Sujet 1:

Découverte et mise en place des fichiers.

(durée max: 15min)

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

- ➔ **Remplissez** le fichier `nom.txt` avec vos noms, prénoms, et groupe. Suivez les consignes du fichier: `annexe_completion_fichier_nom`.

Les fichiers de votre archive deviennent donc vos propres fichiers à titre nominatif.

Notez qu'une complétion de ce fichier ne suivant pas les consignes vous fera perdre des points de respect des consignes.

Compilation

Pour l'instant, aucun Makefile n'est fourni (vous l'écrirez dans les séances futures).
Un fichier de script Shell est fourni et réalise la compilation de l'ensemble des fichiers présents.

Notion avancée: Notez que part rapport au Makefile que vous créerez plus tard ce fichier est long à écrire, difficile à lire et à manipuler, et impose une compilation de l'ensemble des fichiers dans un ordre spécifique.

Observez le contenu du fichier de compilation `compilation.sh` et l'ouvrant à l'aider d'un editeur de texte (Gedit, Kate, Vi, XEmacs, etc).

Rappel de la formation Linux:
Un script Shell est un fichier contenant du texte débutant par `#!/bin/bash`.
(`#!` indique qu'il s'agit d'un script, et `/bin/bash` est le chemin vers le programme shell utilisé)
Ce fichier contient ensuite des instructions de la ligne de commande qui seront exécutés les unes derrière les autres lors de l'appel de ce script.

Pour exécuter un script Shell depuis la ligne de commande, il faut:
- Lancez une **ligne de commande** et placez vous **dans le répertoire contenant le script**.
- Donnez potentiellement les **droits d'exécutions** au fichier en question:

```
$ chmod +x compilation.sh
```

Note : Le symbole $ *est utilisé pour indiquer une instruction à lancer en ligne de commande. Ce n'est pas un caractère à taper.*

Cette opération pour les droits d'exécutions n'est à faire qu'une seule fois dans la vie du fichier. En effet, par défaut les fichiers Linux peuvent être lues et écrit mais pas exécutés par soucis de sécurité.

Il faut donc leur donner ce droit de manière explicite une fois.

- **Lancez** le script:
`$ ./compilation.sh`

➔ **Compilez** le programme et assurez vous d'avoir un exécutable créé dans le répertoire *bin/*.
➔ **Ouvrez** un terminal dans le répertoire *bin/* ainsi qu'un autre terminal dans le répertoire *src/*.

Ainsi pour compiler, vous pourrez écrire dans un terminal (répetoire `src/`):
`$ ./compilation.sh`

Et exécuter directement à partir de l'autre terminal (répertoire `bin/`)
`$ ./jeu_siam`

Note: Pour l'instant, un mode interactif se lance, mais le jeu n'est pas jouable.

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

(durée max: 3h)

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.*

L'étape d'analyse de la structure du code à déjà été réalisé en grande partie pour ce projet. Une analyse type "top-down" qui s'intéresse à définir d'abord les concepts de hauts niveaux pour arriver aux concepts de niveaux plus bas indique que le jeu de siam peut être vue de la manière suivante:

- Un **jeu complet** contient un plateau de pièce et un joueur courant qui doit jouer un tour (abstraction d'un jeu, haut niveau d'abstraction).
    - Un **plateau** du jeu est un ensemble de cases ou de pièce jouable (abstraction d'un plateau).
        - Chaque **pièce** est formée d'un type et possède pour certaines d'entre elles une orientation (abstraction d'une piece).
            - Un **type** peut être un animal (éléphant ou rhinocéros), un rocher, ou une case vide (abstraction d'un type, bas niveau d'abstraction)
            - Une **orientation** peut correspondre à un déplacement (haut,bas,gauche,droite), ou ne correspondre à rien pour certaines pièces tel que le rocher (aucune_orientation). (abstraction d'une orientation, bas niveau d'abstraction).

Notez que les niveaux d'abstractions les plus élevés correspondent à des concepts qui permettent de comprendre facilement le jeu, mais qui semble complexes ou floues en terme de code. Les niveaux d'abstractions les plus bas ne permettent pas d'entrevoir le jeu dans sa globalité, mais correspondent à un code simple, ou l'ensemble des configurations sont généralement énumérables de manière exhaustives.

