Projet C++

Polytech' Paris-Saclay - Et4 info
Marc Fonvieille et Joël Gay

1 Objectifs du projet

L'objectif de ce projet est de mettre en œuvre les connaissances que vous avez acquises durant l'enseignement de la programmation orientée objet appliquée au langage C++. Dans ce projet vous allez devoir rédiger un cahier des charges sous la forme d'un diagramme de classes accompagné de commentaires, afin de comprendre ce que vous devez développer avant de commencer l'étape de programmation.

Ce projet est orienté objet, c'est-à-dire qu'il vous est demandé de mettre en pratique les notions vues en cours (héritage, encapsulation, classes, polymorphisme, surcharge, template, etc.). Le rapport est à rendre au plus tard le **mardi 26 janvier 2021 à 23H59 (Heure de Paris)**. Le projet aura une séance de soutenance le **vendredi 29 janvier 2021** suite à laquelle le code devra être rendu. Tout retard sera pénalisé.

Vous continuerez à travailler sur les exercices de TP **en parallèle** avec le projet. A l'issue de la dernière séance, vous aurez toujours la possibilité de poser des questions par mail. Ce projet doit être réalisé **en binôme** ou **seul**. Aucun groupe de taille plus grande ne sera accepté. Dans le cas d'un binôme, le travail de chacun devra être apparent dans le rapport. Le sujet du projet a été rédigé de manière à décrire les fonctionnalités basiques du système. Vous êtes libres de réaliser des fonctionnalités supplémentaires avec l'explication appropriée. De plus, il n'y a pas de précisions concernant l'implémentation ou la structure interne du projet qui vous sont laissées au choix.

2 Consignes générales et évaluation

L'évaluation comportera deux volets : l'évaluation du code et celle d'un rapport succinct.

Le projet doit être indépendant de la plateforme utilisée. Il doit pouvoir être compilé par `g++` sans warning (et évidemment sans erreur). Vous fournirez obligatoirement un *makefile* qui permettra de lancer la compilation facilement. Le respect des fonctionnalités imposées et la qualité du code (structure, utilisation **pertinente** d'un maximum de concepts - héritage, polymorphisme etc. - vus en cours et en TP, commentaires) seront primordiaux. L'ajout de fonctionnalités sera un plus.

Le rapport succinct prendra la forme d'un texte **au format PDF** qui expliquera les choix de conception que vous avez faits et en quoi ils permettent de répondre au cahier des charges. Vous inclurez dans le rapport la version finale du cahier de charges, avec le diagramme des classes et commentaires sur le rôle de chaque classe dans le projet. Vous devrez également rapporter les problèmes rencontrés et comment vous les avez surmontés - ou en quoi ils vous ont empêché de terminer une partie du projet. Pensez à inclure des captures d'écran de votre programme en fonctionnement (en annexe si vous manquez de place).

Votre travail (code + rapport) doit être rendu sous forme d'une archive ZIP. Vérifiez que le code envoyé inclue seulement les fichiers `.cpp`, `.h(pp)` et `.txt` avec les données nécessaires au fonctionnement de votre programme, ainsi que le `Makefile`.

3 Sujet

Le but de ce projet est de concevoir et implémenter un mini-jeu sur console. Ce jeu est (de façon lointaine) inspiré de *Age of War*[1], qui se joue à deux (humain contre IA). Nous développerons ici une adaptation de ce jeu avec un mode **"humain" contre "humain"** et un mode **"humain" contre "IA"**, en tour par tour.

3.1 Aire de jeu

L'aire de jeu est une ligne de 12 cases numérotées de 0 à 11. Sur les cases extrêmes (0 et 11) se trouvent les bases de chacun des joueurs. Le but de chaque joueur est de détruire la base adverse. Pour cela, chaque joueur peut créer une fois par tour (à la fin du tour) sur la case de sa base une unité. L'unité pourra ensuite avancer, attaquer d'autres unités ou la base adverse.

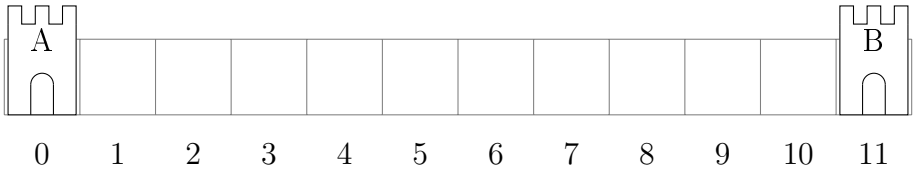

FIGURE 1 – Aire de jeu

Chaque base a au début de la partie 100 points de vie.

