Seance 1:

Découverte et mise en place des fichiers.

(durée max: 1h)

- ➔ **Observez** la structure globale du projet.
- ➔ **Retrouvez** les différents niveaux d'abstractions implémentés dans ce projet.

- ➔ **Remplissez** les cases: *Nom*, *Prenom* et *Groupes* de tous les fichiers suivant les règles indiquées dans les consignes.

Ces fichiers deviennent donc vos propres fichiers à titre nominatif. Tout fichier créé devra obligatoirement contenir cette syntaxe.

Notez qu'aucun Makefile n'est fourni. Ce sera à vous de le coder dans les séances futures.
Pour compiler le projet, on se servira du script compile.sh que l'on appellera de cette manière:

```
$ ./compile.sh
```

Note: Si le fichier compile.sh n'est pas reconnu en tant que programme exécutable, lancez la commande suivante:
$ chmod +x compile.sh
qui permet d'indiquer qu'un fichier est exécutable.

Note2 : Le symbole $ *indique une commande à lancer en ligne de commande. Ce n'est pas un caractère à taper.*

Vous disposez pour vous aider à la compréhension du projet, d'un binaire exécutable réalisant la solution minimale de ce projet. Ce binaire fonctionnera jusqu'à la troisième séance, au delà il cessera de fonctionner.

Ecriture de votre première fonction pas à pas, et méthode de développement par contrats.

(durée max: 3h)

Manipulation de piece:

Le fichier *piece.h* se compose de deux parties:
1/ La *première partie* comporte la **déclaration des types** et des **enumerations** associées à l'abstraction de piece.
2/ La *seconde partie* comporte les **signatures** (en-tête) des fonctions associées à piece.

- ➔ Observez la structure piece.
- ➔ Quelles valeurs peut prendre un type_de_piece ?

Vous devez être capable d'utiliser une enumeration, demandez à un enseignant si vous ne comprenez pas cette partie.

- ➔ Dans la fonction main, tapez:

```
int main()
{
   piece p;
   p.type=tour;
   p.coord_x=5;
   p.coord_y=8;
   printf("%d %d %d\n",p.type,p.coord_x,p.coord_y);

   return 0;
}
```

- ➔ Expliquez l'affichage obtenu.

Vous devez être capable d'afficher des variables de type:
int (%d) ; float/double (%f) ; chaine de caractere (%s)
à l'aide de ***printf***. Demandez à un enseignant si vous ne comprenez pas cette partie.

- ➔ Quelles valeurs peuvent être affichées suivant le type de piece?
- ➔ Que peut on conclure si la commande suivante affiche “2” ?

```
printf(“%d \n”,p.type); //p etant une struct de type piece.
```

La fonction "*nom_type_de_piece*", permet de réaliser la correspondance entre un type_de_piece et sa chaîne de caractère.

➔ Tapez et commentez le résultat de l'appel suivant:

```
int main()
{
    piece p;
    p.type=reine;
    printf("Ma piece est une %s \n",nom_type_de_piece(p.type) );

    return 0;
}
```

Implémentation naive:

La fonction *piece_ecrit_coordonnee* permet d'ecrire les coordonnees (x,y) dans la structure de *piece* passée en paramètre.

Notez:
- Que le mot **piece** est répété en début de fonction afin de renseigner sur quelle **structure** agit cette fonction.
- Que le **premier paramètre** de cette fonction est un **pointeur** vers une **struct piece** (voir règles de codage).
- Que les paramètres suivants (les coordonnées à compléter) sont passées par **copie**.
- Que les commentaires avant la signature de la fonction décrivent la fonction suivant une règle de type "**programmation par contrat**".

➔ Implémentez la fonction de cette manière:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int
nouvelle_coordonnee_x,int nouvelle_coordonnee_y)
{
    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

➔ Quels sont les défauts de cette implémentation?