La réflexion sur la structure du jeu peut ainsi se réaliser suivant une approche top-down. Le code quant à lui se réalise plus souvent de manière bottom-up. C'est à dire que l'on code d'abord les niveaux d'abstractions les plus bas. Puis, une fois que ceux-ci sont validés par des tests, les niveaux plus élevés sont ajoutés par couches successives.

Nous débutons dans ce projet par les niveaux bas de type de pieces et d'orientation, puis viendront par la suite intégrer la notion de piece, de plateau et de jeu afin d'obtenir une première version du jeu jouable.

Type de piece:

Le fichier `type_piece.h` contient la définition d'une énumération nommée type_piece décrivant l'ensemble des pièces rencontrées sur un plateau de manière exhaustive.
Ce fichier définit également des en-tête (ou signature) de fonctions associées à ce type de piece.

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

Le fichier `type_piece.c` contient quant à lui le corps des fonctions (ou implantation/implémentation réalisant les fonctions).

➔ Dans le fichier `main.c`, supprimer le code existant et tapez à la place:

```c
#include "type_piece.h"
#include <stdio.h>

int main()
{
    type_piece type=elephant;

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

    int est_ce_un_elephant = (type==elephant);
    int est_ce_un_rocher   = (type==rocher);
    printf("%d %d\n",est_ce_un_elephant,est_ce_un_rocher);

    return 0;
}
```

➔ Expliquez l'affichage obtenu à l'execution.

Vous devez être capable d'afficher des variables de type:
int (%d) ; float/double (%f) ; caractère (%c) ; 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?

La fonction `type_nommer` permet de réaliser la correspondance entre un type et une chaine de caractère plus lisible par un humain (et non un entier).

➔ Tapez et commentez le résultat de l'appel suivant (vous pouvez supprimer ou stocker ailleurs vos tests précédents):

```c
#include "type_piece.h"

#include <stdio.h>

int main()
{

    type_piece type=rhinoceros;

    const char* nom_du_type=type_nommer(type);
    printf("Le type est un %s. \n",nom_du_type);

    printf("Le type est un %s. \n",type_nommer(rocher));

    return 0;

}
```

Fonctions de type_piece

Observons les autres fonctions de type_piece.

`int type_etre_integre(type_piece type);`
permet de vérifier qu'un argument `type_piece` définisse bien un type viable d'après sa définition. Ici on entend par "integrité" le fait que sa valeur soit une valeur possible de type_piece.

Remarque importante: Tous les types (enumerations, structures, etc) définit par l'utilisateur

doivent avoir un moyen de vérifier qu'ils sont valides.
Certaines vérifications sont automatiquement faites par le compilateur sur les types (ex. non signés, flottants, entiers, etc). Et d'autres doivent être vérifiées explicitement par l'utilisateur (ex. positivité, plage de valeur, etc).

Observation de la structure des commentaires.

Dans notre cas, un type de piece doit être un éléphant (0), un rhinocéros (1), un rocher (2) ou une case vide (3).
Etant donné qu'une énumération est de type entier, il est possible de définir un type de pièce dont la valeur vaut 12 et ne correspondrait à aucun type viable. Cette valeur de type est qualifié de non intègre.
La fonction `type_etre_integre` a pour vocation de vérifier que le type en argument est bien viable de manière simple et explicite dans le programme.

- ➔ Observez les commentaires de `type_etre_integre` dans `type_piece.h`. Observez que ces commentaires sont structurés en 3 parties: Une partie de description du **rôle de la fonction**, une partie de **condition(s) nécessaire(s)** pour que la fonction s'exécute, et une partie de **garantie(s)** sur le déroulement de la fonction. Cette organisation devra toujours être respectée dans chacune des fonctions que vous coderez et son application permet la mise en place de la méthode de programmation dite **par contrat**.

- ➔ Observez l'implémentation de `type_etre_integre` dans `type_piece.c`. Observez que cette fois, les commentaires **décrivent le choix d'implémentation** réalisé. Il s'agit du type de commentaire attendu dans l'implémentation des fonctions. On pourra également rencontrer à d'autres endroit des commentaires de **type algorithme**.

Vous devez être capable de comprendre le lien entre les commentaires et le corps de la fonction. Si vous avez un doute sur la compréhension appelez un enseignant.

Première fonction.

La signature de la fonction `type_etre_animal` est donnée dans le fichier .h.

- ➔ Complétez le corps de la fonction (dans le .c) réalisant ce que la documentation indique.

