
    printf("Desormais, la piece pointee en (%d,%d) est un %s.\n",
           piece_pointee_2->coord_x,
           piece_pointee_2->coord_y,
           nom_type_de_piece(piece_pointee_2->type));

    return 0;
}
```

➔ **Commentez** le résultat obtenu étape par étape.

➔ **Modifiez** maintenant les deux lignes de déclaration de piece_pointee par la syntaxe suivante:

```
const piece *piece_pointee_1=jeu_recupere_piece_information(&mon_jeu,1,3);
const piece *piece_pointee_2=jeu_recupere_piece_information(&mon_jeu,1,5);
```

- ➔ Que se passe-il lors de la compilation, quelle ligne est indiquée en erreur ?
- ➔ **Supprimez** les deux dernières instructions, puis relancez le programme.
- ➔ Concluez sur l'utilité du mot clé **`const`**.

Vous devez être capable de manipuler le mot clé const à bon escient. Si vous ne comprenez pas cette partie, appelez un enseignant.

Algorithmique:

- ➔ **Implémentez** la fonction **`jeu_supprime_piece`**. Cette fonction sera appelée lorsqu'une pièce du jeu est supprimée par le joueur adverse.

A la fin de cette étape, vous disposez des classes de base de votre échiquier. Le reste du programme consiste à vérifier le bon comportement des déplacement des pièces.
Des fonctions du type `deplace_piece` *de (x0,y0) vers (x1,y1) devront être proposées pour simplifier la gestion globale du jeu.*

Compilation.

(durée max: 2h)

Le projet vous est fourni sans Makefile.
La compilation se réalise en appelant les lignes de compilations pour chaque fichier dans le script `compile.sh`.
Ce fichier est un ensemble de lignes qui sont lues et exécutées les unes derrière les autres.

Rappels sur la compilation:

On rappelle les étapes de la compilation:

1. **Compilation**: Transformation d'un fichier **source** (.c) en fichier **binaire objet** (.o)

Cette étape est réalisée pour chaque fichier source.
Elle peut se réaliser indépendamment d'un fichier source à un autre, on appelle cela la compilation séparée.

Pour lancer la compilation d'un fichier source vers un fichier objet, on utilise la syntaxe suivante en ligne de commande:

```
$ gcc -c fichier.c -o fichier.o
```

L'option *-c* permet d'indiquer l'arrêt du processus au niveau du fichier objet.

2. **Edition de liens**: Permet de passer d'un ensemble de fichiers **binaires objets** (.o) en un seul fichier binaire **executable**.

Pour chaque exécutable, il n'y a qu'un seul appel à l'édition de liens.
L'édition de liens prend comme entrée un ensemble de fichiers objets (.o) et fournit en sortie un exécutable.
L'édition de liens peut être vue comme l'assemblage de toutes les fonctionnalités en un seul endroit.
C'est également au moment de l'édition de liens que l'on se lie aux librairies.

