EFCS – PARTIE PRATIQUE
420-W31-SF ALGORITHMIQUE AVANCEE
MASTERMIND
PONDERATION 15%

Mode de réalisation : **Travail en équipe de 2**

Ce travail comporte 3 parties et vise à utiliser la liste chaînée dans un contexte de résolution de problème.

MISE EN CONTEXTE

Contrairement à la pile et la file, la liste chaînée nécessite souvent de se déplacer de nœud en nœud. Ceci est évidemment fait dans les méthodes de la liste chaînée, mais la définition actuelle de la liste permettrait à l'utilisateur (le programme) de le faire aussi (getFirstNode, getNext, etc.)

Le but de ce travail est de cacher l'implémentation du nœud à l'utilisateur grâce à un itérateur et d'utiliser la liste et son itérateur dans le cadre du jeu de société **Mastermind**.

TRAVAIL A EFFECTUER

Votre code de départ contiendra les structures de données dans une répertoire Structs/Include (le code est vide mais il compile) et le projet du jeu Mastermind.

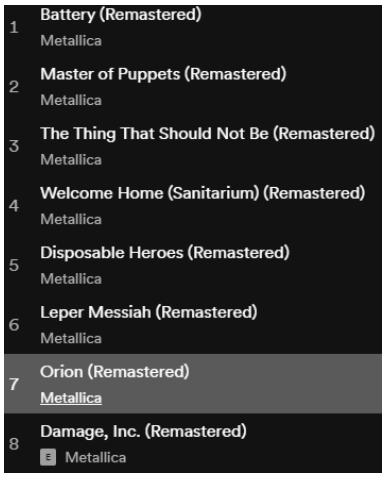
1) Remplacez les .h dans Structs/Include par vos .h de structures de données.
2) Ouvrez le projet du jeu Mastermind. Il est configuré pour voir les structures de données.
3) Compilez le code pour tester l'intégration avec vos structures de données.
4) Les tests unitaires donnés ne doivent **en aucun temps être modifiés pour favoriser la compilation**.

Partie 1 – Légère modification de la DataStructure

Renommez la méthode **getNumNodes()** de la classe **DataStructure** par **getLength()**. Ajustez votre code pour le faire compiler.

Partie 2 – Création d'un itérateur

Contrairement à la pile et la file qui sont des structures mécaniques d'ajout/retrait, la liste doit souvent être parcourue par l'utilisateur lors de son utilisation. Par exemple, si on utilise une liste dans un contexte de « playlist » de musique avec les chansons en ordre numérique. Avec une liste unidirectionnelle, on pourra descendre dans la liste pour jouer des chansons à la pièce :



Ce qui pourrait correspondre à du code (très simplifié) de ce genre dans la liste :

```
/*.h*/
Node * currentNode = listDouble.getFirstNode();

/*.cpp*/
public void onKeyDown()
{
    currentNode = currentNode->getNext();
}
public void play()
{
    player.play(currentNode->getSong()->getTitle());
}
```

C'est un peu une faiblesse à cause de l'exposition des nœuds de la liste. Afin de pallier à une telle exposition, nous allons implanter un itérateur sur la structure.

Qu'est-ce qu'un itérateur ?

Un itérateur peut être vu comme un petit « curseur » qui pointe vers un nœud de la structure, il travaille de concert avec cette dernière. C'est lui qui va travailler avec le nœud courant à l'interne (en composition faible, losange blanc). Il nous abstrait son utilisation en nous exposant des opérateurs pour :

1) Se déplacer d'un nœud vers la droite (++)
2) Obtenir le pointeur du contenu dans le noeud (&)
3) Savoir si 2 itérateurs distincts sont sur le même nœud (==)
4) Savoir si 2 itérateurs distincts ne sont pas sur le même nœud (!=)

Pour ce faire, c'est la liste qui nous retournera un itérateur sur sa structure. On ajoutera 2 méthodes dans la liste chaînée double circulaire :

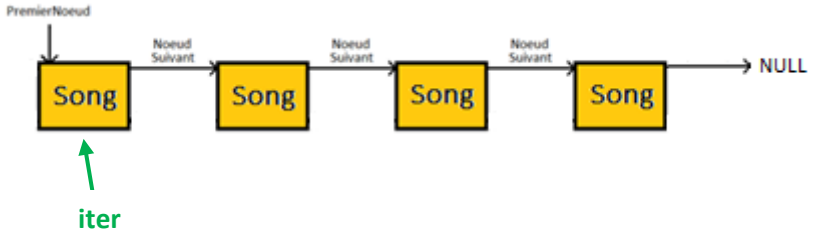
1) begin() qui retournera un itérateur sur son premier nœud (code donné)
2) end() qui retournera un itérateur sur NULL (le end est toujours situé immédiatement <u>après</u> la structure)