- ➔ Observez le corps de la fonction `type_nommer_non_cours`. Observez qu'un éléphant est associé au caractère 'e', un rhinocéros au caractère 'r', un rocher aux caractères "RRR", et une case vide au caractères "***".

Analyse du codage d'une fonction nécessitant un contrat.

Nous détaillons dans la suite le cheminement aboutissant à la construction du corps de la fonction `type_correspondre_caractere_animal` de manière attendue.

➔ Observez l'en-tête de cette fonction. Assurez-vous que vous compreniez bien son rôle, appelez un enseignant au besoin.

Une première implémentation possible de cette fonction pourrait être l'implémentation triviale suivante:

```
type_piece type_correspondre_caractere_animal(char type)
{
    type_piece type_enum;

    if(type=='e')
        type_enum=elephant;
    else if(type=='r')
        type_enum=rhinoceros;

    return type_enum;
}
```

Cette implémentation **ne satisfait pas** les contraintes fixées dans la documentation.
Premièrement, il n'y a pas de contrôle sur l'intégrité du paramètre d'entrée.
Ici, la fonction nécessite un caractère désignant un animal, c'est à dire soit 'e', soit 'r' et ne devrait pas accepter d'autres caractères.

Si la fonction reçoit un autre caractère, alors son comportement devient indéterminé.
Supposons par exemple que cette fonction reçoive 'u'. Dans ce cas, aucune condition "if" n'est vérifiée, et le type de retour est celui-donné par défaut à un `type_piece`.
La valeur est donc généralement 0 sur les PC standards, mais peut valoir n'importe quel nombre positif sur d'autres type d'ordinateurs ou systèmes d'exploitations.

➔ Mettez en place cette fonction et cet appel erroné, et vérifiez la valeur obtenue dans votre cas.

Cette implémentation, bien que réalisant correctement la fonction dans le cas attendue ne peut pas être considérée comme une fonction valide.
En effet, une erreur de programmation qui viendrait lui fournir en paramètre d'entrée une valeur inattendue aboutit à un comportement indéterminé.

Contrairement à ce que l'on pourrait penser intuitivement, ce type d'erreur est grave. En effet, ce comportement ne sera pas forcément détecté comme un bug lors du développement: le programme ne plante pas et donne potentiellement une sortie raisonnable.
Ce type de problèmes peut être parfois détecté bien plus tard, une fois le logiciel finalisé et vendu, engendrant des corrections excessivement difficiles (problème difficile à identifier) et coûteuses sur de grands logiciels.

- Quelle solution apporter ?

- Premièrement, il est inacceptable de définir une variable sans lui donner de valeur par défaut. En effet, cela peut aboutir à des comportements indéterminés (la norme ne précise pas quelle valeur doit avoir une valeur entière dont la valeur n'est pas donnée explicitement).

Une version améliorée du programme pourrait être celle-ci:

```
type_piece type_enum=elephant;

if(type=='e')
    type_enum=elephant;
else if(type=='r')
    type_enum=rhinoceros;

return type_enum;
```

Note importante de bonne pratique: Toute variable de votre programme devra **toujours** avoir une valeur par défaut au moment de son initialisation. Toute variable non initialisée directement sera considérée comme incorrecte.

- Deuxièmement le problème n'est pas entièrement traitée. En effet, le caractère 'u' engendre cette fois une sortie valant l'enum `elephant` de manière certaine, mais cela ne correspond toujours pas au contrat de la fonction.
Une solution à ce problème serait le code suivant:

```
type_piece type_enum=elephant;
if(type=='e')
    type_enum=elephant;
else if(type=='r')
    type_enum=rhinoceros;
else
{
    puts("Erreur: caractere non reconnue: ni 'e', ni 'r'");
}

return type_enum;
```

Note: La fonction `puts()` permet d'afficher une ligne à l'écran puis de sauter à la ligne. Lorsqu'il n'y a pas d'affichage formaté de variables, on préférera l'utilisation de `puts` que de `printf`.
En effet, en utilisant `puts` on indique au lecteur du code que l'affichage ne contient pas de variable ce qui rend le code plus expressif à lire.

Cette fois, toute entrée autre que 'e' ou 'r' aboutit à l'affichage d'un message d'erreur.

- Peut-on encore améliorer la fonction ?

