

Sujet 2:

Piece_siam, debug d'erreurs mémoires..

(durée max: 30min)

Lors du développement de tout projet, il est courant de rencontrer des erreurs mémoires (voir fiche erreurs_memoires.pdf).

Dans la fonction `main`, simulez une erreur mémoire générant une erreur de segmentation (segmentation fault):

Par exemple:

```
void fonction()
{
    int* pointeur;
    pointeur[80208]=12;

    printf("%d\n",pointeur[80208]);
}

int main()
{
    fonction();

    return 0;
}
```

- ➔ Utilisez `gdb` pour retrouver de manière automatique la ligne en cause de l'erreur.
- ➔ Utilisez `valgrind` et observez les messages d'erreurs.
- ➔ Enlevez de votre programme les lignes associées à l'erreur mémoire et relancez `gdb` et `valgrind` afin de vérifier les messages de sorties associées à l'absence d'erreurs.

Générons également une erreur d'utilisation mémoire silencieuse. C'est à dire que le programme a de grandes chances de donner un résultat correct, alors que la mémoire n'est pas gérée correctement:

```
int* fonction()
{
    int a=5;
    int *pointeur=&a;

    return pointeur;
}
```

```
int main()
{
    int *a=fonction();
    int b=7+*a;

    printf("%d\n",b);

    return 0;
}
```

- ➔ Utilisez Valgrind et vérifiez qu'il indique bien la présence d'erreurs mémoires (notez que valgrind peut indiquer le problème au moment de l'utilisation effective de la variable posant problème (ici lors du printf), et non au moment de son initialisation (ici initialisation à b)).

Notez qu'ici, gdb ne détecte pas de problème tant qu'il n'y a pas d'erreurs de type Seg Fault.

Role sur le projet:

Il est désormais de **votre responsabilité** de vérifier que **votre code ne contienne pas d'erreur mémoire**.

Vos rendus seront testés par l'utilisation de Valgrind. Toute indication d'erreur de Valgrind doit être corrigée avant vos rendu. Il vous est conseillé de tester régulièrement votre programme à l'aide de Valgrind tout au long de son développement et de prendre l'habitude de corriger ce type d'erreurs.

Plateau_siam.*(durée max: 1h30min)*

Un plateau est un ensemble de pièces qui sont stockées sous la forme d'un tableau à deux dimensions.

La structure de tableau est elle-même encapsulée dans une struct dénommée `plateau_siam`.

Une pièce du jeu est donc accessible depuis ses coordonnées. Notez que la gestion explicite des coordonnées (x,y) est réalisée dans le fichier `coordonnees_plateau`.

- Ecrivez le code suivant dans votre fonction `main`, et déduisez en le fonctionnement du plateau.

```
int main()
{
    plateau_siam plateau;

    piece_siam p;
    piece_definir(&p, elephant, haut);

    plateau.piece[3][1]=p;

    puts("piece aux coordonnees (x,y)=(3,1) :");
    piece_afficher( &(plateau.piece[3][1]) );

    return 0;
}
```

Remarques sur le choix de la structure du plateau.

Ce choix de stockage sous forme de tableau 2D n'est pas l'unique choix possible.

Il aurait également été possible de stocker uniquement les pièces de type animal et pierre sans stocker explicitement les cases vides. Un tableau unidimensionnel ou une liste chaînée aurait alors été une structure de stockage possible. Ce type de stockage aurait été plus léger en mémoire, mais la recherche d'une pièce à une coordonnée donnée aurait été plus coûteuse.

De manière naturelle, vous auriez également probablement défini le plateau comme étant directement un tableau:

```
piece plateau[5][5];
```

Cette définition est également possible, mais possède certains désavantages:

En effet, à chaque fonction manipulant un plateau, il aurait été nécessaire de passer en argument le type `piece plateau[5][5]`.

Cette syntaxe est plus longue et moins expressive que de passer en argument un type `plateau_siam*`.

(en particulier, le fait que `plateau` soit un pointeur n'est pas clairement indiqué).

Enfin, le fait d'indiquer explicitement la structure de donnée utilisée dans chacune des fonctions empêche toute évolution de celles-ci. Il ne sera ainsi plus possible d'imaginer stocker le plateau sous forme de liste chaînée sans modifier l'ensemble des fonctions manipulant celui-ci.