➔ Ecrivez les lignes suivantes dans le fichier main:

```
int main()
{
```

```
    piece p;
    piece_ecrit_coordonnee(&p,5,6);
    printf("%d %d\n",p.coord_x,p.coord_y);

    piece_ecrit_coordonnee(&p,5,12);
    printf("%d %d\n",p.coord_x,p.coord_y);

    piece *p2;
    piece_ecrit_coordonnee(p2,3,4);
    printf("%d %d\n",p2->coord_x,p2->coord_y);

    return 0;
}
```

- ➔ Qu'obtenez vous?
- ➔ Qu'est ce qui n'a pas été respecté vis à vis du contrat donné en commentaires dans le fichier .h ?

Apprendre à debugger:

- ➔ Initialisez désormais p2 au pointeur NULL.
  `piece *p2=NULL;`
  Est-ce que cela permet de debugger l'erreur ?

- ➔ En ligne de commande, lancez désormais le programme suivant:
  `$ gdb ./jeu_echec`
  Puis tappez:
  `(gdb) run`

- ➔ Observez les dernières lignes. Cela indique-il l'endroit de l'erreur?
  Pour obtenir d'avantage de précisions, tapez “`where`”.

Vous devez être capable de debugger vos programme vous même. Lorsque vous demandez de l'aide, vous devez au moins avoir localisé votre erreur mémoire. Demandez à un enseignant si cette partie n'est pas claire.

Cette méthodologie permet de déterminer où ont lieu les erreurs mémoire. Vous devez toujours être capable de localiser où a lieu une erreur mémoire.
Notez qu'il est indispensable de compiler avec le mode “debug” (option -g de gcc) pour obtenir ces informations.

Autre méthode de debug:

➔ Tappez
```
$ valgrind ./jeu_echec
```

➔ Observez la ligne "*Invalid write*": celle-ci indique le type et le lieu de l'erreur mémoire.

➔ Commentez l'appel à:
```
// piece_ecrit_coordonnee(p2,3,4);
// printf("%d %d\n",p2->coord_x,p2->coord_y);
```

➔ Recompilez et relancez votre programme.
Relancez:
```
$ valgrind ./jeu_echec
```

➔ Vérifiez que cette fois, aucune fuite mémoire n'est detectée.

Valgrind est un programme permettant (entre-autres) de détecter des fuites mémoires dans l'exécution d'un programme. Il détecte également les fuites mémoires n'aboutissant pas forcément à une "seg-fault".

Pour aller plus loin:
Il permet également de vérifier que toute mémoire allouée dynamiquement est bien dé-allouée avant de quitter le programme.

Code évitant les bugs:

Reprenez l'implémentation de la fonction piece_ecrit_coordonnee.
L'erreur mémoire peut être évitée lorsque deux conditions sont remplies:
- Le pointeur invalide est initialisé à **NULL**.
- La fonction vérifie que le pointeur qu'elle utilise n'est pas **NULL** (et donc valide).

Il est donc possible de modifier votre fonction pour prendre en charge cette fonctionnalitée:

➔ Complétez votre fonction avec cette syntaxe:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int nouvelle_coordonnee_x,int
nouvelle_coordonnee_y)
{
    if(piece_du_jeu==NULL)
    {
       printf("Erreur pointeur NULL\n");
       exit(1);
```

```
    }

    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

Note: *exit()* permet de quitter votre programme en renvoyant ici le code d'erreur 1 à la ligne de commande
Pour plus d'informations, tapez:
`$ man 3 exit`
en ligne de commande.

➔ Relancez votre code en ré-appelant l'appel sur **p2** initialisé à NULL.
Cette fois votre programme détecte bien l'erreur.

Analyse:

Il existe 2 désavantages au code que nous venons de mettre en place:

1- L'erreur est bien pris en charge, mais **l'endroit** précis de celle-ci n'est pas indiqué automatiquement.
Dans un projet pouvant contenir des milliers de lignes de codes et des centaines de fonctions, il est nécessaire de connaître plus de précisions pour debugger rapidement.

