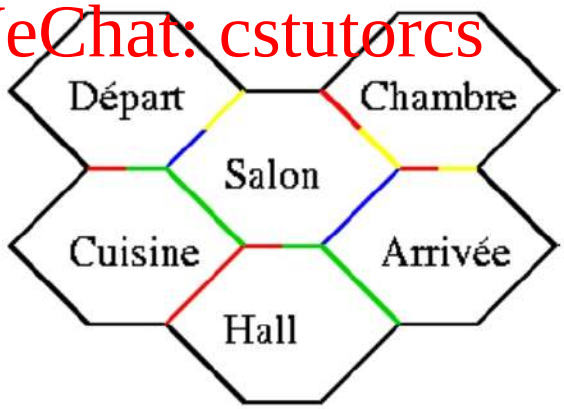
1 Mise en contexte

Imaginons un labyrinthe composé de pièces hexagonales reliées entre elles par des portes de couleurs différentes. Quatre personnages sont placés dans une pièce déterminée du labyrinthe appelée la case départ, et doivent se rendre en un minimum de déplacements à une autre case désignée comme la case d'arrivée. Les quatre personnages sont représentés par les couleurs rouge, vert, bleu et jaune. Chaque porte est aussi colorée en rouge, en vert, en bleu ou en jaune. Un personnage ne peut traverser que les portes de sa couleur. Plusieurs portes peuvent permettre le passage entre deux mêmes pièces. Par exemple, la case de départ pourrait avoir une porte rouge vers la case du bas, une porte jaune vers la case du bas, et une autre porte jaune vers la case en diagonale en bas à droite. Les portes ne sont pas à sens uniques. Elles peuvent donc être utilisées dans les deux sens. Entre deux pièces, il peut y avoir jusqu'à quatre portes, soit une de chaque couleur. De plus et étant donné que les pièces sont hexagonales, chaque pièce qui n'est pas sur la bordure du labyrinthe est adjacente à six autres pièces.

Pour un labyrinthe donné, les chances de chacun des joueurs d'arriver à la case d'arrivée le premier ne sont pas égales. Par exemple, des portes rouges pourraient être placées de façon à permettre au joueur rouge d'arriver à la case finale par un chemin en ligne droite, tandis que le joueur jaune pourrait être obligé de faire un détour coûteux.

Le but du présent travail est de déterminer, pour un labyrinthe donné en entrée au programme, lequel des quatre joueurs peut se rendre le plus rapidement, c'est-à-dire en le moins de déplacements de pièce en pièce, à la case d'arrivée. Par exemple, un labyrinthe peut ressembler au suivant :

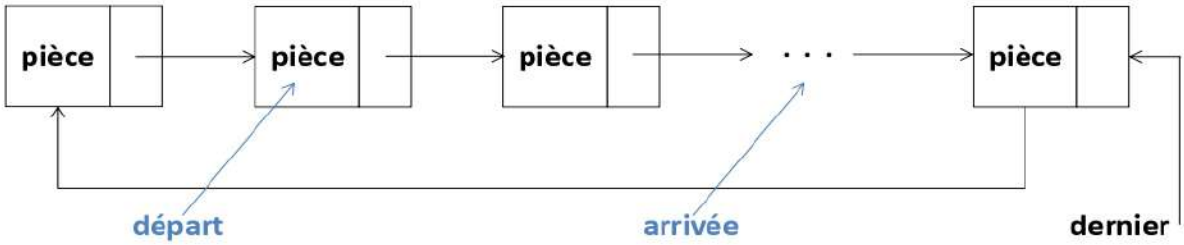


Dans ce cas, le joueur rouge peut se rendre du départ à l'arrivée en 5 déplacements, le joueur jaune peut le faire en 3 déplacements, le joueur vert peut y arriver en 4 déplacements, puis le joueur bleu peut y arriver en 2 déplacements. C'est donc le joueur bleu qui a le plus de chances de gagner.

2 Modélisation d'un labyrinthe hexagonale

L'ensemble des classes que vous devez utiliser pour réaliser ce travail pratique est défini dans les fichiers fournis (Labyrinthe.h, Porte.h, Piece.h, etc). Vous pouvez ajouter d'autres méthodes à ces classes, qu'elles soient publiques ou privées. Vous pouvez même ajouter des données membres, cependant, il est interdit de modifier les définitions qui y sont déjà présentes. Vous pouvez par contre ajouter des m_ aux données membres des classes fournies ou des p_ aux paramètres des méthodes fournies, mais ce n'est pas obligatoire. Le TP utilise d'ailleurs les classes suivantes:

1. Un labyrinthe utilisant une liste chaînée circulaire de pièces:



Elle contient un pointeur vers la dernière pièce de la liste, un pointeur vers la pièce de départ et un pointeur vers la pièce d'arrivée. Il ne faut pas confondre la première pièce de la liste avec la pièce de départ. L'ordre dans lequel les pièces sont placées dans la liste de pièces ne signifie rien. La première pièce de cette liste n'est pas nécessairement la pièce de départ, et la dernière pièce de cette liste n'est pas nécessairement la pièce d'arrivée du labyrinthe. Pour connaître la pièce de départ et la pièce d'arrivée, il faut utiliser les pointeurs *depart* et *arrivee*. Vous devez également utiliser une file (*queue* de la STL) afin d'implanter la méthode permettant de solutionner le labyrinthe. L'algorithme qu'il faut utiliser est bien détaillé dans le code fourni avec ce TP.

