INF-111 Hiver 2022

Travail pratique #2

Groupes : 1 et 3

Enseignant : Abdelmoumène Toudeft

Équipe : 3 étudiants maximum: (un seul rapport). Composition des équipes avant le **dimanche 20 février 23h59.**

Remise : **dimanche 13 mars à 23h59.**

Auteur : Frédéric Simard (H2022)

1 - Introduction

1.1 - Contexte académique

Ce devoir vous amène dans le monde du développement de jeux vidéos et vise à pratiquer les notions suivantes:

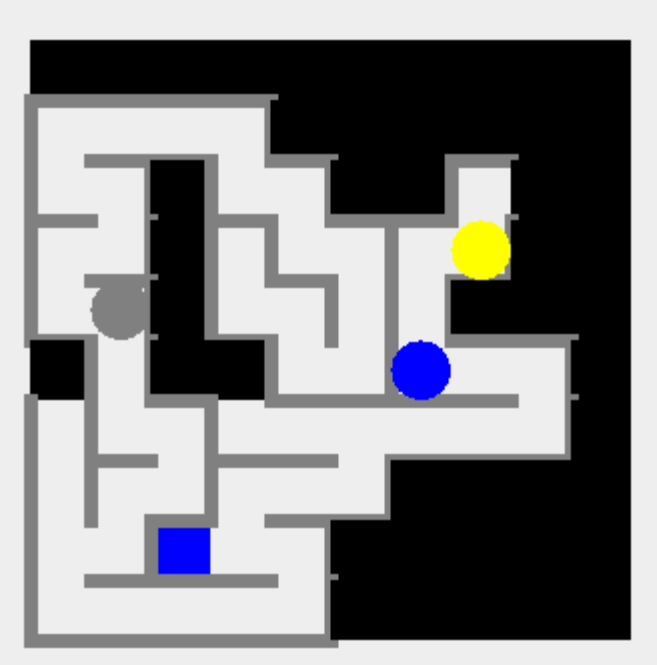
- Notions de base sur l'utilisation des Classes et de la programmation orienté-objets,
- L'organisation en packages
- Les tableaux 2D
- Les types de données abstraits (TDA), collections java et chaînage
- Algorithme simple.

Il est bon de noter que le même sujet sera poursuivi pour le devoir suivant, bien que la suite du développement visera d'autres objectifs.

1.2 - Description du problème

Pour ce devoir, vous devez construire les premières composantes d'un engin de contrôle pour un jeu vidéo.

La thématique du jeu se décrit comme suit, un héros décide de s'enfoncer dans un donjon réputé sans fin, dans le but d'ajouter un acte héroïque à sa liste d'exploits. Dans son périple, il fera face à de nombreux ennemis et profitera des équipements abandonnés par ceux qui l'ont précédé.



NOTE: Vous aurez à utiliser votre imagination dans ce TP, les moyens graphiques à notre disposition sont limités, dans l'image, le point jaune est le héros, les points gris et bleu sont des monstres du labyrinthe et le carré bleu est la sortie du niveau.

Le problème consiste à créer la mécanique de jeu. Quelques éléments graphiques vous sont fournis, mais vous êtes responsable de réaliser pratiquement tout l'engin de contrôle.

1.3 - Réalisation du travail

La conception du projet se fait selon une approche ascendante. Vous commencerez par créer les modules et classes qui forment la base nécessaire à la construction du donjon (Section 2), puis mettrez en place l'algorithme de génération aléatoire du donjon (Section 3). Ensuite, vous créerez les personnages qui évoluent dans ce donjon (Section 4), après quoi vous établirez la mécanique de jeu (Section 5).

Prenez note que la section 2 appelle principalement à des notions que vous devriez maîtriser déjà et des concepts simples. La réalisation de cette section devrait se faire rapidement. L'algorithme de la section 3, est probablement le défi le plus important de ce devoir. Les sections 4 et 5 ne sont pas excessivement longues, mais seront très difficiles pour ceux qui ne maîtrisent pas les concepts d'héritage et polymorphisme enseignés en classe.

Des indicateurs d'échéancier sont donnés dans l'entête de chaque section. Si vous ne respectez pas ces échéances, vous ne passerez pas un bon moment.