3.2 Les unités

Il existe 4 types d'unités :

- les fantassins, unités de combat rapproché
- les archers, unités d'attaque à distance
- les catapultes, unités d'attaque de zone à distance
- les super-soldats, unités de combat rapproché qui ne peuvent être recrutés directement

Les caractéristiques sont précisées dans le tableau 1.

	Fantassin	**Archer**	**Catapulte**	**Super-soldat**
Prix (pièces d'or)	10	12	20	-
Points de vie	10	8	12	10
Points d'attaque	4	3	6	4
Portée	1	1,2 ou 3	2 à 3 ou 3 à 4	1

TABLE 1 – Caractéristiques des unités

La portée définit quelles sont les cases sur lesquelles peut attaquer l'unité. Par exemple, un soldat du joueur B situé sur la case 5 ne peut attaquer qu'une unité ennemie sur la case 4.

L'archer peut attaquer une case parmi les 3 cases qui se présentent devant lui. **Il attaquera forcément l'unité ennemie qui est la plus proche de lui**.

1. `http://www.larouille.fr/jouer/defense/age-of-war`

La catapulte tire sur deux cases à la fois y compris si une unité alliée s'y trouve (cette unité prendra alors des dégats). Comme l'archer, elle tirera en priorité de façon à toucher la cible la plus proche.
Le tableau 2 récapitule de façon exhaustive quelle case sera attaquée dans quelle situation.

Unité	**Position de l'ennemi le plus proche**			
	$c+1$	$c+2$	$c+3$	$c+4$
Fantassin/Super-soldat	$c+1$	-	-	-
Archer	$c+1$	$c+2$	$c+3$	-
Catapulte	(*)	$c+2$ et $c+3$	$c+3$ et $c+4$	$c+3$ et $c+4$

TABLE 2 – Case(s) attaquée(s) par une unité du joueur A placée sur la case c. (*) Si la case $c+1$ est occupée par un ennemi, c'est la position du second ennemi le plus proche qui compte pour la catapulte.

Chaque unité a à sa disposition jusqu'à 3 actions précises qui dépendent uniquement de son type. Un fantassin et un archer ne peuvent attaquer qu'une seule fois par tour. Une catapulte ne peut effectuer qu'une seule action par tour (soit attaquer soit avancer d'une case).

	Fantassin	**Archer**	**Catapulte**	**Super-soldat**
Action 1	Attaquer	Attaquer	Attaquer	Attaquer
Action 2	Avancer	Avancer	-	Avancer
Action 3	Attaquer (*)	-	Avancer (*)	Attaquer

TABLE 3 – Actions disponibles pour chaque unité. (*) Les fantassins et catapultes ne peuvent effectuer l'action 3 que s'ils ont été dans l'impossibilité d'effectuer leur action 1.

Il ne peut y avoir qu'une seule unité par case ; une unité ne peut donc avancer que si la case devant elle est libre. Le fonctionnement du jeu impose donc que les unités ennemies se trouvent entre ses propres unités et la base adverse. Si une unité occupe la case de la base adverse, elle doit être détruite avant de pouvoir attaquer la base elle-même.
Une unité ne peut se déplacer que jusqu'à la case adjacente à la base adverse : case 10 pour les unités du joueur A et 1 pour les unités du joueur B.

3.2.1 Déroulement d'un tour de jeu

Chaque tour de jeu se déroule de la façon suivante :

1. Chaque joueur reçoit 8 pièces d'or.
2. Tour de jeu du joueur A
   (a) Phase de résolution des actions 1 des unités du joueur A
   (b) Phase de résolution des actions 2 des unités du joueur A
   (c) Phase de résolution des actions 3 des unités du joueur A
   (d) Création éventuelle d'une nouvelle unité du joueur A
3. Tour de jeu du joueur B
   (a) Phase de résolution des actions 1 des unités du joueur B
   (b) Phase de résolution des actions 2 des unités du joueur B

(c) Phase de résolution des actions 3 des unités du joueur B

(d) Création éventuelle d'une nouvelle unité du joueur B

L'ordre dans lequel les unités effectuent leur action dépend de la phase :

- Lors de la phase de résolution des actions 1, **l'unité la plus proche** de la base du joueur courant tente d'effectuer son action en premier, puis **la deuxième unité la plus proche** et ainsi de suite jusqu'à **l'unité la plus lointaine** de la base du joueur.
- Lors des phases de résolution des actions 2 et 3, **l'unité la plus lointaine** de la base du joueur courant tente d'effectuer son action en premier, puis **la deuxième unité la plus lointaine** et ainsi de suite jusqu'à **l'unité la plus proche** de la base du joueur.