En cachant le fait qu'un plateau soit stocké sous forme d'un tableau 2D, cela rend possible la modification de la structure avec un impact beaucoup moins important sur les autres fonctions.

Vocabulaire: Le fait de cacher l'organisation interne d'une structure aux autres fonctions s'appelle l'**encapsulation**. Il s'agit d'une notion essentielle de la programmation orientée objet que vous apprendrez par la suite.

L'encapsulation permet de réduire la complexité apparente d'un code, et permet la modification et l'évolution d'un code de manière plus aisée.

➔ Observez la fonction initialisation d'un `plateau_siam`.

Mot clé `const`

➔ Codez le corps de la fonction `plateau_obtenir_piece` et `plateau_obtenir_piece_info` (voir note à la suite).

Note: Ces deux fonctions ont les même arguments à la différence près que l'une travaille sur un plateau modifiable (`plateau_obtenir_piece`), et l'autre avec un plateau non modifiable (`plateau_obtenir_piece_info`).

Dans le premier cas, le pointeur de `piece` renvoyé est un pointeur permettant de modifier la `piece` du plateau, alors que dans le deuxième cas, le pointeur désigne nécessairement une `piece` non modifiable.

Ces deux fonctions ont un code identique, mais possède des applications différentes.

➔ Copiez les lignes suivantes dans la fonction `main`:

```
plateau_siam plateau;  
plateau_initialiser(&plateau);  
  
piece_siam* p1 = plateau_obtenir_piece(&plateau, 2, 3);  
  
plateau_afficher(&plateau); puts("");  
p1->type = rocher;  
plateau_afficher(&plateau); puts("");
```

L'affichage final devrait être celui-ci :

```

[4] *** | *** | *** | *** | *** |
[3] *** | *** | RRR | *** | *** |
[2] *** | RRR | RRR | RRR | *** |
[1] *** | *** | *** | *** | *** |
[0] *** | *** | *** | *** | *** |
    [0]  [1]  [2]  [3]  [4]

```

Qui indique que le plateau à bien été modifié par le biais de p1.

➔ Tentez maintenant de compiler le code suivant.

```

plateau_siam plateau;
plateau_initialiser(&plateau);

const piece_siam* p1=plateau_obtenir_piece_info(&plateau,2,3);

plateau_afficher(&plateau);puts("");
p1->type=rocher;
plateau_afficher(&plateau);puts("");

```

Observez quelle ligne est en erreur. Ici, le fait de travailler sur des pointeurs qualifiés par le mot clé `const` empêche la modification du contenu.

Le rôle du mot clé `const` est un rôle protecteur: Il empêche qu'un programmeur modifie par inadvertance une variable devant garder sa valeur.

Note importante:

- Un pointeur non qualifié par le mot clé `const` peut être converti en un pointeur qualifié de `const` de manière automatique par le compilateur.
- Un pointeur qualifié de `const` ne peut pas être converti en un pointeur non `const`.

Dans le reste de votre code, toute variable de type pointeur ne nécessitant pas d'être modifiée à l'intérieur de la fonction devra **toujours être passé sous sa forme `const`**.

Exemple de codage d'une fonction

(durée max: 30min)

Nous nous intéressons à la fonction `plateau_denommer_type`. Cette fonction permet de compter combien de pièce d'un type donné sont présents sur l'échiquier.

Une première approche directe de programmation de cette fonction pourrait être la suivante:

```
int i=0;
int j=0;
int x=0;
for(i=0 ; i<5 ; ++i)
{
    for(j=0 ; j<5 ; ++j)
    {
        if(plateau->piece[i][j].type==type)
            x++;
    }
}
return x;
```

Ce type de code peut être amélioré en terme de lisibilité et de sécurité.

- Premièrement, il est nécessaire de vérifier le contrat indiquant que `plateau` ne doit pas être un pointeur `NULL`, cela doit être vérifié par l'assertion `assert(plateau!=NULL)` ;
- Deuxièmement, la taille du tableau est indiquée en dur dans les boucles. Il faut généralement éviter d'indiquer directement de tels nombres. Dans notre cas, nous avons défini une constante dans le fichier `coordonnees_plateau.h` du nom de `NBR_CASES`.