1.4 - Fichiers fournis

Un certain nombre de fichiers vous ont été remis. Ceux-ci sont soit à utiliser comme tel, à compléter ou à intégrer dans votre code. À tout moment, vous devrez vous assurer que le code que vous avez développé s'intègre bien avec le code fourni. Une bonne façon d'y arriver: respecter les noms des méthodes et constantes indiquées.

Les fichiers fournis sont à intégrer en deux parties. Le fichier de configuration est à intégrer dès le départ, alors que les autres sont ajoutés à la section 3.

Voici une courte description des fichiers fournis:

- *donjon::Configuration*, un fichier contenant les valeurs qui définissent les configurations du donjon.
- *vue::** Toutes les classes responsables de l'affichage graphique. Il peut valoir la peine de les regarder, puisqu'elles font appels aux méthodes que vous développerez.
- *observer::*;interfaceUtilisateur::** Classes responsable de la communication entre les éléments du jeu et de la gestion du clavier pour les mouvements du héros.

Bon travail!!!

2 - Modules et classes initiales (Semaine 1)

Cette section vous guide dans la réalisation des trois classes suivantes: physique::Position, physique::Direction et pile:PileSChainee.

Note: La convention utilisée est package::Classe

2.1 physique::Position

Cette classe implémente l'objet Position qui est utilisé pour garder en mémoire une position dans le donjon, ou pour garder en mémoire une position donnée en Pixel (vue::*). La convention utilisée dans le projet est qu'une position est toujours donnée en coordonnées i, j. La coordonnée i se développe dans l'axe vertical et la coordonnée j, dans l'axe horizontal. Prenez note qu'une paire de coordonnée: {1,0} indiquera i=1, j=0, ce qui donne un vecteur qui pointe directement vers le haut. Cette convention est différente de la norme {x,y} par contre elle à l'avantage de fonctionner partout dans le programme, de la même façon.

2.1.1 champs membres

Cette classe contient donc les membres suivants:

- i et j, des entiers donnant la valeur de la position.

2.1.2 méthodes membres

Cette classe doit offrir les services suivants:

- **constructeur par paramètre**, qui reçoit des valeurs pour i et j.
- **constructeur par copie**, qui reçoit une référence sur une position.
- **accesseur {informateur/mutateur}** pour chacune des coordonnées
- **clone()** copie profonde
- **equals()** comparaison profonde
- **additionnerPos**, qui reçoit une position *pos* et qui exécute l'opération suivante:

  $$this_i = this_i + pos_i$$
  $$this_j = this_j + pos_j$$

  cette méthode n'a pas de retour.
- **soustrairePos**, qui reçoit une position *pos* et qui exécute l'opération suivante:

  $$this_i = this_i - pos_i$$
  $$this_j = this_j - pos_j$$

  cette méthode n'a pas de retour.
- **multiplierPos**, qui reçoit deux doubles pos_i et pos_j, et qui exécute l'opération suivante:

  $$this_i = this_i * pos_i$$
  $$this_j = this_j * pos_j$$

  cette méthode n'a pas de retour.

2.1.3 validation

Créez un programme principal et faites quelques tests pour vous assurer que votre classe fonctionne bien. Montrer rapidement ce programme à votre chargé de cours, si vous avez des doutes.

2.2 physique::Direction

Cette classe implémente le module utilitaire qui est utilisé pour effectuer des opérations décrites sous forme de direction. Elle offre également certains services pour travailler avec des positions.

2.2.1 constantes publiques

Les constantes(valeurs) suivantes définissent les directions.

- HAUT(0), BAS(1), GAUCHE(2), DROITE(3)

2.2.2 champs membres

Cette classe possède une instance d'un générateur de nombres pseudo-aléatoires (Random).

2.2.3 services offerts

Cette classe doit offrir les services suivants:

- **directionOpposee**, ce service permet d'obtenir la direction opposée à celle reçus en paramètre.
- **directionAPosition**, ce service permet de convertir une direction en sa représentation Position. Voici la table de correspondance à utiliser. Prenez note que la convention utilisée est très importante et n'a pas été choisie au hasard. Les raisons deviendront évidentes un peu plus loin dans le travail.

HAUT	**{-1,0}**
BAS	**{1,0}**
GAUCHE	**{0,-1}**
DROITE	**{0,1}**

- **positionADirection**, ce service permet d'effectuer la conversion inverse

{-1,0}	**HAUT**
{1,0}	**BAS**
{0,-1}	**GAUCHE**
{0,1}	**DROITE**
autre	**-1**

- **obtenirDirAlea**, ce service permet d'obtenir une direction aléatoire.

