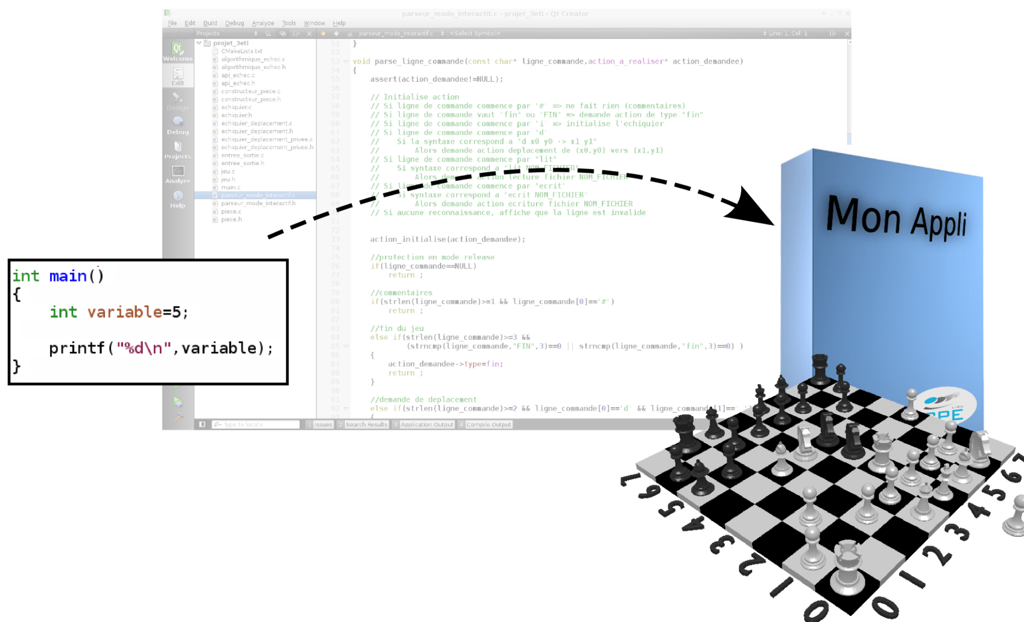
3ETI

2013/2014

LYON
CPE

Developpement logiciel en C

Fascicule de projet



Sujet 1:

Découverte et mise en place des fichiers.

(durée max: 30min)

- ➔ **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 sources (.c et .h) 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 *compilation_seance_1.sh* (pour la séance 1) et *compilation_projet.sh* (pour la suite) que l'on appellera de cette manière:

```
$ ./compilation_seance_1.sh
```

Note: Si le fichier *compilation_seance_1.sh* n'est pas reconnu en tant que programme exécutable, lancez la commande suivante:

```
$ chmod +x compilation_seance_1.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.*

- ➔ **Compilez** le programme et assurez vous d'avoir un exécutable dans le repertoire *bin/*.

- ➔ **Ouvrez** un terminal dans le répertoire *bin/* ainsi qu'un autre terminal dans le repertoire *src/*.

Ainsi pour compiler, vous pourrez écrire dans un terminal (répetoire *src/*):

```
$ ./compilation_seance_1.sh
```

Et exécuter directement à partir de l'autre terminal (répertoire *bin/*)

```
$ ./jeu_echec
```

Note: Pour le choix d'un éditeur de texte ou IDE afin d'écrire votre code, reportez vous à l'annexe en question. Notez également une annexe détaillant l'organisation des répertoires et précisant d'où les différents programmes doivent être exécutés.

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

(durée max: 1h30min)

Manipulation des pièce:

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 énumération, demandez à un enseignant si vous ne comprenez pas cette partie.

- ➔ Dans la fonction main, supprimer le code existant et tapez à la place:

```
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 “*nommer_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=dame;
    printf("Ma piece est une %s \n",nommer_type_de_piece(p.type));

    return 0;
}
```

Fonctions de pièce

Nous allons compléter les fonctions relatives au niveau d'abstraction d'une pièce.
Une pièce est formée d'un type et de deux coordonnées entières. Lors de manipulations plus avancées par la suite on évitera de manipuler directement ces champs manuellement, pour cela nous allons définir des fonctions de bases applicables sur un type *piece* qui faciliterons la manipulation de ce type.

La fonction **piece_est_valide** consiste à vérifier si une pièce peut être considérée comme valide ou non. La fonction devra retourner un entier qui possédera le rôle d'un booléen *vrai* ou *faux* si il vaut respectivement 1 ou 0.

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 sera un **pointeur** vers une **struct piece** (voir règles de codage).
- Les conditions suivants lesquelles une pièce est considérée comme valide est décrite avant la définition de la structure *piece* dans le fichier *piece.h.*

Nous détaillons la démarche permettant de construire cette fonction sous forme de tutorial.

Ecriture du contrat.

Ecrivons le contrat et la documentation de cette fonction (description dans le fichier *piece.h*).

Note: Les descriptions des fichiers .h sont les plus importantes car elles sont constamment lues afin de connaître les conditions d'utilisation des fonctions.

La description de *piece_est_valide* peut être la suivante:
```
Verifie qu'une piece passee en parametre possede un type et des
coordonnees valides.
```

Le **contrat** d'une fonction consiste à définir les conditions acceptables des paramètres d'entrée (pre-conditions) et les garanties à la sortie (post-conditions).

Dans le cas de *piece_est_valide*, nous supposerons en entrée que le pointeur reçu par la fonction n'est pas NULL et pointe bien vers une structure de *piece*.
La garantie en sortie est alors la suivante:
```
- Un retour valant 1 si le type de la piece est valide et que ses
coordonnees sont valides (c.a.d. comprises dans l'intervalle
[0,7])
- Un retour valant 0 sinon.
```

- ➔ Écrivez la documentation et le contrat de **piece_est_valide** dans le fichier *piece.h*.

Implémentation de la fonction.

Une implémentation triviale de la fonction serait la suivante:

```
int piece_est_valide(piece* piece_du_jeu)
{

    if(piece_du_jeu->type<=6 &&
            piece_du_jeu->coord_x>=0 &&
            piece_du_jeu->coord_x<=7 &&
            piece_du_jeu->coord_y>=0 &&
            piece_du_jeu->coord_y<=7)
    {
        return 1;
    }

    return 0;
}
```

Cependant, il existe déjà une fonction permettant de valider si les coordonnées sont valides. L'utilisation de cette fonction permet d'éviter des répétitions.

Note: Règle importante en programmation: Évitez toujours les répétitions de code!
Règle connue sous le nom de **DRY=Don't Repeat Yourself**.

- ➔ Ecrivez l'implémentation de la fonction *piece_est_valide* en utilisant la fonction *coordonnee_est_valide* pour la vérification des coordonnées.

- ➔ Pourquoi n'est-il pas nécessaire de vérifier *piece_du_jeu->type>=0 ?*

Il est également important de commentez le code. Les commentaires ne doivent pas être un redit du code, mais expliquer l'intention de l'utilisateur. Par exemple, on pourra écrire en commentaire l'algorithme suivant:

```
Algorithme:
Si les coordonnees sont valides, et que le type de piece est
valide
  Alors renvoie Vrai
Sinon
  Renvoie Faux
```

Test de la fonction.

Créez une fonction spéciale de test appelée *piece_tester_fonction_coordonnee_est_valide* sans paramètres.
Placez différents tests dans le corps de cette fonction, par exemple on pourra y écrire:

```
void piece_tester_fonction_coordonnee_est_valide()
{
    piece piece_1={pion_noir,0,5};
    piece piece_2={tour,7,5};
    piece piece_3={dame,4,9};
    piece piece_4={fou,-2,6};
    piece piece_5={17,5,3};

    //tests valides
    if(piece_est_valide(&piece_1)!=1)
        puts("Erreur piece_1");
    if(piece_est_valide(&piece_2)!=1)
        puts("Erreur piece_2");

    //tests invalides
    if(piece_est_valide(&piece_3)!=0)
        puts("Erreur piece_3");
    if(piece_est_valide(&piece_4)!=0)
        puts("Erreur piece_4");
    if(piece_est_valide(&piece_5)!=0)
        puts("Erreur piece_5");
}
```

Note: Pour l'aide sur la fonction puts, vous pouvez faire appel aux pages de manuels dites "pages man" à l'aide de la commande:

```
$ man 3 puts
```

Notez l'utilisation de l'initialisation directe d'une *struct* lors de la déclaration.

➔ Pourquoi, par défaut, cette fonction n'affiche-t-elle rien?

Note: L'avantage de coder des jeux de tests sous forme de fonction est de pouvoir les réutiliser plus tard si jamais on fait des modifications dans le code de la fonction testée.

➔ Appelez désormais depuis la fonction main le code suivant:

```
piece* piece_incorrecte=NULL;
piece_est_valide(piece_incorrecte);
```

On observe que le programme quitte brutalement en indiquant une *erreur de segmentation* ou *SegFault*.
Ce type d'erreur indique un problème d'accès mémoire à un endroit non autorisé. Ici, une tentative d'accès sur un pointeur NULL.

Un programme dit robuste ne doit pas finir en erreur de segmentation, ces erreurs sont à éviter car elles n'indiquent pas leurs origines. De plus, dans ce cas le contrat de la fonction n'a pas été respecté et cela n'a pas été détecté à temps. Pour cela, il est nécessaire de placer des barrières empêchant ce type de comportement.

Debug.

Lorsqu'une erreur de segmentation se produit, utilisez les outils de debugs appropriés.
Un rappel sur le fonctionnement de ceux-ci est disponible dans les fiches annexes.
Reportez vous toujours à ces outils lorsque vous rencontrez une erreur mémoire.

Résumé:

Invoquez la commande suivante:
```
$ valgrind ./jeu_echec
```

Notez l'indication de l'erreur à la ligne correspondante: Cette indication permet de trouver si votre programme présente des erreurs mémoires et à quelles lignes elles se produisent.

Invoquez également:
```
$ gdb ./jeu_echec
 (gdb) run
 (gdb) bt
 (gdb) quit
```

Notez que l'affichage de la ligne à l'origine de l'erreur est également affichée et la commande "bt" signifie "backtrace" et permet de connaître les appels de fonctions au moment de l'erreur mémoire.

Attention: Les erreurs mémoires ne sont pas toujours indiqués comme des erreurs de segmentations

et peuvent passer inaperçu. Il est de **votre responsabilité d'appeler valgrind** pour vérifier l'absence d'erreurs ou fuites qui passerait inaperçue. **La notation tiendra compte de l'absence de tout type d'erreur mémoire.**

Protection contre les erreurs mémoires.

Pour se protéger contre ce type d'erreur mémoire, une possibilité consiste à compléter la fonction avec les lignes suivantes:

```
int piece_est_valide(const piece* piece_du_jeu)
{
    if(piece_du_jeu==NULL)
    {
        puts("Piece du jeu est NULL dans la fonction piece_est_valide");
        abort();
    }
...
```

Cependant cette approche possède divers inconvénients:
- La syntaxe est lourde (3 lignes de codes).
- L'utilisateur doit préciser lui même l'endroit de l'erreur et devient source d'oublis par copier-coller.
- La vérification est faite en permanence alors qu'il ne peut s'agir que d'une erreur de code lors du développement du logiciel.

Une fonction est spécialement dédiée à ce type de protection: la fonction **assert**.
La fonction d'assert viens certifier que son paramètre est vrai. Dans le cas présent, cette protection se traduit par la syntaxe suivante:

```
int piece_est_valide(const piece* piece_du_jeu)
{
    assert(piece_du_jeu!=NULL);
 ...
```

Les avantages sont les suivants:

- La syntaxe est plus courte
- En cas d'erreur, la localisation est automatique
- La vérification disparaît lorsque le logiciel est considéré comme finale (attention, l'assert ne doit pas être utilisé pour des vérifications d'entrée utilisateur devant avoir lieu dans le logiciel finale, mais uniquement pour des erreurs de programmation).

➔ Observez le message d'erreur obtenue lorsque l'*assert* n'est pas vérifié.

Note: Toutes les préconditions liées à la programmation du code doivent être vérifié par des asserts.

Les vérifications d'entrées utilisateurs doivent être vérifiés par des conditions "if" classiques.

Pointeurs constants.

➔ Modifiez votre code de *piece_est_valide* de manière à ajouter cette ligne juste avant le "return 1;"