Exemple d'utilisation avec un itérateur sur une liste chaînée de chansons :

1) Obtention de l'itérateur sur le premier nœud:

```
// On obtient un itérateur sur le premier noeud, c'est la liste qui nous le prépare
Iterator<Song> iter = list.begin();
```

Ce qui nous donnerait un « curseur » positionné à cet endroit :



En surchargeant l'opérateur ++ et & dans l'itérateur, on pourrait ainsi faire des opérations du genre :

```
/*Attention, un itérateur n'est pas une chanson, c'est un objet qui contient un noeud courant
(qui contient une chanson). L'opérateur & est surchargé dans l'itérateur, il retourne le pointeur sur la chanson*/

iter++; //déplace l'itérateur sur le 2e noeud
Song* song = &iter; //pointeur sur la chanson
```

Ainsi, le code (très simplifié) de la « playlist » ci-haut deviendrait :

```
/*.h*/
Iterator<Song> iter = list.begin();

/*.cpp*/
public void onKeyDown()
{
    iter++;
}

public void play()
{
    player.play((&iter)->getTitle());
}
```

Ce qui nous abstrait complètement de la manipulation des nœuds.

Travail à effectuer :

1) begin() est donnée, codez le end() dans la liste doublement chaînée circulaire
2) Codez la classe iterator.h, c'est un template également

Tests unitaires

Le test nommé **iteratorShouldCompile** doit compiler sans modification de code dans le test. Revoyez vos noms de méthodes ou attributs si ce n'est pas le cas !

3) Écrire une suite de tests pertinents pour les opérateurs de l'itérateur

Partie 3 – Mastermind

Le déroulement du jeu est simple. Avant de démarrer la partie, le maître du jeu (c'est vous) doit choisir une combinaison secrète de 4 couleurs (parmi 8 proposées). Les combinaisons possibles (4096 possibilités) sont stockées dans une liste chaînée. Il est à noter que l'ordre importe peu, tant que toutes les combinaisons y soient.

Le Mastermind doit essayer de deviner la combinaison de 4 couleurs que le maître du jeu a choisi. Pour ce faire, le Mastermind doit proposer au maître du jeu une combinaison de 4 couleurs qu'il a lui-même choisi de façon aléatoire dans la liste.

Le maître du jeu doit ensuite dévoiler des indices sur la combinaison secrète en fonction de la combinaison proposée par le joueur (le verdict)

Les indices suivants sont délivrés couleur par couleur à partir de la combinaison proposée par le joueur :

Touche 1 : La couleur est présente dans la combinaison secrète et est à la bonne place dans la combinaison proposée;

Touche 2 : La couleur est présente dans la combinaison secrète mais n'est pas à la bonne place dans la combinaison proposée;

Touche 3 : La couleur n'est pas présente dans la combinaison secrète.

Suivant le verdict, la liste est nettoyée des combinaisons qui ne sont plus possibles. Le jeu se poursuit jusqu'à ce que le joueur trouve votre combinaison (il y a 8 essais possibles)

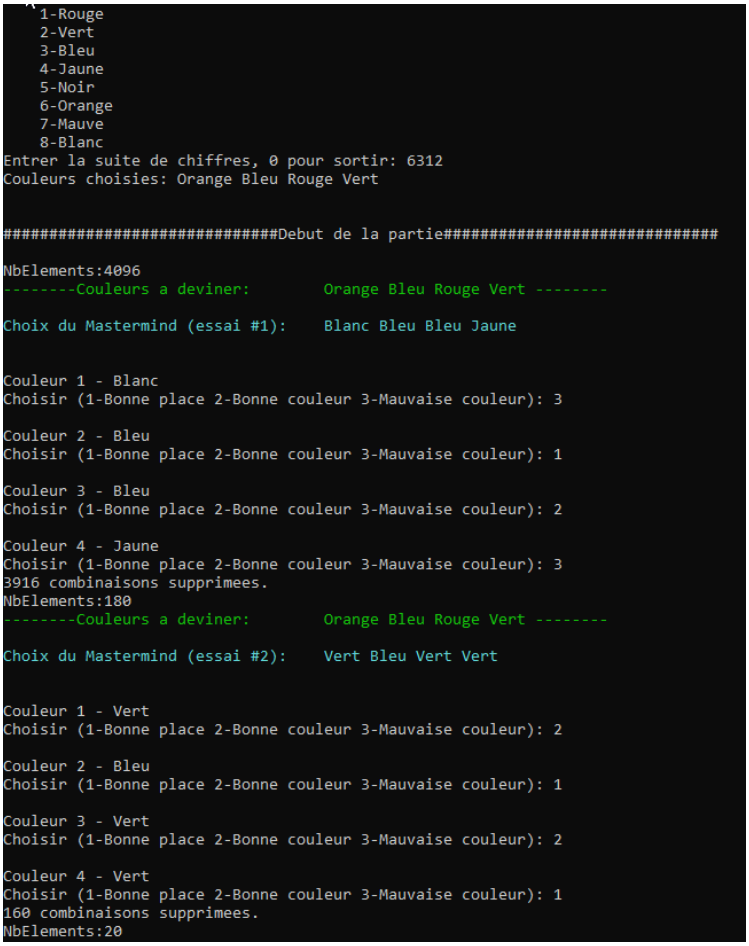
Travail à effectuer :