2.2.4 validation

Ajouter au programme principal quelques tests pour vous assurer que votre classe fonctionne bien. Tester toutes les méthodes. Montrer rapidement ce programme à votre chargé de cours, si vous avez des doutes.

2.3 pile::PileSChainee

2.3.1 validation

Vous devez programmer et intégrer une pile à votre programme. L'architecture choisie est la pile simplement chainée. Les services nécessaires:

- **empiler**, ajoute un élément à la pile
- **depiler**, enlève un élément de la pile
- **regarder**, retourne une référence sur le prochain élément de la pile sans l'enlever
- **estVide**, indique si la pile est vide.

Réalisez cette pile en suivant les instructions de votre chargé de cours.

2.3.2 validation

Ajouter au programme principal quelques tests pour vous assurer que votre classe fonctionne bien. Tester toutes les méthodes. Montrer rapidement ce programme à votre chargé de cours, si vous avez des doutes.

3 - Donjon (Semaine 1-2)

3.1 donjon::Case (semaine 1)

La case est l'objet de construction du donjon. Dans les faits, le donjon est un tableau 2D de cases. Voici donc les détails pour l'implémentation de la case.

3.1.1 champs membres

- une référence à une position
- un booléen indiquant si la case a été découverte par le héros.
- un booléen pour indiquer si la case est la fin d'un niveau.
- un booléen indiquant si la case a été développée par l'algorithme de labyrinthe.
- un tableau de 4 Case, pour contenir des références sur les voisins connectés.

3.1.2 services offerts

- Constructeur par paramètre qui reçoit une référence à une position.
- Informatrice pour obtenir une copie de la position membre.
- accesseurs {informateur/mutateur}, pour le champ boolean: developpe
- setVoisin, permet d'attacher un voisin pour la direction donnée. L'index du tableau est donné sous la forme d'une direction (section 2.2)
- getVoisin, permet d'obtenir un voisin. L'index du tableau est donné sous la forme d'une direction (section 2.2)
- accesseurs {informateur/mutateur} **getFin**, pour le champ indiquant si la case est une fin.
- accesseurs {informateur/mutateur} **get/setDecouverte**, pour le champ indiquant si la case a été découverte par le héros.
- accesseurs {informateur/mutateur} **get/setDeveloppe**, pour le champ indiquant si la case a été développée.
- toString, cette méthode n'est pas utilisée dans le programme, mais pourrait s'avérer utile pour le débuggage.

3.2 donjon::Donjon (semaine 2)

3.2.1 champs membres

- référence à la case départ
- référence à la case de fin
- tableau 2D de Cases
- instance de la classe Random

3.2.2 services offerts

- **contructeur par défaut**:, le constructeur sera complété plus tard, mais pour l'instant vous pouvez effectuer les opérations suivantes:
    - obtient une référence aux configurations
    - initialise le tableau 2D à l'aide des dimensions provenant des configurations.
    - définit une case départ choisie au hasard
- **informatrice** pour case de départ et case de fin
- **informatrice** pour obtenir une référence sur le tableau 2D
- **getPositionAlea**, retourne une une position, choisie aléatoirement à l'intérieur du donjon.
- **getNbVoisinsNonDeveloppe**, cette méthode reçoit en paramètre la position de la case présentement évaluée et compte le nombre de voisins non développé d'une case.

Voici le pseudo-code de cette méthode:

```
compte <- 0
Pour toutes les directions
      calcule la position de la case voisine (directionAPosition, additionnerPos)
      SI la position est dans le labyrinthe et
      SI la case n'est pas développée
            incrémente le compte
retourne le compte
```

- **getVoisinAlea**, cette méthode reçoit en paramètre la position de la case présentement évaluée et retourne une référence vers un voisin choisi aléatoirement.