Pour aller plus loin:
Des macros spécifiques au C permettent de fournir des sorties automatiques sans avoir à ecrire manuellement le nom des fonctions:
Remplacer le printf affichant l'erreur par le suivant et observez la sortie:
printf("Erreur NULL pointeur: fonction %s \nLigne %d du fichier %s\n",__FUNCTION__,__LINE__,__FILE__);

2- La condition de validité "if" est exécutée en **permanence**, gaspillant de la ressource machine pour une condition normalement respectée.
Cela peut pénaliser le temps d'exécution de fonctions appelées un très grand nombre de fois. (ex. Récupération d'une valeur dans un tableau avec vérification des bornes plusieurs millions de fois).

Pour pallier à ces désavantages, il existe la fonction "**`assert`**" dont le rôle est de vérifier les **assertions.** C'est à dire des conditions de programmations qui sont supposées vraies lors du développement du logiciel.

Les assertions:

➔ Observez la page man de **`assert`** avec la ligne de commande:
`$ man assert`

Vous devez être capable de lire une aide des pages **man** (manual). Toutes les fonctions standards du C sont dans les pages man. Demandez à un enseignant si vous ne comprenez pas la page man.

Dans notre cas, la fonction devient alors:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int nouvelle_coordonnee_x,int
nouvelle_coordonnee_y)
{
    assert(piece_du_jeu!=NULL);

    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

➔ Modifiez votre code en suivant le modèle donné.

La fonction **`assert`** possède deux **avantages**:

1- Elle indique la **localisation** de l'erreur de codage rencontrée explicitement.

➔ Testez le programme et observez la ligne d'erreur. Vous devez être capable de la comprendre.

2- La vérification de la condition est réalisée uniquement lorsque le programme est compilé en mode de développement (dit de “debug”).
Lorsque la librairie/logiciel est terminé (on parle alors de mode “release”), la vérification n'est plus effectuée et évite le sur-coût de son évaluation.

➔ Pour indiquer que l'on ne souhaite plus évaluer les assertions, ajoutez l'option de compilation **-DNDEBUG** à chaque ligne de compilation (fichier compile.sh). Observez que l'erreur mémoire apparaît à nouveau.

Vous devez être capable de modifier les paramètres de compilation, appelez un enseignant si vous avez des questions.

Note: Lors de la compilation, l'ajout de l'option -D<X> est similaire à l'ajout de la ligne #define <X> en début de chaque fichier. Ici, cela revient à ajouter la ligne #define NDEBUG en début de chaque fichier. NDEBUG signifiant: No Debug.

Note importante: La fonction **`assert`** est particulièrement utile dans un but de debug lors du développement d'un code. Elle peut être largement utilisée de manière systématique afin de vérifier les certifications des fonctions.
Elle ne doit être utilisée que dans ce cadre là. C'est à dire pour détecter:

- Des **erreurs de codage** qui ne doivent jamais arriver une fois le projet fini.
- Des erreurs qui impliquent forcément un arrêt **immédiat** du programme suivit d'un debug du code.

En l'occurrence, on n'utilisera pas la fonction `assert` pour traiter:

- Des erreurs critiques qui doivent également être traitées dans le **code finalisé** compilé en mode "release".
  ex. Verification de l'espace disque ou RAM restant lors du remplissage d'une base de données, etc)
- Des erreurs non critiques devant être gérées spécifiquement pouvant avoir lieu dans la **version finale** du logiciel.
  ex. Entrée utilisateur incorrecte, chemin vers un répertoire inexistant, …

Vérification des contrats:

Le commentaire de la fonction écrit dans le fichier .h indique deux conditions sur les paramètres d'entrée. La seconde condition n'est cependant pas vérifiée dans l'implémentation de la fonction.

En particulier, la ligne:
```
piece_ecrit_coordonnee(&p,5,12);
```
Fonctionne correctement, alors qu'elle ne **vérifie pas le contrat** donné en commentaires.