Il faut désormais se poser la question de l'utilisateur de la fonction. Il y a généralement 3 types d'utilisateurs:
- Un *utilisateur extérieur* (typiquement le client) qui va rentrer un caractère depuis la ligne de commande et dont le paramètre est directement envoyé à cette fonction ?
- Un *programmeur externe* qui va utiliser cette fonction comme une librairie externe? (exemple, lorsque vous utilisez la fonction printf, strlen, malloc, etc)
- Ou le *programmeur même* de la librairie qui n'appellera cette fonction qu'en interne à partir d'autres fonctions ?

Généralement, la majorité des fonctions écrites entrent dans le dernier cas.

Notes avancées:
Les fonctions faisant le lien entre un utilisateur et un code nécessitent des entrées dites protégées qui demande d'entrer une valeur correcte tant que celle-ci ne l'est pas. Ces fonctions sont des fonctions dites d'interfaces, ou d'IHM (Interface Homme Machine).
Les fonctions permettant d'utiliser un code comme une librairie externe pour laquelle nous ne sommes pas le programmeur sont des fonctions dites d'API (Application Programming Interface). Elles doivent être robustes aux entrées non souhaitées et indiquer clairement les entrées attendues. Nous réaliserons des fonction d'API dans les séances futures.

Dans ce dernier cas, les fonctions n'ont vocations qu'à être utilisées en interne par le développeur de l'application. Une erreur de paramètre envoyé à cette fonction est forcément une erreur de programmation. Une erreur de programmation doit se détecter le plus tôt possible lors du développement et être corrigée par une modification du code.

Si le programme contient une **erreur de programmation, le mieux à faire est de quitter le programme et d'indiquer clairement où se trouve l'erreur pour faciliter son debug (/sa correction).**

Justification:
En indiquant uniquement une erreur par une phrase du type "*Erreur: caractere non reconnue: ni 'e', ni 'r'*", il est possible que le message passe inapercu (au milieu d'autres sorties d'écrans), ou bien *noie* l'utilisateur sous de trop nombreux messages si la fonction est appelée plusieurs milliers de fois.
De plus, la fonction n'indique pas la localisation du code posant problème. Sur un code de plusieurs milliers de lignes, un message aussi peu précis engendrerait une localisation longue et difficile.

Nous pouvons désormais proposer la solution suivante:

```
if(type!='e' && type!='r')
{
   puts("Erreur fonction type_correspondre_caractere_animal");
   puts("type!='e' et type!='r'");
   puts("Erreur ligne 121.");
   abort();
}

type_piece type_enum=elephant;

if(type=='e')
    type_enum=elephant;
else
    type_enum=rhinoceros;

return type_enum;
```

Cette fois, la fonction répond aux différentes contraintes:
- L'erreur est détectée le plus tôt possible (avant d'exécuter le code).
- Le message d'erreur est précis pour un programmeur et permettra la localisation du problème.
- Le programme quitte (la fonction `abort()` termine l'exécution du programme immédiatement).

Par contre, elle possède d'autres inconvénients:
- Le code est beaucoup plus long (gestion d'erreur plus grande que le code utile de la fonction).
- La maintenance du message d'erreur est irréalisable pour un grand code (numéro de ligne à mettre à jour à chaque modification du fichier lors du développement, nom de la fonction qui peut être changée, etc).

Il existe une solution alternative permettant d'obtenir un message d'erreur précis dans un tel cas et mis à jour automatiquement: la fonction **`assert`**.

Considérez désormais le code suivant:

```
assert(type=='e' || type=='r');

type_piece type_enum=elephant;

if(type=='e')
    type_enum=elephant;
else
    type_enum=rhinoceros;
```

```
return type_enum;
```

➔ Observez comment réagit ce code en cas d'envoi de paramètre intègre, et non intègre.

Explication du rôle de `assert`.

La fonction `assert` permet de certifier ('to assert' = certifier) au programmeur que l'expression donnée en paramètre est vraie.
C'est à dire que si type vaut 'e', ou 'r', alors la fonction ne fait rien et laisse le programme continuer.
Si cela n'est pas vérifié, le programme quitte en indiquant le fichier, la fonction, la ligne d'erreur ainsi que l'expression qui n'est pas valide.
Ce message s'adresse donc au programmeur qui peut ensuite intervenir sur le code pour corriger celui-ci.