Cette variable possède un nom plus explicite qui indique qu'il donne le nombre de cases.

De plus, si le projet doit évoluer afin de prendre en compte un nombre de case différente, il suffira de modifier ce paramètre une seule fois dans tout le programme.

- Troisièmement, la ligne `if(plateau->piece[i][j].type==type)` est trop complexe.
Si cette ligne est assez simple à écrire dans la logique de la recherche du type de la case (i,j), elle rend le code difficile à lire.

Ici beaucoup de notions sont à décrypter pour comprendre cette ligne:

La notion du `if`, le référencement de `plateau`, l'accès aux coordonnées (i,j), l'accès au type, la comparaison des types.

Cela donne 5 niveaux de complexités sur une même ligne.

Dans un code de bonne qualité, on préférera rester en dessous de 2-3 niveaux de complexité par ligne. Chaque ligne doit réaliser une instruction la plus unitaire qu'il soit et ce de manière lisible.

Nous allons donc scinder cette opération en deux étapes:

1. Recherche de la pièce aux coordonnées courantes.
2. Comparaison des types entre le paramètre et la pièce courante

Une possibilité de code plus lisible est la suivante:

```
assert(plateau!=NULL);
// Algorithme:
//
// Initialiser compteur <- 0
// Pour toutes les cases du tableau
//     Si case courante est du type souhaite
//         Incrémenter compteur
// Renvoie compteur

int compteur=0;
int kx=0;

for(kx=0 ; kx<NBR_CASES ; ++kx)
{
    int ky = 0;
    for(ky=0 ; ky<NBR_CASES ; ++ky)
    {
        const piece_siam* piece=
            plateau_obtenir_piece_info(plateau,kx,ky);
        assert(piece!=NULL);

        if(piece->type == type)
            compteur++;
    }
}

return compteur;
```

Notez différents points:

- Chaque manipulation d'un pointeur reçu d'une fonction ou d'un paramètre est vérifié par une assertion certifiant que celui-ci n'est pas NULL.
- La récupération de la pièce courante se fait en une étape séparée du `if`. Celle-ci se fait à l'aide de fonctions que l'on a codé pour éviter toute répétition.

- La condition du `if` est beaucoup plus simple à lire.
- Les variables portent des noms plus explicites (`compteur` à la place de `x`), indication de l'axe (`x,y`) plutôt que (`i,j`).
- Utilisation de `NBR_CASES` plutôt que 5.
- Les variables sont déclarées au dernier moment et pas toutes en début de fonction (un code de bonne qualité moderne limite la portée de ses variables).
- Un algorithme a été ajouté en commentaire en début de fonction.

Dans la suite de votre projet, vous vous efforcerez de rendre votre code **le plus lisible et le plus commenté possible en limitant la complexité de chaque ligne.**

Après l'écriture de chaque fonction, il vous est conseillé de la **relire** attentivement et de tenter de la **simplifier** tant que possible (plusieurs passes sont souvent nécessaires).

Fin du fichier plateau

(durée max jusqu'à la fin du TP: 1h30min)

- ➔ Implémentez le corps des fonctions suivantes: `plateau_etre_integre`, et `plateau_exister_piece`.

Complétez également les fonctions de `coordonnees_plateau`.

- ➔ Implémentez le corps de la fonction `coordonnees_etre_bordure_plateau`.
- ➔ Ecrivez la documentation de la fonction `coordonnees_appliquer_deplacement` (en tête).

Jeu_siam

Un `jeu_siam` contient l'ensemble des éléments définissant l'état d'un jeu: un plateau et le joueur courant.

Notez que le jeu est une juste une structure conteneur et on ne gère pas les règles du jeu dans ce fichier.

- ➔ Implémentez le corps de la fonction `jeu_verifier_type_piece_a_modifier` et `jeu_obtenir_type_animal_courant`.