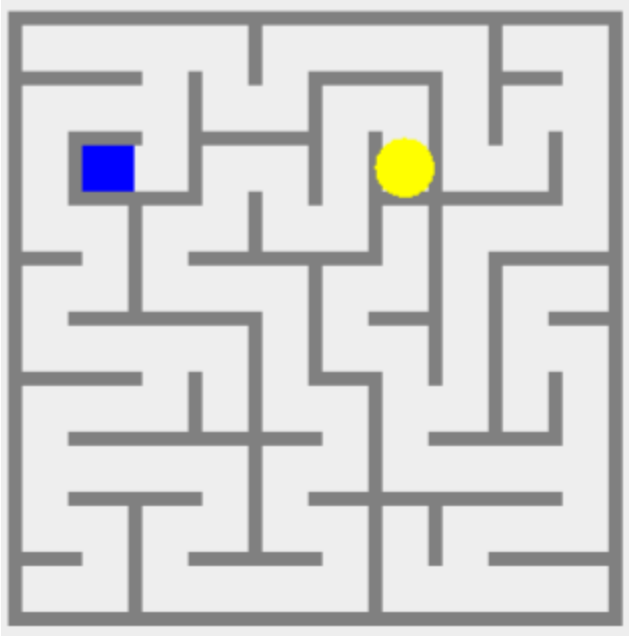
Voici le pseudo-code de cette méthode:

```
faire
     choisi une direction aléatoire
     transforme la direction en position
     additionne la position de référence à la position aléatoire
TANT QUE la position générée n'est pas valide
retourne la position valide
```

- **getVoisinLibreAlea**, cette méthode reçoit en paramètre la position de la case présentement évaluée et retourne une référence vers un voisin libre qui n'a pas été développé.
  Cette méthode se développe en utilisant la méthode précédente, mais répète l'opération jusqu'à ce que la case tirée aléatoirement soit non développée.
- **produireLabyrinthe**, cette méthode crée le labyrinthe dans le donjon.

Voici la source de l'algorithme de labyrinthe utilisé:
https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_d%27un_labyrinthe#Exploration_exhaustive

et voici un exemple de labyrinthe tiré de la solution.



L’algorithme de production de labyrinthe utilise la pile. Si, à ce moment-ci, votre pile n’est pas testée, faites-le maintenant, sinon vous ne passerez pas un bon moment.

Voici les commentaires tirés de la solution., il vous reste à y ajouter le code (chaque * étoile représente une ou des lignes de codes). Note, une ligne de code vous est fournie, pour vous aider.

```
// développe le labyrinthe à partir de la case départ
// l'empile
*

//tant que la pile n'est pas vide, continue
*

      // prend la case du haut de la pile sans l’enlever
     *

      // obtient sa position
     *

      // indique que cette case est maintenant développée
     *

      // vérifie si cette case a un voisin non développé
     *
            // oui, choisit une case non développée voisine au hasard
            *

            // obtient la position du voisin
```

```
            *

            // calcul la direction du voisin
            // position voisin moins position case courante
            // -> position à direction
            *

            // ajoute à la case, comme voisin réciproque
            // appel à setVoisin pour les deux cases
            // note: la droite d'une case est la gauche de l'autre,
            // utiliser directionOpposee
            *

            // ajoute le voisin à la pile
            *

            // définit la fin comme étant la dernière case développée
            this.fin = (Case)pile.regarder();
      //sinon
            // il s'agit d'un cul-de-sac, dépile une case
            *
```

Ajoutez maintenant les deux lignes suivantes à votre constructeur:

```
// produit le labyrinthe
this.produireLabyrinthe();

// assigne la fin
this.fin.setFin();
```

3.2.3 Intégration du code fournir

Il est maintenant temps d'intégrer le fichier .zip fournis pour la Section 3. Dans le paquet, on trouve:

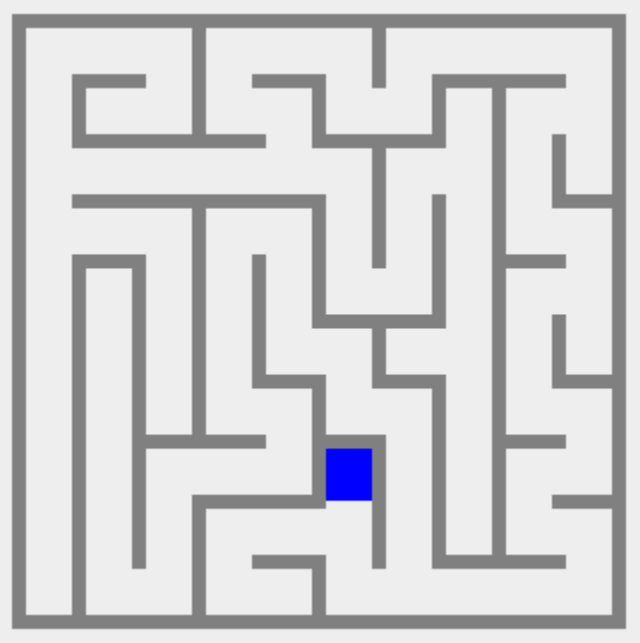
- le package **vue**, qui contient toutes les classes relatives à l'engin graphique. Plusieurs de ces classes utilisent votre code. Par contre, ce fichier ne requiert pas de modification pour l'instant.
- le package **programme**, qui contient le programme principal.
- le package **observer**, qui contient un élément de signalement entre classes (présenté plus tard dans la session). Il est complet et sera présenté en classe**.**
- le package **interfaceUtilisateur**, qui effectue la gestion des événements flèche de clavier.