Note importante sur le développement logiciel:
Les erreurs dites "*silencieuses*" sont les plus **difficile** à détecter et debugger.
En effet, une erreur critique ou de programmation (ex. Dépassement mémoire) est automatiquement détectée par le système (Segmentation fault) ou par le compilateur.
Une erreur non détectée par le système peut se poursuivre de manière *silencieuse* jusque dans le logiciel finalisé et aboutir à des comportements étranges ou à des failles de sécurité une fois distribué.
Une bonne programmation permet de **détecter** et corriger **le plus tôt possible** le comportement inapproprié des fonctions afin de prévenir tout comportement non attendu par la suite.

Par ordre de priorité:
1- Au moment de la compilation (le mieux).
2- Par assertions à l'exécution.
3- Au moment de tests unitaires.
4- Au moment de tests d'intégration.
5- Lors de l'utilisation en production/par le client. (le pire).

Dans le cas de notre fonction, il est possible et permis d'affecter la valeur 12 à un entier (niveau compilateur).
Cette valeur est cependant incorrecte vis à vis des *coordonnées d'un échiquier*. Il est donc important de ne pas permettre la manipulation de telles coordonnées dans les fonctions de notre jeu.

Le fait que les coordonnées passées en paramètres soient comprises dans l'intervalle [0,7] font parties du **contrat** passé avec le programmeur. La certification de cette fonctionnalité peut être vérifiée par la fonction **`assert`**.

Les deux syntaxes suivantes répondent au problème:

```
assert(coordonnees_x>=0);
assert(coordonnees_y>=0);
assert(coordonnees_x<=7);
assert(coordonnees_y<=7);
```

Ou bien:

```
assert(coordonnees_x>=0 && coordonnees_y>=0 && coordonnees_x<=7 &&
coordonnees_y<=7);
```

La seconde solution possède l'avantage de se réduire à une ligne, mais l'identification d'un problème éventuel est moins précise.

Pour aller plus loin:
En C, il aurait été possible de certifier que les valeurs sont positives par le compilateur, en déclarant les coordonnées comme des "unsigned int", et non des "int". La vérification ayant uniquement lieu vis à vis du nombre 7.

Fonction de certification de coordonnées:

Il est à prévoir que cette vérification ait lieu un grand nombre de fois dans diverses fonctions. Afin d'éviter la **duplication** de ces lignes de codes dans plusieurs fonctions, nous allons centraliser l'appel à ces vérifications en créant une nouvelle fonction d'aide permettant de valider si une coordonnée est valide.

La signature de cette nouvelle fonction est la suivante:

```
/**
 * Fonction est_coordonnee_valide:
 * ************************
 *    Verifie que deux coordonnees (x,y) sont valides sur un echiquier.
 *    C'est a dire que x et y sont deux entiers compris dans l'intervalle [0,7].
 *
 *    Necessite:
 *      - Deux coordonnees de type entieres
 *    Garantie:
 *      - Un retour valant 1 si les coordonnees sont valides.
 *      - Un retour valant 0 si les coordonnees ne sont pas valides.
 */
int est_coordonnee_valide(int coordonnee_x,int coordonnee_y);
```

Note: La condition nécessaire du type entier est directement vérifiée par le compilateur. Il n'est donc pas nécessaire d'ajouter de ligne de code spécifique.

➔ Implémentez cette fonction. Notez qu'il s'agit de la réponse à la question "*est ce que les coordonnees sont valides*". Le mot clé "***est***" est donc présent dans le nom de la fonction.

Test du bon fonctionnement de la fonction:

➔ Vérifiez la sortie de cette fonction en entrant différentes coordonnées (valides et non valides) dans la fonction main.