```
$ gcc fichier_1.o fichier_2.o -o mon_executable
```

Remarques:

L'étape de compilation peut échouer s'il existe des erreurs de syntaxe, ou des chemins d'inclusion non valides (voir préprocesseur).

L'étape d'édition de lien peut échouer si

- un prototype de fonction est déclaré, utilisé, mais le corps de la fonction n'existe pas ou n'est pas accessible.
- une librairie n'est pas valide.
- il existe plusieurs ou aucune fonction “main”.

Le processus de compilation est lui même scindé en 3 étapes visibles:

1.a) Etape de **préprocesseur**
1.b) Création de code **assembleur**
1.c) Conversion en **binaire**

1.a) Preprocesseur:

Le préprocesseur est une première passe sur un fichier de code (.c) pour le rendre traitable par le compilateur. Il consiste en 2 points principaux:

- Suppression de tout les commentaires.
- Exécution des directives commençant par le signe #.

En particulier:
* Toute commande ou variable **Macro** définie à l'aide de `#define` voit sa valeur directement écrite dans le code.
* Les appels de type `#include` sont remplacés par l'intégralité du texte situé dans le fichier pointé.
* Les commandes de types `#ifdef, #ifndef, #else`, sont executées en tant que conditionnelles.

L'étape du préprocesseur est une partie programmable en **amont** du début de compilation du code C à proprement parler.
Le code issu du préprocesseur est visualisable. Pour cela, on utilise l'option **-E** de GCC.

Par exemple, on pourra écrire:
```
$ gcc -E mon_fichier.c > mon_fichier.i
```

➔ Utilisez cette commande sur les fichiers d'exemples fournies (répertoire `01_preprocesseur/`) et répondez aux questions décrites dans ces même fichiers.

Vous devez être capable de comprendre et d'utiliser les commandes du préprocesseur. Appelez un enseignant si vous ne comprenez pas.

Pour aller plus loin :
Pour les plus avancés, répondez aux questions décrites dans le répertoire `01b_preprocesseur_avance/`

1.b) Création du code assembleur:

Une fois l'étape de préprocesseur établie, le code issue de celui-ci est transcrit en assembleur spécifique au processeur.
La sortie du code assembleur peut être visualisée grâce à l'option de `gcc: -s.`

Par exemple,
`$ gcc -S mon_fichier.c`
Génère le fichier: `mon_fichier.s` contenant le code assembleur correspondant.

1.c) Code binaire:

Une fois le code assembleur généré, celui-ci est transcrit en binaire qui va pouvoir être lu par le processeur. Ce binaire est le **fichier objet .o**

Options de gcc:

Les différentes options de gcc sont disponibles ici:
http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

En particulier, on y retrouvera une liste exhaustive des options: de *debug*, *d'optimisation*, du *préprocesseur*, de l'*éditeur de liens*, de *l'assembleur*.

Rappel des options principales de compilation.

-g: Active les symboles de debug. Permet l'utilisation des debuggers sur l'exécutable (gdb, valgrind, …). Rend l'exécutable plus gros.
Lorsque vous développez, **utilisez toujours cette option**.

-O1 ou **-O2** ou **-O3**: Active les optimisations du code de niveau 1, 2 ou 3. Par défaut: niveau 2.
N'utilisez ces options que lorsque votre projet est terminé et lorsque celui-ci nécessite une optimisation en terme de vitesse d'exécution.

-I <PATH>: Permet d'inclure le chemin <PATH> dans les répertoires courants du projet.
Par défaut les répertoires courants sont: /usr/include/ , /usr/local/include, ...
Lorsqu'un fichier d'en tête est situé dans un repertoire autre. (ex. /home/mon_nom/mes_entete/), il est possible d'ajouter ce chemin dans le projet:
`$ gcc -I /home/mon_nom/mes_entete/ fichier.c -o mon_executable`

-D NAME=VALUE: permet de définir une variable globale NAME de valeur VALUE. (permet de se substituer à un "#define NAME VALUE" pour chaque fichier compilé.

-fPIC: permet de rendre l'adresse des fonctions indépendant de leur location. Permet la création de librairie dynamique (.so).

Options de Warnings:
Les warnings sont des aides précieuses permettant d'éviter des erreurs subtiles. Il faut toujours en tenir compte. Activez un maximum de warning lors de votre développement. Un projet final ne doit plus contenir de warning.
Une liste exhaustive est disponible ici:
http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options

-Wall -Wextra: sont les options de **Warning minimales** à **toujours** placer lors de vos compilations.

D'autres warnings sont également utiles pour vous assurer que votre code suit les règles de bonne programmation:

-**Wfloat-equal**: affiche un warning lorsque deux flottants sont comparés par un opérateur d'égalité (rappel: Ne jamais comparer deux float/double avec l'opérateur ==, ou !=)

-**Wshadow**: affiche un warning lorsqu'une variable locale en cache une autre.

-**Wswitch-default**: affiche un warning lorsque l'on oublie le cas "default" lors d'un switch.

-**Wswitch-enum**: affiche un warning lors de l'oubli d'un cas d'enum sur un switch.

-**Werror**: permet de transformer chaque warning en erreur: Permet de s'assurer que son programme ne contient plus de warnings.

-**Wwrite-strings**: affiche un warning lors de la conversion d'un const char[] vers char *.

-**Wpointer-arith**: affiche un warning lors de l'ajout/soustraction sur des pointeurs de type void*

-**Wcast-qual**: affiche un warning lors d'un cast supprimant un const.

-**Wredundant-decls**: affiche un warning lorsqu'une déclaration a lieu plusieurs fois.

-**Winit-self**: affiche un warning si une variable est initialisée avec elle-même.

- **pedantic**: affiche un warning si le strict respect du standard C n'est pas respecté.

Les options de l'éditeur de liens:

-l<NAME_LIB>: permet d'inclure la librairie: lib<NAME_LIB>.a ou lib<NAME_LIB>.so au projet.

-L <PATH>: permet d'inclure le chemin <PATH> dans le chemin de recherche standard de librairie.

➔ Utilisez les différents fichiers d'exemples pour manipuler certaines options de gcc (répertoire `02_compilation/`).

Pour aller plus loin :
Pour les plus avancés, répondez aux questions décrites dans le répertoire `02b_compilation_avance/`

Travail en autonomie

(durée estimé: 3h):

- Révision chaîne de compilation + fin des exercice d'exemples (2h).
- Rendu du travail + fin de complétion du code (1h)