- Le **programmePrincipal**, ne contient que le code nécessaire à lancer la *vue.* Vous n'avez pas à vous en soucier, autrement que ce fichier devrait remplacer le programme que vous avez écrit pour valider votre code.

NOTE: il serait sage de garder une copie de votre programme de test, mettez-le ailleurs dans le projet

- Le package **modele**, contient la classe **PlanDeJeu**. Cette classe est incomplète, mais n'a pas à être modifié avant la section 4. de même pour l'**interfaceUtilisateur.**

Après avoir intégré les nouveaux fichiers dans votre projet, vous devriez être capable d'exécuter programme et obtenir un résultat similaire à ceci:



Si votre code ne fonctionne pas, commencez par vérifier que vous avez utilisé les bons noms de classes, de package et que toutes les méthodes fonctionnent correctement.

Vérifiez également votre implémentation de l'algorithme de labyrinthe. Si ce dernier ne fonctionne pas, demander au chargé de cours de vous donner un coup de main. En échange de la démonstration que vous avez fourni un bon effort, le chargé de cours vous donnera un coup de pouce pour mettre tout en ordre.

4 - Personnages (Semaine 3)

Il est maintenant temps d'ajouter un peu de vie à notre labyrinthe. Il y a deux types de personnages dans le labyrinthe, les créatures et le héros. De plus, les créatures viennent en trois types: araignées, minotaure et dragon.

Cette description nous amène à déduire qu'une conception par héritage est approprié.

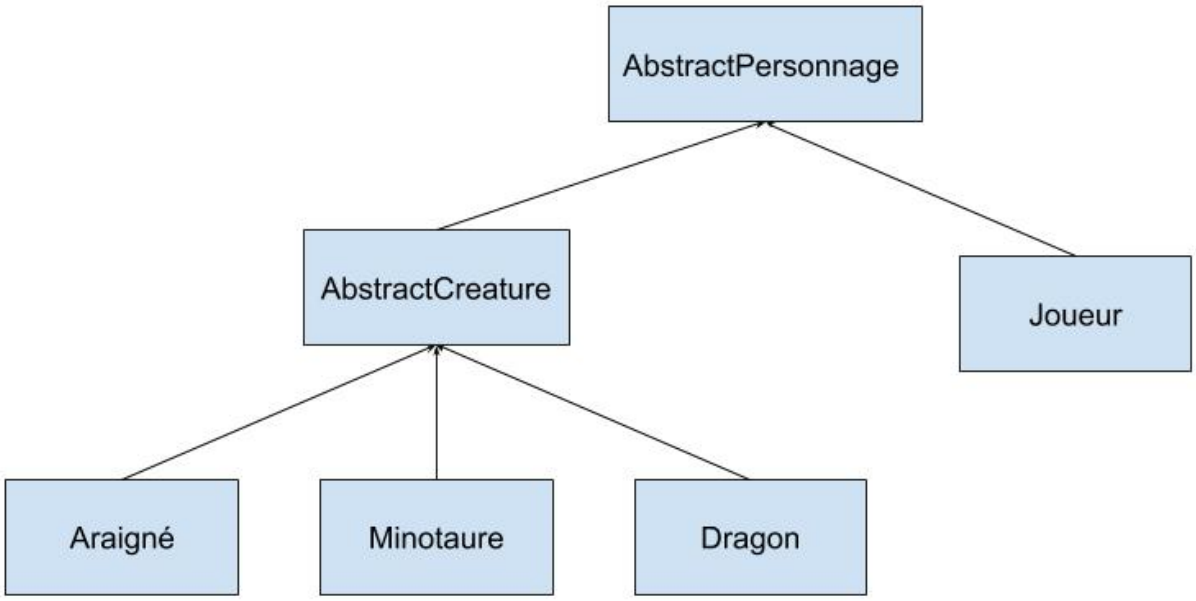
4.1 Hiérarchie de classes dérivant de Personnages

Cette description nous amène à déduire qu'une conception par héritage est appropriés. Une classe de base vous est fournie (fichiersFournisSection4). Cette classe est personnage::AbstractPersonnage. Nous reviendrons en classe sur le concept de classe abstraite, mais pour l'instant il suffit de considérer que ce sera la classe de base de notre hiérarchie.