```
piece_du_jeu->type=tour;
```

C'est à dire que la condition est évaluée, mais que la pièce est modifiée par la suite.
Notez que les tests proposés ne détectent pas cette erreur.

➔ Proposez un test supplémentaire permettant de détecter cette erreur.

Note: La modification d'une valeur passée par pointeur est une source courante d'erreur. Il existe un moyen de s'en prémunir automatiquement grâce à l'aide du compilateur.

Un pointeur dont le contenu ne peut pas être modifié est qualifié de *const*. Cela certifie que le contenu du pointeur ne sera pas modifié. 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. Il s'agit d'un exemple de *documentation par le code*.

➔ Modifiez désormais l'en-tête de votre fonction (dans *piece.h* et *piece.c*) afin de satisfaire celle-ci:

```
int piece_est_valide(const piece* piece_du_jeu)
```

➔ Notez que la compilation échoue à la ligne où vous tentez d'affecter le type "tour" à la pièce.
➔ Supprimez la ligne d'erreur et notez que la compilation se déroule correctement cette fois.

Note: A chaque fois qu'un pointeur est passé en paramètre et que celui-ci n'a pas à être modifié, passez celui-ci en tant que *const*. Vous obtiendrez ainsi l'aide du compilateur qui pourra détecter rapidement des erreurs de programmation sans avoir à écrire les tests correspondants.

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.

- ➔ Réalisez une archive de votre projet vérifiant les consignes de rendus.
- ➔ Téléchargez le script de vérification des consignes. Et testez celui-ci sur votre archive (en suivant la démarche décrite dans la fiche technique "script de vérification des consignes"). Assurez vous que votre archive respecte bien ce script.

Note. Il vous est conseillé d'utiliser ce script avant chaque rendu. Il n'assure pas que l'ensemble des consignes sont respectés, mais permet d'éviter des oublis courants rendant votre travail non analysable.

Implémentation de l'invalidation:

- ➔ **Complétez** la fonction *piece_invalider* afin de respecter le contrat proposé. Notez que les post-conditions devront être vérifiées à l'aide d'assertions.

Jeu de pièces:

(durée max: 1h30min)

(Note: Avant la fin de séance, vous devriez au minimum avoir fini la fonction *jeu_compter_nombre_piece*.)

Nous allons dans la suite commencer à compléter le reste du projet en commençant par l'abstraction d'un jeu de pièces.

- ➔ **Incluez** dans le fichier main.c (#include) les fichiers d'en tête suivants: echiquier.h, entree_sortie.h et api_echec.h

- ➔ **Compilez** le projet à l'aide du second script de compilation *compilation_projet.sh*.

- ➔ **Vérifiez** que l'exécutable du projet est bien créé dans le repertoire bin/.

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. Cette structure contiendra soit les pièces d'un jeu blanc, soit les pièces d'un jeu noir sur l'échiquier.

Pour différencier les pièces valides des pièces qui ont étés supprimés/capturées 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.

La fonction *jeu_compter_nombre_piece* vient donner le nombre de pièces valides d'un jeu.
Pour cela, on viendra incrémenter un compteur en parcourant les pièces du jeu jusqu'à rencontrer une pièce invalide.

L'algorithme pourra être le suivant:

```
//Initialiser compteur <- 0
//Tant que compteur<MAX_PIECE et piece courante est valide
//  Incremente compteur si piece courante est valide
//Retourne compteur
```

➔ Implémentez la fonction *jeu_compter_nombre_piece* satisfaisant le contrat.

➔ Testez votre fonction en construisant des jeux fictifs (vous pouvez faire des fonctions de tests dédiées) et en vérifiant le nombre de pièces dans différents cas.

Aide:

- La k-ième pièce du jeu est accessible à l'aide de la syntaxe:
`jeu_de_piece->ensemble_de_piece[k]`

- L'adresse de la k-ième pièce du jeu est accessible à l'aide de la syntaxe:
`&jeu_de_piece->ensemble_de_piece[k]`

- Assurez vous que toutes les lignes de votre code soient les plus simple à lire. Évitez les lignes réalisant plus de une opération (appels de fonctions en chaînes, etc).
Par exemple, on évitera la syntaxe lourde du type:
`if(piece_est_valide(&jeu_de_piece->ensemble_de_piece[k])==1) ...`
Car celle-ci contient 3 opérations enchaînées.

On préférera scinder l'opération en éléments simples:

```
const piece *piece_courante=&jeu_de_piece->ensemble_de_piece[compteur];
int piece_valide=piece_est_valide(piece_courante);
if(piece_valide==1) ...
```

Lors du déroulement d'une partie, nous manipulerons les pièces d'un jeu en désignant leurs coordonnées. Nous aurons donc besoin de fonctions permettant de trouver des pièces d'un jeu étant

donnée leurs coordonnées.

- ➔ Implémentez la fonction *jeu_existe_piece* satisfaisant le contrat. On écrira l'algorithme en commentaire avant de coder la fonction.

- ➔ Implémentez la fonction *jeu_recuperer_piece* et *jeu_recuperer_pièce_information* satisfaisant le contrat.

Notez que la seule différence entre les deux fonctions est que l'une manipule un pointeur constant et l'autre non. Il peut dans certain cas être nécessaire d'avoir accès à une pièce d'un jeu en tant que simple information (quel est sont type, etc), et dans d'autres cas de récupérer la pièce pour la modifier. On utilisera ainsi l'une ou l'autre fonction.
Le corps des fonctions seront identiques, il s'agit d'un cas de fonctions ayant le même code (répétition) toléré pour la programmation.

- ➔ N'oubliez pas de tester vos fonctions.

Notez qu'il n'est pas possible de récupérer un pointeur *non const* à partir d'un pointeur *const*. L'inverse étant possible.

- ➔ Implémentez la fonction *jeu_recuperer_roi_information* satisfaisant le contrat.

Lorsqu'une pièce du jeu est capturée, celle-ci devient alors invalide. La structure de stockage du tableau impose que la pièce invalide soit placée derrière toutes les pièces valides. On pourra utiliser l'algorithme suivant:

```
//Parcours tableau jusqu'a trouver la piece designee
//Pour les pieces suivantes jusqu'a l'avant derniere
//  copier chaque pieces d'une position vers l'avant dans le tableau
//Invalider la derniere piece
```

- ➔ Implémentez la fonction *jeu_supprimer_piece* satisfaisant le contrat et suivant l'algorithme proposé.
- ➔ Prenez soin de tester votre fonction en construisant un jeu fictif, supprimant différentes pieces et vérifiant que votre jeu est bien correcte. On pourra également utiliser la fonction pré-codée *jeu_est_integre*.

Travail en autonomie.

(durée estimée: 3h)

- Révision cours, structure du projet, programmation par contrats, assertions. (1h)
- Avancement du code du projet (2h)

➔ **Testez** consciencieusement vos fonction sur différents exemples.

Pour cela vous initialiserez un jeu, et vérifierez les sorties de vos différentes fonctions dans le cas où les pré-conditions sont respectées, mais également lorsqu'elles ne le sont pas (pointeurs NULL, coordonnées invalides, coordonnées ne pointant pas vers une piece, etc).

➔ **Déposez** votre projet en suivant les consignes de rendus sur les dépôts de fichiers avant la date limite. (Aidez vous du script de vérification des consignes)

Sujet 2:

L’échiquier.

(durée max: 30min)

Les fichiers *echiquier.h* et *echiquier.c* contiennent les informations relatives à l'état d'un échiquier.
La structure **L’échiquier** est une structure contenant 2 jeux et un entier:

- le jeu des pièces blanches
- le jeu des pièces noires.
- le joueur courant (entier valant 0:blanc ou 1:noir).

L'état de l'échiquier peut être initialisé à l'aide de la fonction `echiquier_initialise`.

➔ Etant donné une variable *echiquier_courant* de type *echiquier*, quelle syntaxe permet de récupérer un pointeur vers la 4ème pièce blanche?

Afin de pouvoir manipuler plus aisément les joueurs courants et adverses dans les fonctions de contrôle du mouvement par la suite, on pourra faire appels aux fonctions:

```
echiquier_recupere_jeu_courant
echiquier_recupere_jeu_adverse
echiquier_recupere_jeu_courant_information
echiquier_recupere_jeu_adverse_information
```

➔ **Implémentez** ces 4 fonctions d'après les contrats de programmations indiqués.

Noter qu'il est demandé à plusieurs reprises de vérifier qu'un échiquier est valide. Pour cela, la fonction *echiquier_est_integre* est prévue. L'implémentation de cette fonction est laissée en tant que travail supplémentaire. Réfléchissez quelles critères doivent être vérifiés pour qu'un jeu d'échec soit viable.

Interface de contrôle de haut niveau: l'API.

(durée max: 1h30min)

La désignation d'**API** indique que ce sont les fonctions de ce fichier qui vont servir à interfacer le moteur du déplacement de pièce avec tout autre programme ou librairie.
En d'autre termes, vue de l’extérieur, un programmeur n'a besoin d'accéder qu'aux fonctions décrites dans l'**API** pour réaliser un jeu d’échec jouable.

Pour lancer le mode interactif de l'API, celle-ci doit être appelée par la fonction principale du programme.

➔ **Copiez** le code suivant pour lancer le mode interactif. **Compilez** le programme complet, executez le (certains lignes d'erreurs s'affichent car toutes les fonctions ne sont pas complètes, mais le programme ne doit pas s'arrêter). O**bservez** l'echiquier à l'aide du visualiseur.

```
int main()
{

    api_echec_lance_mode_interactif();
    return 0;
}
```

Une fois le projet réalisé, l'échiquier doit être commandable à l'aide de *phrases* envoyées par l'utilisateur en **ligne de commande**.

Par exemple:
`> d 1 3 -> 3 2`
Devra permettre de déplacer le pion situé en (1,3) vers la case (3,2).

Les autres commandes sont les suivantes:
`> i :` Initialisera l'état de l'échiquier à l'état initial.
`> # ceci est un commentaire:` Toute phrase commençant par # est un commentaire non analysé.
`> lit <nom_du_fichier> :` permet de lire et de charger l'échiquier contenu dans "*nom_du_fichier*".
`> lit_h <nom_du_fichier> :` permet de lire et de charger l'historique d'un échiquier contenu dans "*nom_du_fichier*".
`> fin :` Quitte le programme.

Chaque type de commande se traduit par une fonction de **l'API** du jeu.
Par exemple, la fonction `api_echec_deplacement_piece`, permet de demander le déplacement d'une pièce.
Ces fonctions sont considérées comme de "***haut niveau***". Leur paramètres ne doit pas faire apparaître **publiquement** des détails d'implémentation ou de contrat de code spécifiques.
Ce sont les fonctions qu'un **utilisateur externe** ne connaissant pas votre projet doit pouvoir **utiliser** facilement.

Le coeur du projet consiste à permettre le traitement correct du déplacement des pièces suite à la commande **`d`**.

La fonction `api_echec_deplacer_piece` est la fonction de gestion du déplacement des pièces et doit être écrite.

- ➔ **Observez l'algorithme** que doit satisfaire cette fonction.

Vérification de l'échiquier va consister à appeler la méthode permettant d'assurer qu'un échiquier est dans un état viable.

Les deux lignes suivantes de l'algorithme vont vérifier qu'un mouvement est demandé à des coordonnées plausibles, sans cela on indiquera que les coordonnées sont invalides, et on quittera la fonction pour demander une nouvelle entrée utilisateur.

La troisième ligne demande à vérifier qu'une pièce existe aux coordonnées de départs. Pour cela, il est nécessaire de récupérer le jeu courant, et de vérifier qu'une pièce existe bien à ces coordonnées. La quatrième ligne demande à vérifier qu'il n'existe pas de pièce de la même couleur que le joueur courant aux coordonnées d'arrivées.

- ➔ **Implémentez** les lignes de code correspondants à ce début d'algorithme (ne pas tenter de déplacer la pièce).

Notez que toutes les fonctions nécessaires sont déjà écrites, il ne s'agit que de les appeler et de vérifier leurs résultats.
Vous ne devez pas avoir à rentrer dans des champs de détails des structures des jeux ou des pièces. Lors du développement d'un logiciel, il est important de n'utiliser que des fonctions de hauts niveaux dans l'API, on ne doit limiter tant que possible les détails d'implémentations des structures à ce niveau (manipulation de tableaux, indice d'énumérations, etc), cela permet de réduire la complexité du code.

L'idée du reste de l'algorithme est le suivant:

- On tente de jouer le déplacement demandé.
- Si celui-ci à été réalisé, alors on enregistre ce coup dans l'historique des coups et on permute le joueur courant.
- Si le mouvement n'a pas été réalisé, alors on affiche une erreur.

Cet algorithme correspond aux appels suivants:

```
    code_erreur_deplacement code_erreur; // un code permettant d'enregistrer
                                         //  le type d'erreur
    code_erreur.type=pas_erreur;         // initialise le code d'erreur
    type_deplacement coup=deplacement_invalide; //un code permettant
d'enregistrer le type de deplacement

    echiquier_deplacement_realise_si_possible(echiquier_courant,
                                              x_depart,y_depart,
                                              x_destination,y_destination,
                                              historique,&code_erreur,&coup);

    if(code_erreur.type==pas_erreur)
    {
        historique_ajoute_coup(historique,
                               x_depart,y_depart,
                               x_destination,y_destination,coup);
        echiquier_changer_joueur_courant(echiquier_courant);
    }
    else
    {
        printf("Deplacement impossible\n");
        printf("Code erreur %d
(%s)\n",code_erreur.type,nommer_type_erreur(code_erreur.type));
    }
```

➔ **Copiez** ce code pour finir l'implémentation de la fonction.

Si vous tentez de déplacer une pièce du jeu blanc, cela ne fonctionne pas encore (mais le joueur doit changer de main). Analysons le code de la fonction *echiquier_deplacement_realise_si_possible:*

➔ **Observez** le contrat de cette fonction**.** (Notez que pour l'instant, on ne s'interessera pas aux coups spéciaux).

➔ **Observez** le corps de cette fonction et retrouvez les éléments suivants:

- Une série de vérifications préalables afin de détecter rapidement les erreurs de programmations.
- Un test de validité du déplacement du coup. Notez que pour l'instant, l'observation de l'implémentation de cette fonction indique que tout coup est reconnue comme valide indépendamment du type de pièce et des coordonnées.
- Une copie temporaire de l'état de l'échiquier
- La mise à jour de la variable coup lorsqu'un déplacement standard est reconnu comme valide.
- La demande du déplacement de la pièce sur l'échiquier temporaire lorsque le coup est reconnu comme valide.

- La mise à jour de l'échiquier courant par l'échiquier temporaire.

Remarque: A priori, l'utilisation de la copie du contenu de l'échiquier dans une variable temporaire semble inutile à ce niveau. En fait, cette mise en place est prévue dès maintenant afin de faciliter la prise en compte de la mise en échec plus tard. En effet, un coup vue comme valide et réalisé initialement, pourra être invalidé à posteriori si il met le roi en echec. Il sera alors aisé *d'annuler* un déplacement en ne mettant pas à jour l'échiquier courant par l'échiquier temporaire dans ce cas.

Le déplacement des pièces n'est pas encore possible car la fonction *echiquier_deplacement_realise_coup_standard* n'est pas implémentée.

➔ **Implémentez** cette fonction satisfaisant au contrat.

N'oubliez pas de vous servir des fonctions pré-codées du type: *echiquier_recupere_jeu_courant* et *echiquier_recupere_jeu_adverse*, *jeu_existe_piece*, et *jeu_supprimer_piece*. Ne recoder pas la même chose en rentrant à nouveau dans le détail de l'implémentation d'un jeu sous forme de tableau.

➔ **Vérifiez** que cette fois vous devenez capable de déplacer des pièces sur l'échiquier. Notez que tout mouvement est possible. Vérifiez également que la capture de pièce adverse se passe correctement.

Déplacement standards des pièces du jeu.

(durée max: 2h)

Un déplacement que l'on appellera *standard* consiste aux déplacements des pièces du jeu d'échec en dehors des coups spéciaux.
Les coups spéciaux sont: le roque, la promotion du pion, la prise en passant.
De plus, on ne considérera la mise en echec dans les déplacements standards (c.a.d qu'un mouvement sera possible, même si il laisse le roi du joueur courant en echec).

Chaque pièce possède des déplacements standards permit ou non. C'est le rôle de la fonction *echiquier_deplacement_coup_standard_est_valide* de vérifier si le déplacement est valide. Dans le cas présent, cette fonction retourne toujours vraie, validant ainsi tout déplacement.
Nous allons nous intéresser dans la suite à l'approche permettant de compléter cette fonction.

Considérons par exemple le cas d'un **cavalier** positionné aux coordonnées (x0,y0).
Supposons que l'on souhaite déplacer celui-ci des coordonnées (x0,y0) vers (x1,y1).
Nous avons déjà vérifié lors d'une étape préalable que (x0,y0) et (x1,y1) sont des coordonnées valides, qu'il n'existe pas de pièce alliée aux coordonnées (x1,y1).

Notons dx=|x1-x0|, et dy=|y1-y0|. Le déplacement d'un cavalier est possible si (dx,dy)=(2,1), ou si (dx,dy)=(1,2).

Ce type de raisonnement devra être généralisé à toute les pièces. Pour éviter d'avoir une grande fonction implémentant tout les cas possible, nous séparons la gestion du déplacement de chaque pièce séparément.
Vous trouverez ainsi en commentaire dans la fonction *echiquier_deplacement_coup_standard_est_valide* les appels prévus:

```
    int deplacement_est_valide=0;
    switch(piece_a_deplacer->type)
    {
    case tour:
deplacement_est_valide=echiquier_deplacement_est_valide_tour(echiquier_courant,piece_a_deplacer,x_destination,y_destination,erreur);
        break;
    case cavalier:
deplacement_est_valide=echiquier_deplacement_est_valide_cavalier(echiquier_courant,piece_a_deplacer,x_destination,y_destination,erreur);
        break;
    ...
```

Ainsi chaque déplacement d'un type de pièce sera analysé dans une fonction qui lui est propre.

Les fonctions *echiquier_deplacement_standard_est_valide_tour/cavalier/fou/...* sont des fonctions qui ne devraient pas être visible depuis l'API. Ce ne sont pas des fonctions que l'on souhaite appeler directement sans passer préalablement par l'appel à *echiquier_deplacement_coup_standard_est_valide.*

Pour modéliser ce type de comportement, on implémentera ces fonctions dans un nouveau fichier qui ne sera inclue que par *echiquier_deplacement.c*. Cette implémentation restera d'une certaine manière *privée* par rapport à l'utilisation de l'API.

- ➔ **Créez** deux nouveaux fichiers *echiquier_deplacement_standard.c* et *echiquier_deplacement_standard.h*.
- ➔ **Complétez** ces deux fichiers avec les informations de vos noms prénoms et groupe afin de le rendre conformes aux consignes. Placez y également les *includes guards* (#ifndef, #define ...).
- ➔ **Ajoutez** la ligne nécessaire dans le script de compilation pour prendre en compte ce nouveau fichier.
- ➔ **Ecrire** la signature de la fonction *echiquier_deplacement_standard_est_valide_cavalier*. Au vues de conditions déjà étudiée, écrivez sont contrat.
- ➔ **Implémentez** cette fonction.
- ➔ **Décommentez** l'appel à la fonction *echiquier_deplacement_est_valide_cavalier* dans le fichier *echiquier_deplacement.c* de manière à faire appel à cette fonction lorsque vous demandez le déplacement d'un cavalier.
- ➔ **Testez** le déplacement du cavalier en mode interactif manuellement.

Travail en autonomie

(durée estimée: 3h):

- ➔ **Finissez** de l'implémentation du cavalier si nécessaire.
- ➔ En vous inspirant de l'approche du déplacement du cavalier, **implémentez** de manière similaire le déplacement du roi.
- ➔ **Lire** la documentation sur la compilation pour la prochaine séance.
- ➔ **Faire et répondre** aux questions du tutoriel sur les étapes du préprocesseur et de compilation.

Pour cela, vous devez télécharger l'archive contenant les tutoriaux d'entraînements.
Les programmes correspondants aux préprocesseurs et à la compilation sont dans les répertoires 01_preprocesseur, et 02_compilation.
Pour chaque répertoire, il y a 3 niveaux d'exercices:

- *a_minimal*: A faire obligatoirement avant la semaine suivante, ce sont les bases minimales à maîtriser absolument avant la séance suivante.
- *b_recommandé*: Notions recommandées à connaître. Couvre différents points qui sont supposés être maîtrisés pour le partiel final.
- *c_avancé*: Notions avancées pour les étudiants souhaitant aller plus loin. Ces notions ne seront pas exigibles pour le partiel.

Note: Les réponses ne seront pas ramassées: posez des questions si vous n'êtes pas sûr. Les points couverts se retrouveront dans le partiel écrit.

Sujet 3:

Compilation.

(durée max: 1h)

Jusqu'à présent le script de compilation vient compiler chaque fichier l'un derrière l'autre de manière systématique. Nous avons pris soin de définir l'ordre dans lequel chaque compilation doit avoir lieu, et nous compilons chaque fichier même si celui-ci n'a pas été modifié.
Ceci entraîne un temps de compilation allongé non compatible avec des projets volumineux.

Nous allons utiliser l'outil **`make`** pour simplifier et enrichir la gestion de ce processus de compilation.

`Make` est un outil standard permettant d'automatiser et d’enchaîner des taches de manière simple.

Note: Make est totalement **indépendant** du compilateur `gcc`, et peut être utilisé pour réaliser d'autres tâches que la compilation.

L'appel à `make` en ligne de commande vient lire un fichier du nom de **Makefile**.
La syntaxe associée dans le fichier Makefile suit le modèle suivant:

```
fichier_a_realiser : dependances
	ligne(s) de commandes a executer
```

Explication:
fichier_a_realiser = Le fichier que l'on cherche à créer (ou nom de l'action à réaliser).
dépendances = Le ou les fichiers qui doivent être présents afin de créer ce fichier
ligne(s) de commandes à exécuter = La commande ou les commandes qui doivent être lancées afin de réaliser ce fichier. Chaque ligne est précédée d'une **tabulation** (et non d'espaces). Dans le cas d'actions standards (compilation par ex.) cette ligne peut être inférée automatiquement par **make** et devient alors optionnelle.

- ➔ **Réalisez** les tutoriaux d'entrainements sur le Makefile du niveau minimal.
- ➔ **Ecrivez** le Makefile correspondant au projet.

Mouvements des pièces.

(durée max: 3h)

- ➔ **Implémentez** le mouvements de pièces du jeu: tour, fou et dame. Essayez d'écrire un code lisible et concis en factorisant les parties qui peuvent l'être.

- ➔ **Testez** que vos mouvements se passent bien à l'aide du mode interactif.

Travail en autonomie

(durée estimée: 3h):

- Réalisez les tutoriaux recommandés sur les Makefiles.
- Finalisez l'implémentation du mouvements de vos pièces du jeu.

Sujet 4:

Mouvements des pièces.

(durée max: 1h)

- ➔ **Finissez** l'implémentation du mouvements de vos pièces avec le pion_blan et le pion_noir.
- ➔ **Testez** que vos mouvements se passent bien à l'aide du mode interactif.

Création de fichiers de tests.

(durée max: 3h)

Au fur et à mesure du développement de votre projet, il est important de réaliser des tests.

- Des **tests unitaires** après l'écriture de chaque fonctionnalité importante qui vont tester une ***fonctionnalité "unitaire"***.
- Des **tests d'intégrations** après un certain nombre d'étapes qui vont tester un ***comportement global***.

Le **développement par tests** consiste à écrire les tests des fonctionnalités à vérifier **avant** l'implémentation de la fonction elle même.
Il est en effet souvent plus facile de décrire les **appels** que l'on **souhaite** et la **sortie attendue**, plutôt que de décrire directement comment la fonction va être implémentée.

Nous avons déjà vu comment réaliser des tests au niveau du code et des fonctions dans les TP précédents. Dans cette partie, nous allons tester le comportement du programme au complet en envoyant une entrée textuelle d'un ou plusieurs déplacement, et en observant la sortie attendue.

- ➔ **Observez** les fichiers "*etat_echiquier.txt*" qui encodent l'état de l'échiquier après chaque déplacement. L'analyse de ces fichiers permet de définir si un coup c'est correctement passé ou non en comparant celui-ci à une sortie attendue.

Mise en place des tests de l'application:

L'application doit permettre de déplacer les pièces de manière cohérente vis à vis d'une partie d'échec.

Par exemple, la combinaison