Aide : Pour rappel, n'oubliez pas d'utiliser les fonctions préalablement écrites, on pourra par exemple réutiliser les fonctions de vérification d'intégrité d'un jeu, et des pièces. On pourra utiliser les fonctions d'analyse de coordonnées pour vérifier que celle-ci soient bien dans le plateau. Interdisez-vous d'utiliser tout accès direct au tableau 2D de la structure d'un `jeu_siam`, mais utilisez à la place les fonctions `plateau_obtenir_piece(/_info)`. Pensez également qu'il existe des fonctions

pouvant vous aider dans les fichiers `joueur.h` et `joueur.c`, en particulier `joueur_etre_type_animal` et `joueur_obtenir_animal` permettent de vérifier l'adéquation entre le numéro du joueur courant (0 ou 1), et l'animal courant (elephant ou rhinoceros).

Mode_interactif

- ➔ Ecrivez dans votre fonction `main` l'unique appel à `mode_interactif_lancer()` ; (similaire à l'état du fichier `main` lors du téléchargement du projet).

Observez que cet appel de fonction vient lancer le mode interactif du jeu. (Pour l'instant, aucun mouvement n'est possible, vous implémenterez dans les séances futures les différents coups possibles).

API

Les fichiers `api.h` et `api.c` contiennent les fonctions d'appels généraux permettant de jouer au jeu du siam (**API=Application Programming Interface**).
Il s'agit de fonctions de hauts niveaux par rapport au jeu.

Ces fonctions modélisent le jeu comme étant une boîte noire dont les accès sont donnés par les fonctions de l'API. Cela permet à d'autres programmeurs d'utiliser le jeu comme une bibliothèque dans leur code à l'aide d'un certain nombre de fonctions.

Nous proposons ici trois fonctions dont le rôle est d'effectuer une action sur le jeu.

Ces trois fonctions sont:

- **L'introduction** d'une nouvelle pièce dans le jeu.
- Le **déplacement** d'une pièce existante.
- Le **changement d'orientation** d'une pièce déjà existante.

Les fonctions de l'API proposent une interface permettant de tenter la réalisation de ces actions et la modification du jeu si celle-ci sont valides.

- ➔ Observez quels sont les paramètres d'entrées de ces fonctions et quel rôle joue le paramètre de sortie.

L'implémentation des fonctions de l'API va consister à vérifier que les paramètres d'entrées sont corrects et réaliser l'appel aux fonctions de plus bas niveau du jeu permettant de réaliser ces actions.

Les fonctions de l'API vont en particulier appeler les fonctions de `plateau_modification` qui gèrent

la vérification des règles du jeu.

Plateau modification.

Plateau modification contient l'implémentation des règles de mouvements des pièces. Ce sont ces fichiers qui vérifient réellement les règles du jeu.

Pour les 3 types d'actions possibles: introduction d'une nouvelle pièce sur le plateau, déplacement d'une pièce existante, et changement d'orientation d'une pièce sans déplacement, nous définissons une approche en deux étapes:

- Premièrement, la vérification que l'action est valide vis-à-vis des règles du jeu (fonctions terminant par `etre_possible`).
- Deuxièmement, la mise en place effectivement du déplacement sur le plateau de jeu en supposant que l'action est valide.

Notez qu'ici, seul le plateau est pris en compte, et non le joueur courant. On supposera donc qu'à ce niveau, tout animal désigné correspond au joueur courant. Ce sera aux fonctions de plus haut niveau (tel que l'API) de vérifier que l'animal désigné peut être joué par le joueur courant.

- ➔ A partir du contrat donné pour la fonction `plateau_modification_changer_orientation_piece_etre_possible`, implémentez le corps de cette fonction (soyez sûr d'avoir compris les règles du jeu, demandez à un enseignant en cas de doute).

Pour rappel: Tout prérequis défini dans le contrat de la fonction doit être soit vérifié directement par le compilateur soit par l'utilisation de fonctions `assert`. Par contre, si une condition n'est pas requise dans le contrat, alors elle ne devra pas faire planter le programme si celle-ci n'est pas respectée.

- ➔ A partir du contrat donné pour la fonction `plateau_modification_changer_orientation_piece`, implémentez le corps de cette fonction.

Travail en autonomie

(durée estimée: 3h):

- ➔ Finir le sujet.
- ➔ Assurez-vous d'avoir testé les fonctions que vous avez écrites.
- ➔ Réfléchir aux algorithmes de déplacements et d'introduction de pièces.