4.1.1 AbstractPersonnage

La classe de base AbstractPersonnage définit tous les membres et méthodes qui sont communs à tous les personnages, ainsi que les implémentations de base des méthodes.

Voici une présentation de la hiérarchie de classe à réaliser. Dans ce cas-ci la classe AbstractPersonnage vous est fournie, alors que les autres sont à réaliser.



4.2 Créatures

4.2.1 AbstractCreature

La classe abstraite AbstractCreature est définie pour permettre d'appliquer le polymorphisme sur les créatures du donjon. La classe elle-même ne change pas les fonctionnalités du personnage. La seuel méthode à définir est le constructeur par paramètre qui ne fait qu'appeler le constructeur de AbstractPersonnage.

4.2.2 Minotaure et Dragon

Ces deux classes ne changent pas le comportement de AbstractCreature, mais ne sont pas des classes abstraites. Ces classes seront utilisées plus tard dans le projet, pour l'instant elles ne sont pas particulières. Suffit de répéter le constructeur par paramètre et appelé celui de la classe de base.

4.2.3 Araigné

La créature de type araignée à la particularité de se déplacer en faisant des bonds. En plus de répéter le constructeur par paramètre, elle redéfinit la méthode héritée seDeplacer, de façon à ce que la méthode de la classe de base soit appelée deux fois plutôt qu'une.

4.2.4 Intégration des créatures dans le programme

Voici les éléments à modifier/ajouter au programme pour que des créatures commencent à peupler votre donjon.

PlanDeJeu

- ajouter un vecteur de AbstractCreature
- ajouter une méthode privée initCreature, cette méthode se comporte comme ceci:

  - obtient une référence aux cases du donjon
  - obtient une référence aux configurations
  - vide le vecteur de créature (au cas où il y en aurait)
  - pour le nombre de créatures définies dans les configurations
    - choisit un type de créature aléatoirement
    - tire une position aléatoirement
    - initialise une créature du type définit
    - attache le plan de jeu comme observer (les créatures sont observable)
    - définit la case de la créature à la case correspondant à la position aléatoire.
    - ajoute la créature au vecteur

- ajouter un appel à initCreature dans la méthode nouveauNiveau
- ajouter une boucle à la méthode run() qui déplace toutes les créatures, dans une direction aléatoire.
- ajouter une méthode getCreatures qui retourne une référence au vecteur.

EnginDessinDonjon

- décommenter les méthodes:
  - dessinerCreatures
  - dessinerCreature

4.2.5 Validation

Pour le test, commenter le **IF** suivant:

```
if(creature.getCase().getDecouverte()){}
```

mais garder la ligne

```
dessinerCreature(g2, convertirIJaPixel(creature.getPos()), creature);
```

active.

Lancer le programme et valider que des créatures (cercles de couleur) se déplacent dans votre donjon.

Réactiver le **IF**.

4.3 Joueur

4.3.1 Intégration

Le joueur est votre dernier défi pour ce devoir. Pour l'intégration du joueur, vous devez suivre les mêmes étapes que pour les créatures, à l'exception de l'étape qui génère les déplacements aléatoires. À la place, vous devez intégrer le joueur dans la classe interfaceUtilisateur::ControleurClavier. Le code est déjà fourni et commenté.

4.3.2 Validation

Lancer le programme et valider qu'un joueur est dans le labyrinthe et qu'il est possible de le déplacer avec les flèches du clavier.

5 - Finalisation (Semaine 3)

Quelques lignes de codes vont ont été fournis commenté, pour que vous puissiez voir le labyrinthe. Suffit maintenant de décommenter les lignes dans la méthode EnginDessinDonjon::dessinerCase, pour cacher les cases non explorées du donjon.

Félicitation, vous avez terminé le devoir #2