La fonction `assert` est une fonction standard permettant de valider qu'une expression est vraie lors de la phase de développement.
La fonction `assert` doit être utilisée tant que possible pour vérifier au maximum les affirmations que l'on doit vérifier sur un code. Ce type de programmation fait partie de la **programmation dite par contrat**.

C'est à dire que le programmeur passe un contrat pour le fonctionnement de sa fonction:
Le programmeur garantie qu'il ne passe que certains arguments à la fonction, et en contrepartie, la fonction garantie qu'elle donne un résultat attendue.
Tout manquement à ce contrat aboutit à l'arrêt du programme et l'indication précise du problème.

Dans le reste de votre code, tout prérequis et toute garantie (non vérifiée par le compilateur) devra être vérifiée explicitement (généralement par des `assert` si le programme doit quitter immédiatement).

Une fois la phase de développement terminée, il est possible de supprimer l'évaluation des assertions (options de compilation `-DNDEBUG`).
C'est à dire qu'une fois le logiciel finalisé, les assertions ne sont plus vérifiées explicitement par le programme. Cela implique deux choses:
- Les `assert` ne doivent être utilisés que pour détecter des erreurs de développements de code, et non pour tester des cas d'erreur pouvant arriver suivant les choix de l'utilisateur.
- Les `assert` ne ralentissent aucunement le temps d'exécution du logiciel final puisqu'ils peuvent être éliminés au final. Il n'y a donc pas besoin de se restreindre sur l'utilisation de vérifications d'assertions pendant l'étape de développement.

Finalement, la fonction peut encore être améliorée. Le code contenu dans la fonction `assert` :
`assert(type=='e' || type=='r');`
à déjà été écrit. En effet, ce code consiste à vérifier que le caractère correspond à un animal. C'est justement le rôle de la fonction
`int type_caractere_animal_etre_integre(char type)`

Il est donc possible de réutiliser la fonction déjà écrite. La fonction finale devient donc:

```
type_piece type_correspondre_caractere_animal(char type)
{
    assert(type_caractere_animal_etre_integre(type));

    type_piece type_enum=elephant;

    if(type=='e')
        type_enum=elephant;
    else
        type_enum=rhinoceros;

    return type_enum;
}
```

Note sur la répétition de code et programmation DRY:

En programmation, il faut toujours éviter d'avoir à plusieurs endroits le même code qui réalise la même action (typiquement du copié-collé à différent endroits).
En effet, cela génère un code plus long qu'en le factorisant par une fonction, et généralement moins lisible (ici par exemple la fonction `type_caractere_animal_etre_integre` est plus explicite car elle décrit dans son titre son action).
De plus, lors de l'évolution du programme tel que l'ajout d'un animal, ou la modification d'un caractère, il suffirait de modifier le comportement à un seul endroit du programme dans la fonction de vérification et non à chaque fonctions répétant le code correspondant.
Ce principe de non répétition est désigné par la méthode de programmation **DRY** (Don't Repeat Yourself).

Résumé sur les asserts:
- La fonction `assert` permet de vérifier qu'une **expression doit être vraie** lors de l'exécution du programme.
- Les assertions permet de certifier les **préconditions** (conditions nécessaire au bon fonctionnement de la fonction) et **postconditions** (garanties que la fonction peut donner).
- Lorsqu'une assertion n'est pas vérifiée, **le programme s'arrête forcément**. Cela doit nécessairement signifier qu'il y a un bug dans le code et que celui-ci doit être modifié.
- Un `assert` ne doit **jamais influencer** le fonctionnement du programme (ex. incrémentation d'une variable, etc). En effet, ils doivent pouvoir être supprimés automatiquement du programme final sans que cela change le fonctionnement de celui-ci.
- Attention: Une assertion ne doit pas être confondue avec une conditions de type "if" permettant de traiter les cas d'erreurs. Une assertion non vérifiée fait **forcément quitter le programme**.
- Attention: Une assertion doit être limité à vérifier des **erreurs de programmation**: c'est à dire que des conditions qui sont vérifiées doivent toujours être vraie. Peut importe l'utilisation du programme. Si la condition n'est pas vraie, alors le programme doit nécessairement avoir un bug. Une condition qui pourrait, dans certains cas, ne pas être vraie lors de l'exécution d'un programme (ex. Chargement d'un fichier qui n'existe pas, etc) n'est pas une erreur de programmation et ne doit pas être vérifié par un `assert`.

Orientation deplacement.