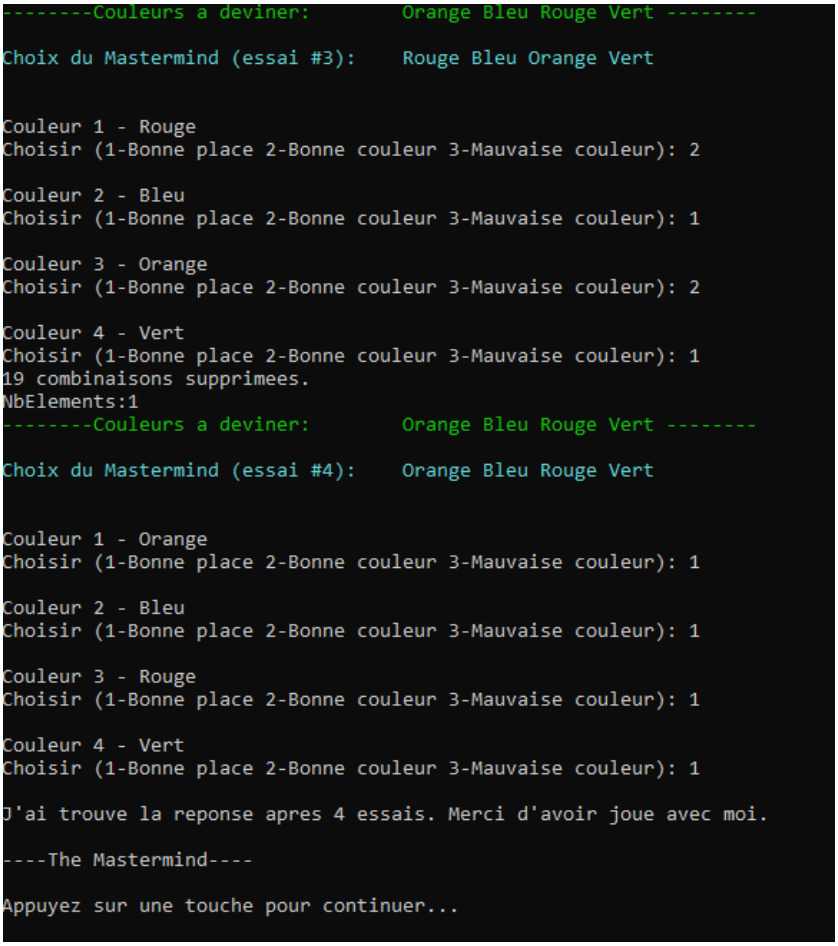
1) Prendre connaissance du code de départ, le déroulement du jeu dans le menu est donné et fonctionnel (saisies de la séquence et des verdicts)
2) Codez l'opérateur == de la classe Combinaison, il vise à vérifier si les combinaisons sont identiques (mêmes couleurs aux mêmes endroits)
3) Complétez la classe Mastermind pour faire fonctionner le jeu, les compositions de mémoire respectent celles montrées au cours.
4) Commentez en profondeur les méthodes développées dans le Mastermind

Contrainte :

Tout manipulation sur la liste dans Mastermind doit impérativement utiliser l'itérateur. Les fonctions de récupération de nodes et les déplacements avec getNext() sont interdits.

Exemple d'une partie :





MODALITES DE REMISE

La date de remise de ce travail est le : **mardi 1er octobre 2024 fin de journée (29h59)**

1) Remise LEA

- Remise par 1 membre de l'équipe seulement (désignez un responsable)
- Solution nettoyée (clean/nettoyer solution + .vs + extern supprimés)
- Le travail est archivé dans un fichier nommé XXXXXXX_XXXXXXX _TP2.zip où XXXXXXX représente vos matricules
- Le programme compile et s'exécute sur Visual Studio 2022 (logiciel supporté pour le cours, voir plan de cours)

 Les programmes qui ne compilent pas ou ne s'exécutent pas ne seront pas corrigés.