Exemple:

```
int main()
{
    int valeur_01=est_coordonnee_valide(0,0);
    int valeur_02=est_coordonnee_valide(1,3);
    int valeur_03=est_coordonnee_valide(7,7);
    int valeur_04=est_coordonnee_valide(-5,5);
    int valeur_05=est_coordonnee_valide(5,-5);
    int valeur_06=est_coordonnee_valide(-5,-5);
    int valeur_07=est_coordonnee_valide(8,5);
    int valeur_08=est_coordonnee_valide(8,12);
    int valeur_09=est_coordonnee_valide(14,12);
    int valeur_10=est_coordonnee_valide(14,-12);
    int valeur_11=est_coordonnee_valide(-14,12);

    printf("01: %d \n",valeur_01);
    printf("02: %d \n",valeur_02);
    printf("03: %d \n",valeur_03);
    printf("04: %d \n",valeur_04);
    printf("05: %d \n",valeur_05);
    printf("06: %d \n",valeur_06);
    printf("07: %d \n",valeur_07);
    printf("08: %d \n",valeur_08);
    printf("09: %d \n",valeur_09);
    printf("10: %d \n",valeur_10);
    printf("11: %d \n",valeur_11);

    return 0;
}
```

Pour aller plus loin:
Notez que l'on teste les 8 combinaisons possibles des valeurs fausses.
A l'aide d'une double boucle, il serait également possible de tester de manière exhaustive tous les cas valides.

➔ Utilisez désormais cette fonction pour simplifier l'appel à **`assert`** dans la fonction `piece_ecrit_coordonnee`

Le code final devra alors ressembler à:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int nouvelle_coordonnee_x,int
nouvelle_coordonnee_y)
{
    //vérification des donnees d'entrees
    assert(piece_du_jeu!=NULL);
    assert(est_coordonnee_valide(nouvelle_coordonnee_x,
nouvelle_coordonnee_y)==1);

    // La piece du jeu recoit les nouvelles coordonnees en x et y
    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

Notez l'ajout des **commentaires** aidant à l'explication du code.

Démarche de codage:

Notez que l'écriture d'une fonction de 2 lignes a nécessité une réflexion importante sur le **type d'erreur à traiter** et les **conditions d'utilisation** de cette fonction.

Pour chaque fonction que vous allez coder par la suite, il vous est demandé de respecter cette démarche:

- Commentaires
- Codage par contrat
- Vérification des assertions
- Tests des fonctionnalités

Votre évaluation portera en grande partie sur vos analyses, vos tests, vos commentaires et description du comportement, et respect des contrats.
Posez des questions à un enseignant si cette partie n'est pas claire.

Travail en autonomie.

(durée estimée: 2h)

- Révision cours, structure du projet, programmation par contrats, assertions. (1h)
- Avancement du code (1h)

Une pièce est **invalide** si son *type* ou ses *coordonnees* ne sont pas reconnue comme étant valide.
Pour différencier clairement une pièce invalide d'une pièce valide, la fonction `piece_invalider` place l'ensemble des coordonnées à la valeur -1 et le type à 0.

➔ **Complétez** la fonction `piece_est_valide()`.
Fonction qui renvoie 1 si la pièce est valide, et 0 sinon.

➔ **Observez** la structure jeu.
Un jeu est une structure contenant un **tableau** de pièces.
Ce tableau fait une taille fixe de 16. Ce tableau contient donc forcément toujours 16 pièces.

Pour différencier les pièces valides des pièces qui ont étés supprimés au cours du jeu, nous considérons qu'une pièce supprimée est une **pièce invalide placée en bout de tableau**.
Le début du tableau doit donc toujours commencer par un ensemble de pièces valides.

➔ **Complétez** la fonction

```
int jeu_compter_piece(const jeu *jeu_de_piece);
```

dont l'algorithme est le suivant:

```
//initialiser compteur a 0
//Pour toutes les pieces du jeu
//  Si la piece courante est valide
//     Incremente compteur
//Retourne compteur
```

➔ **Complétez** la fonction `jeu_existe_piece`.
Réfléchissez à l'algorithme, écrivez **le en commentaire** de votre fonction et **implémentez** celle-ci.

➔ **Vérifiez** vos fonction dans le main sur différents exemples.
Pour cela vous initialiserez un jeu, et vérifierez les sorties de jeu_existe_piece dans le cas où les coordonnées pointent: vers une piece, vers une case vide, en dehors des limites du jeu.

➔ **Déposez** votre projet (**avec vos tests**) sur les dépôts de fichiers avant la date limite.