Le fichier `orientation_deplacement.h` définit une énumération désignant l'orientation possible d'une case du plateau. Si il s'agit d'un animal, alors celui-ci peut avoir l'une des 4 orientations (gauche,droite,haut,bas), et si il s'agit d'un rocher ou d'une case vide, alors on lui associe `aucune_orientation`.

- ➔ Complétez la documentation de l'en tête de `orientation_etre_integre` (en vous inspirant de `type_piece`). Notez qu'il est nécessaire d'avoir les 3 parties: explication du rôle de la fonction, type de paramètre et plage de valeurs nécessaires (preconditions), et garantie de traitement (postconditions).

- ➔ Complétez le corps de la fonction `orientation_etre_integre_deplacement`.
- ➔ Complétez la documentation de l'en tête de `orientation_caractere_etre_integre`.
- ➔ Complétez le corps de la fonction `orientation_correspondre_caractere` en suivant l'approche vue précédemment pour `type_piece`. (Notez qu'on pourra se servir de la condition switch pour éviter une suite de nombreux `if, else if`).

Tests.

(durée max: 45min)

La pratique de tests des fonctions écrites fait également partie des bonnes pratique de programmation.

Vos tests doivent:

- Être précis et tester des fonctionnalités de manière **unitaire** (c'est à dire une seule chose à la fois).
- Assurer que vos fonctions réalise ce qu'il faut dans les bonnes conditions d'utilisations, mais également que les **cas d'erreurs** soient traités de manière attendue.
  Une fonction qui donne le résultat attendu uniquement dans les bonnes conditions d'utilisation et permet tout de même des comportements indéterminés lors de mauvais cas d'utilisations est considéré comme invalide.
- Être le plus exhaustif possible. C'est à dire tester un maximum de cas intéressants (dans les bonnes et mauvaises conditions d'utilisations).
- Avoir une sortie simple à analyser (ne pas afficher un résultat, mais afficher si le résultat est correct ou non).
- Pouvoir être lancés et relancés aisément sans efforts (ne pas avoir à réécrire les tests à chaque fois qu'on souhaite les lancer).

Une bonne pratique de test consiste à écrire pour chaque fonction, une autre fonction de test lui correspondant.

Considérons l'exemple de la fonction `type_etre_animal`.
Cette fonction doit renvoyer 1 si la valeur d'entrée vaut `elephant` ou `rhinoceros`, et 0 pour tous les autres cas.

Une première manière de tester serait la suivante:

```c
int main()
{
    printf("elephant: %d\n",type_etre_animal(elephant));
    printf("rhinoceros: %d\n", type_etre_animal(rhinoceros));

    return 0;
```

```
}
```

Ce programme devrait afficher:
```
elephant: 1
rhinoceros: 1
```
si les fonctions sont correctement implémentées.

Ce programme de test ne vérifie cependant pas plusieurs points des bonnes pratiques:

- Le programme n'indique pas si le test est concluant ou non. C'est au programmeur d'interpréter après exécution si le chiffre 1 est le chiffre attendu ou non. Cela engendre des tests difficiles à interpréter.

  Il serait préférable d'avoir une sortie indiquant si le test est **OK**, ou **KO**.

- Le programme ne peut pas se relancer facilement. Lors du développement la fonction `main` va évoluer et ces deux lignes vont probablement disparaître. Ce test ne sera ainsi réalisé qu'une fois. Si un autre programmeur vient modifier la fonction `type_etre_animal`, tous le travail de l'écriture des tests devra être refait.

  Il serait préférable de concentrer les tests dans une fonctions simple à appeler et ce de manière pérenne.

- Le test ne vérifie que les cas où la fonction doit renvoyer 1. Les cas où le type n'est pas un animal n'est pas testé. La fonction peut être invalide sans qu'aucun test ne le montre. (Par exemple, la fonction "return 1;" satisfait au test).

Proposons une seconde version plus évoluée.
Ecrivons la fonction suivante dans le fichier `type_piece`:

```
void test_type_etre_animal()
{
    puts("Test type_etre_animal");

    if(type_etre_animal(elephant)!=1)
        puts("elephant KO");

    if(type_etre_animal(rhinoceros)!=1)
        puts("rhinoceros KO");

    if(type_etre_animal(rocher)!=0)
        puts("rocher KO");
    if(type_etre_animal(case_vide)!=0)
        puts("case vide KO");
```

```
    int k=0;
    for(k=4;k<20;++k)
        if(type_etre_animal(k)!=0)
            printf("valeur %d KO\n",k);

}
```

Et appelons cette fonction de test depuis la fonction `main`.