2. Une porte contenant une couleur (rouge, vert, bleu ou jaune) représentant un *enum* et un pointeur vers une pièce du labyrinthe.

3. Une pièce contenant une liste de portes qui est une liste doublement chainée utilisant le conteneur *list* de la STL.

La pièce contient également un nom (chaîne de caractères) qui doit être unique pour chaque pièce ainsi qu'un booléen facilitant l'implémentation de l'algorithme qui résout le labyrinthe pour un joueur.

Dans le modèle d'implantation imposé, une pièce contient une liste de portes qui la relie à d'autres pièces. Par contre, ces portes relient les paires de pièces dans les deux sens, même si elles ne se trouvent que dans la liste de portes de l'une des deux portes reliées. Dans votre algorithme qui résout le labyrinthe, vous devez absolument veiller à vérifier la possibilité, à partir d'une pièce, d'utiliser une porte dans sa propre liste de porte, mais aussi d'utiliser une porte dans la liste de porte d'une autre pièce qui pourrait mener à la pièce.

3 Travail à faire

Il s'agit d'implémenter dans les fichiers respectifs .cpp toutes les méthodes décrites dans les fichiers d'interface .h correspondant. Afin de vous aider, nous vous fournissons un code qui compile, mais qui ne donne pas évidemment le bon résultat. Il faut bien sûr compléter les méthodes demandées en modifiant éventuellement leur valeur de retour. Vous allez aussi devoir ajouter ou enrichir les commentaires d'interface et d'implémentation en format Doxygen dans les divers fichiers .cpp. Il n'est pas nécessaire d'ajouter des commentaires dans les fichiers .h puisque le correcteur va surtout voir ces commentaires dans les fichiers .cpp. Nous vous suggérons très fortement de créer des fichiers permettant de tester séparément les méthodes de chaque classe que vous devez implanter (les méthodes des classes Piece, Porte et Labyrinthe). Vous devez donc faire tous les tests nécessaires à chaque étape importante dans votre implémentation afin de vous assurer que cette implantation est correcte au fur et à mesure de sa conception.

Pour vous aider à tester le bon fonctionnement de votre implémentation (classes impliquées dans ce travail), nous vous fournissons un programme comportant une fonction main ainsi que la méthode qui charge le labyrinthe et celle qui ajoute une pièce. Ce programme devra être compilé avec votre fichier Labyrinthe.cpp complété et fera appel à toutes les méthodes des classes demandées, directement ou indirectement. Ce programme est écrit dans le fichier Principal.cpp que nous vous fournissons, et prend en entrée la description d'un labyrinthe en quatre fichiers, chacun décrivant la position des portes d'une couleur donnée.

La structure de chacun de ces fichiers est la suivante :

- Une première ligne spécifiant le nombre de colonnes du labyrinthe et le nombre de lignes.
- Un dessin du labyrinthe hexagonal tel que vu par l'un des joueurs, à l'aide des caractères /, \, et _. Dans ce dessin, le caractère D désigne la case départ et le caractère A (ou F) désigne la case d'arrivée.

Par exemple, voici un fichier qui décrit les portes rouges d'un labyrinthe :

```
10 8
 _   _   _   _   _   _   _   _   _   _   _
/D\_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \
\ / \_  \   \    _  \_  \   \_   _  \    _/
/   /  _  \_  \_/ \_  \_  \_  \_/   / \_  \
\ /  _/ \_/ \_/  _/ \_/ \_/ \   \ / \_   _/
/ \_/  _   _/   /  _   _   _/  _  \ / \_  \
\ / \_  \_  \ /  _/  _/  _/ \_/ \_/   /   /
/ \ /  _/  _/ \_/ \_  \_   _  \   \_/  _/ \
\_/  _  \_   _   _  \_  \_  \_  \_   _/A _/
  \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/ \_/
```

Remarquez que vous n'avez pas à vous attarder à la description formelle de ces fichiers de labyrinthe, puisque nous vous fournissons la méthode de la classe Labyrinthe qui les lit. Les quatre fichiers qui seront lus par le programme de test que nous vous fournissons correspondent aux portes rouges, aux portes vertes, aux portes bleues, puis aux portes jaunes d'un labyrinthe de même taille. Par exemple, si on considère le labyrinthe à la fin de la page 1 de cet énoncé, les quatre fichiers le décrivant contiendraient le texte suivant:

Rouge.txt :

```
3 4
 _   _
/D\_/ \
\ /   /
/ \ /A\
\_  \_/
  \_/
```

Vert.txt :

```
3 4
 _   _
/D\_/ \
\ / \_/
/   /A\
\_/  _/
  \_/
```

Bleu.txt :

```
3 4
 _   _
/D\_/ \
\_  \_/
/ \_ A\
\_/ \_/
  \_/
```

Jaune.txt :

```
3 4
 _   _
/D\_/ \
\_    /
/ \_/A\
\_/ \_/
  \_/
```

Il faudrait alors exécuter le programme en s'assurant que vous fournissiez ces quatre fichiers. Le programme de test nous répondrait donc que c'est le joueur bleu qui devrait gagner, puisqu'il peut solutionner le labyrinthe en seulement deux déplacements. Nous vous fournissons des labyrinthes déjà générés que vous pouvez utiliser dans vos essais (voir le répertoire « *data* ») ainsi que les résultats attendus de chacun de ces labyrinthes (voir fichier fourni *Resultats-Labyrinthes.pdf*). Il faut savoir en revanche que le correcteur pourra utiliser d'autres labyrinthes pour vérifier que votre code donne toujours le bon résultat.