```
> i
> d 2 1 -> 2 2
> d 3 6 -> 3 5
> d 3 1 -> 3 2
> fin
```

Doit aboutir à un état de l'échiquier **valide** et connue (3 pions ont été déplacés d'une case).

Au contraire, la combinaison:

```
> i
> d 2 1 -> 2 5
> fin
```

Doit aboutir à la détection d'un coup **invalide** (un pion ne pouvant pas se déplacer de 4 cases vers l'avant).

Notez que ces combinaisons peuvent être écrites **dès maintenant** même pour des mouvements non encore implémentés. Dans un code bien construit, ces tests de l'application ne dépendent pas du codage interne de la structure de données utilisée pour manipuler l'échiquier.
Ces tests doivent donc pouvoir être exécutés, peu importe le code interne.
Par exemple, les tests des binômes voisins peuvent être par la suite appliqués sur votre code.

Exemple de fonctionnement:

Nous détaillons le fichier de test donné en exemple test_unitaire_ok_01:
Ce test tente de déplacer un pion sur l'échiquier.
La première ligne consiste à la lecture d'un état d'échiquier simple comportant peu de pièces (un roi et un pion pour chaque joueur).
La seconde ligne demande le déplacement du pion des coordonnées (1,1) vers (1,2).
Ce déplacement doit normalement aboutir, et la troisième ligne demande la fin du programme.

Note:

- Il s'agit bien d'un test unitaire comportant le test d'un seul déplacement d'un pion blanc.
- La première ligne de lecture d'un fichier est optionnelle, on aurait pu se contenter de l'organisation de l'échiquier par défaut en n'indiquant rien, ou un "i".

Pour lancer le test, on appelle en ligne de commande (en supposant que l'on soit dans le répertoire bin/)

```
./[nom_excutable] < ../test/test_unitaire_ok/test_unitaire_ok_01.txt
```

Note: Le sigle "<" indique la redirection de l'entrée standard. C'est à dire que le programme ne va plus lire son entrée à partir du clavier, mais viens directement la lire dans le fichier indiqué.

L'analyse du résultat du test peut se faire de manière visuelle en observant le fichier de sortie "echiquier_etat.txt" ou en observant le résultat à l'aide du visualiseur.
Il est cependant commode d'automatiser le processus d'analyse lorsqu'on relance les tests à plusieurs reprises.

Un troisième fichier est alors utile dans ce cas, le fichier *test_unitaire_ok_01_sortie.txt* indique la sortie attendue de l'état de l'échiquier après exécution du test.
Il est possible de demander la comparaison ligne à ligne entre ce fichier de sortie attendu et le fichier "echiquier_etat.txt" actuel.

Un script réalise cette comparaison automatiquement en parcourant tout les tests les un derrière les autres, et indique si une erreur est détectée par rapport à la sortie attendue.
Pour cela, déplacez vous dans le repertoire script_test_automatique/ et lancez la commande:

```
$ python run_test.py
```

La sortie vous indique pour chaque test si la sortie attendue est bien la bonne.

Note: Cette démarche fonctionne également pour les tests dits de "ko". Dans ce cas, la sortie attendue ne doit pas prendre en compte le déplacement qui doit échouer.

Travail demandé:

Il vous est demandé d'établir la série de tests qui viendra **valider** que votre projet réalise bien la fonctionnalité d'un jeu d’échec en écrivant les fichiers de déplacements, ainsi que les fichiers de sorties attendues (et potentiellement les fichiers d'entrées si nécessaire).
Pour cela, vous allez écrire un ensemble de tests **unitaires** et **d'intégrations**.
Ils devront valider à la fois les coups **possibles** (à placer dans test_unitaire_ok), et également confirmer que les coups **invalides** (à placer dans test_unitaire_ko) sont bien détectés comme tels.

Vos tests devront couvrir un **maximum de cas possibles** et respecter la **structure** décrite dans l'annexe concernant les fichiers de tests.
Réfléchissez à une **stratégie** adéquate.

Vous garderez ces tests tout au long de votre projet. Ils vous permettront de valider votre projet et vous aideront à ne pas oublier de cas particuliers lorsque vous coderez. Si ces tests sont insuffisants, ils ne pourront pas valider clairement votre travail.

Quelques conseils:

- Ne passez pas l'ensemble de vos tests à observer le déplacement valide d'un pion à différents endroits de l'échiquier. Mais **couvrez** l'ensemble des pièces possibles.

- Réaliser au minimum **1 test** de validité **par type** de déplacement
  (ex. Pion avançant en ligne droite, pion avançant en diagonale lorsqu'un adversaire est présent, notez que les pions noirs et blancs n'avancent pas dans la même direction. Tour avançant suivant x>0, cas d'une tour avançant suivant x<0, cas d'une tour avançant verticalement, présence d'une pièce adverse ou non, etc.)

- Pour les tests unitaires et pour les tests de fonctionnalité invalides, réalisez un **unique test** par fonctionnalité. Un test unitaire ne devrait ainsi contenir idealement qu'un seul déplacement.
  (ex. Ne testez pas en même temps la tour et le fou, ne testez pas l'un après l'autre deux coups invalide puisque seul le premier passera à l'exécution, …)

- Réalisez quelques tests **d'intégrations** aboutissant à une configuration de plateau plus complexe.

- Pensez bien aux cas de coups **invalides**, aux **cas particuliers**.
  Etablir les tests des cas particuliers dès maintenant permet de ne pas les oublier plus tard au moment du développement de code.

- Utilisez les **outils** à disposition pour simplifier la mise en scène des tests: *lecture de fichiers de scènes, initialisation*.

- Tout comme le code, vos tests doivent être commentés et satisfaire aux conditions de rendus.

Travail *minimum vitale* demandé:

- Au minimum 1 test valide ainsi que 1 test invalide (avec un fichier d'entrée, et un fichier de sortie attendue) par type de pièce.
- Au minimum 1 test valide attendu d'un coup spécial (potentiellement non encore implémenté) tel qu'un roque, une prise en passant, ou l'invalidation d'un mouvement pour cause de mise en echec.
- Au minimum 2 tests d'intégrations vérifiant que l'agencement de plusieurs coups fonctionnent correctement.

Attention: Tous vos fichiers de tests devront respecter l'organisation et la syntaxe décrite dans l'annexe sur les fichiers de tests.

Note: Vos scripts de tests seront notés. La notation tiendra compte de l'exhaustivité de vos tests, de la précision des tests unitaires, des commentaires.

Travail en autonomie.

(durée estimée: 3h)

- Fin des déplacements (1h)
- Fin des tests d'application (2h)

Sujet 5:

Ecriture et lectures de fichiers:

(durée max: 3h)

Etat de l'échiquier:

A chaque tour de jeu, l'état de l'échiquier ainsi que l'historique est écrit textuellement dans un un fichier sur le disque dur.
La gestion des écritures/lectures dans un fichier est regroupée dans les fichiers *entree_sortie.c* et *entree_sortie.h*.

Toutes les fonctions sont écrites en C, par contre seul la fonction de lecture de l'état de l'échiquier à partir d'un fichier à été écrite de manière lisible.
Les autres fonctions ne suivent pas les standards de lisibilités et doivent être réécrites.

L'algorithme d'écriture de l'échiquier est le suivant:

```
// Ouvre fichier en ecriture
// Ecrit Joueur 1
// Ajouter etoile (*) si joueur_1 est le joueur courant
// Pour toutes les pieces de joueur_1:
//   Ecrire "type coordonnees_x coordonnees_y" de la piece courante
// Fin Pour
//
// Ecrit Joueur 2
// Ajouter etoile (*) so joueur_2 est le joueur courant
// Pour toutes les pieces de joueur_2:
//   Ecrire "type coordonnees_x coordonnees_y" de la piece courante
// Fin Pour
// Ferme fichier
```

Rappel:

L'ouverture d'un fichier se réalise par la commande *fopen*.
La fermeture d'un fichier se réalise par la commande *fclose*.
L'écriture textuelle dans un fichier se réalise par la commande *fprintf*.
La syntaxe de *fprintf* est similaire à la syntaxe de *printf* mise à part que le premier paramètre est le descripteur de fichier retournée par *fopen*.

A titre d'exemple, voici un code permettant d'écrire dans un fichier.

```
int main()
{
    const char *filename="mon_nom_de_fichier.txt";

    FILE *fid=NULL; //struct contenant un descripteur de fichier
    fid=fopen(filename,"w"); //ouverture du fichier “filename” en mode ecriture

    if(fid==NULL)
    {
        printf("Erreur ouverture du fichier %s\n",filename); exit(1);
    }

    fprintf(fid,"J'ecris dans un fichier le nombre %d \n",3);
    fprintf(fid,"Et je sais que 2+2=%d",2+2);
    fprintf(fid,"Enfin si j'ai une piece de type tour, j'ecrirai le nombre
%d\n",tour);

    int c=fclose(fid);
    if(c!=0)
    {printf("Erreur fermeture fichier %s\n",filename);exit(1);}
}
```

Vous devez dans la suite, réécrire la fonction *entree_sortie_ecrit_fichier_echiquier* de manière lisible. Son comportement devra cependant être similaire à la fonction illisible originale, prenez soin de bien vous assurez que les fichiers d'états écrits seront identiques.

- ➔ **Implémentez** l'écriture de l'état de l'échiquier dans un fichier.
- ➔ **Testez** votre code sur différents états de l’échiquier.
- ➔ **Assurez** vous que votre fichier d'état soit bien identiques aux fichiers écrits par la fonction illisible (même ordonnancement, etc).

Note: Pour vous assurer que deux fichiers sont identiques en tout point, la commande
`diff fichier_1 fichier_2`
permet d'afficher toutes différences existantes.

Historique:

De manière similaire, les fonctions de lecture et d'écritures de l’historique doivent être ré-écrite de manière lisibles.

- ➔ **Assurez** vous de bien comprendre le contenu des fichiers d'historiques.
- ➔ **Implémentez** l'écriture de l'état de l'historique dans un fichier.

- ➔ **Réfléchissez** à l'algorithme d'écriture de l'historique à partir d'un fichier.
- ➔ **Implémentez** la lecture de l'état de l'historique à partir d'un fichier.

Correction de votre programme d'après résultats des tests:

(durée max: 1h)

- ➔ **Corrigez** si nécessaire votre programme après application des tests de la séance précédente.
- ➔ **Ajoutez** des tests supplémentaires si nécessaire.

Travail en autonomie.

(durée estimée: 3h)

- Début de rédaction du compte-rendu (2h)
- Ajout de tests supplémentaires et correction du programme (1h)

Sujet 6:

Rendu d'un logiciel livrable minimaliste.

(durée max: 4h)

➔ **Finalisez votre projet.**

Note:

- *Avez-vous suivit les règles de bonne programmation?*
- *Assurez vous que l'ensemble de vos fonctions sont correctement commentées.*
- *Assurez vous de respecter une programmation par contrats. Les fichiers .h sont ils commentés, les corps des fonctions respectent-ils les contrats?*
- *Gérez-vous correctement le mot clé const?*
- *L'ensemble des tests-ont ils été réalisés ? Tous les cas d'erreurs sont-ils testés?*
- *Votre* `Makefile` *est-il à jour, est-il lisible? Peut-on factoriser des arguments?*

➔ Synthétisez les tests effectués et expliquez votre démarche ainsi que vos choix dans un **compte-rendu** de 5 à 10 pages.

Travail supplémentaire.

Différentes améliorations peuvent être mises en places.
En priorité, on s'intéressera au cas de **mise en echec**.
Reportez vous à l'annexe concernant la mise en échec pour l'implémentation de cette partie.

Si la mise en échec fonctionne correctement, vous êtes libre de choisir les améliorations à apporter parmi les choix suivants (le nombre de + indique la difficulté prévisionnelle):

- Gestion de l'échec et mat (+)
- Gestion des coups spéciaux:
  - Prise en passant du pion (+)
  - Roque (++)
  - Promotion du pion (+)
- Gestion du pat :
  - Par absence de coup possible (+)
  - Gestion complète, règle des 50 coups etc (+++)
- Création d'une librairie dynamique pour la gestion du deplacement (+)
- Création d'un intelligence artificielle (++++)

Pour chaque amélioration, reportez vous à l'annexe correspondante afin que celle-ci suive les consignes de rendues.