- Cette fois, la fonction test à la fois des cas valide, mais également des cas non valides et non attendus par l'utilisation standard de la fonction (valeur de 4 à 100 par exemple).
  L'ensemble des valeurs entières possibles ne sont pas testés, mais l'échantillonnage est suffisamment avancé pour considérer que la fonction passant ce test est valide.

- L'affichage en ligne de commande est plus clair. Ici la fonction affiche un problème si celui-ci survient (tentez de modifier la fonction `type_etre_animal` pour vérifier la sortie obtenue).
  Cette fois, il n'y a pas d'analyse humaine à faire à chaque lancement du test, mais simplement à lire si la sortie est KO ou non.

Notez qu'il est évidemment possible de réaliser des variantes d'affichages telle qu'une fonction affichant également les tests passés de manière correcte avec l'affichage d'un 'OK'.

Cette fonction nécessite d'être appelée dans le programme `main`. On peut supposer que si plusieurs fonctions de tests existent, il n'est pas souhaitable d'écrire leurs appels à chaque fois.
Une solution permettant de factoriser toutes les fonctions de test est la suivante:

Dans le main, on écrit désormais une fonction unique `test_lancer()`, qui aura pour rôle d'appeler toutes les autres fonctions de tests écrites.

```
void test_lancer()
{
    test_type_etre_animal();
    // puis tous les autres tests a ajouter
    //  de maniere iterative ...
}

int main()
{
    test_lancer()

    return 0;
}
```

Seul ce `test_lancer` sera écrit directement depuis la fonction `main`. Il sera ainsi possible d’appeler potentiellement des milliers de tests en une seule commande, et de vérifier que le code est

correct pendant le développement.

Note: Il est très important d'écrire vos fonctions de tests **au fur et à mesure** de votre développement. N'écrivez pas l'ensemble des tests uniquement à la fin.
Dès que vous codez une fonction, ajoutez un test à votre base de tests.
De cette manière, il est possible à tout moment de vérifier que vos fonctions sont valides. Dès que vous modifierez une fonction, l'ensemble des tests pourra à nouveau être rejoué et vérifier que vos fonctions sont toujours correctes.

Vocabulaire:
- La réalisation de fonctions de tests vérifiant le bon fonctionnement de l'ensemble des fonctionnalités de manière précise s'appelle la méthodologie de **tests unitaires**.
- La possibilité de vérifier l'ensemble de vos fonctions au fur et à mesure du développement (et non uniquement à la fin) entre dans les méthodologies **d'intégration continue** dans le cadre du développement dit **agile**: à tout moment du développement, vous avez un programme fonctionnel et dont le fonctionnement est validé par des tests.

Synthèse du développement:

Dans le reste de l'ensemble des fonctions que vous coderez, vous veillerez à:

- Avoir une documentation précise décrivant une **méthodologie par contrats** (dans les fichiers .h).
- Certification de la validité de vos contrats par des **assertions**.
- Lorsque cela est utile, décrire dans la documentation l'**algorithme** suivit (dans les fichiers .c).
- Les **tests** de vos fonctions à l'aide d'appels simples qui restent pérenne tout au long du développement.

Vérification des consignes de rendus.

(durée max: 15min)

- ➔ 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 "Utilisation du script de verification"). Assurez vous que votre archive respecte bien ce script.

Note. Il vous est fortement conseillé d'utiliser ce script avant chaque rendu depuis un PC de CPE. Il n'assure pas que l'ensemble des consignes sont respectés, mais permet d'éviter des oublis courants rendant votre travail non analysable.
Ce script ne se substitue pas non plus à votre vérification manuelle. Seule la vérification manuelle précise sur un PC de CPE vous permet de vous assurer que vous avez respecté les consignes.

Travail en autonomie.

(durée estimée: 2h)

- ➔ Réalisez une fonction de test pour l'ensemble des fonctions complétés dans cette séance.
- ➔ Déposez votre projet sur le e-campus avant la fermeture du dépôt de votre groupe en suivant les consignes de rendus (Aidez vous du script de vérification des consignes).
- ➔ Lisez l'énoncé numéro 2 qui demandera un travail long et conséquent afin de ne pas perdre de temps en séance.