A chaque étape, si l'unité a la possibilité de faire son action, elle l'effectue **obligatoirement**.

A la fin de son tour de jeu, le joueur a la possibilité de recruter une unité sur la case de sa base s'il possède assez de pièces d'or (le coût de recrutement est alors retranché de ce qu'il possède) **et si la case de sa base est libre**.

3.3 Combats

Lorsqu'une unité qui a a points d'attaque engage le combat contre une unité qui a p points de vie, cette dernière est blessée et il lui reste à l'issue de l'attaque $p - a$ points de vie. Il en va de même lors de l'attaque de la base adverse.

Lorsque le nombre de points de vie d'une unité est inférieur ou égal à zéro, elle disparaît et la case qu'elle occupait devient vide. Le joueur dont l'unité a remporté le combat reçoit de l'or supplémentaire équivalent à la moitié du prix de l'unité défaite (5 pour un fantassin ou un super-soldat, 6 pour un archer et 10 pour une catapulte).

De plus, en cas de combat mortel entre deux fantassins, le fantassin victorieux se transforme en super-soldat immédiatement, tout en conservant ses caractéristiques courantes (en particulier le nombre de points de vie.). Si l'attaque constituait son action 1, il pourra attaquer une deuxième fois dans le tour.

3.4 Fin du jeu

Le jeu se termine si l'une des deux conditions suivantes est remplie :

- une des bases est détruite : le joueur dont la base est détruite a perdu ;
- un certain nombre de tours (par exemple 100) a été joué par chaque joueur : il n'y a pas de vainqueur.

4 Consignes supplémentaires

4.1 Implémentation (dont rappels)

- Proposez une modélisation objet du problème.
- Proposez un mode d'affichage qui permettra à deux joueurs de jouer de façon relativement conviviale (sans pour autant aller jusqu'à l'interface graphique) :
  - Représentation de l'aire de jeu
  - Différenciation de différents types d'unités / bases des joueurs
  - Affichage des points de vie restants pour chaque unité
  - Affichage des pièces d'or disponibles pour chaque joueur
  - Affichage des actions de chaque unité dans l'ordre
- Choisissez vos structures de données dans la STL.
- Votre implémentation du jeu doit permettre à deux joueurs de jouer ensemble et doit vérifier à chaque étape que les règles du jeu sont respectées.
- Elle doit permettre aussi de jouer à 1 seul joueur, contre l'ordinateur.
- Il faut pouvoir enregistrer l'état courant du jeu dans un fichier texte (XML ou autre), et le recharger pour recommencer le jeu à partir du même point.

4.2 Stratégie

Le travail que vous devez effectuer se concentre sur l'aspect simulation du jeu, qui est déterministe. Seule la stratégie de création d'unité n'est pas imposée. L'implémentation de stratégies complexes ne sera notée (comme bonus) que si tout le reste fonctionne correctement (et seulement dans ce cas, pensez à faire une description de votre stratégie dans le rapport).

Une stratégie simple pour tester votre jeu est celle qui consiste à créer automatiquement l'unité la plus chère que ce que l'argent du joueur permet.

4.3 Proposition de fonctionnalités optionnelles

- Reporter toutes les caractéristiques de jeu (caractéristiques et prix des unités, taille de l'aire de jeu, quantité d'or reçue à chaque tour...) dans un fichier texte de configuration que vous lirez lors du lancement du programme. Ne pas oublier la documentation.

Pour toutes les fonctionnalités qui pourraient requérir un passage d'argument lors du lancement du jeu (fichier de sortie, fichier de configuration, mode de jeu...), un rappel est donné en annexe (5).

5 Annexe : *parsing* d'argument

Comme en C, on peut passer des arguments à la fonction `main` en utilisant :

```
int main(int argc, char * argv[]);
```

`argc` représente le nombre d'argument et `argv` leur liste. `argv[0]` est toujours le nom de l'exécutable. Il est alors très facile de parser les arguments définis par leur position. Par exemple, on pourrait récupérer le mode de jeu de la façon suivante :

```
#include<iostream>
#include<string>  // std::stoi in C++11
//#include<cstdlib> // use std::strtol in C++03

int main(int argc, char * argv[]) {
    if (argc>=2) {
        // convert from *char to int to use switch
        int mode = std::stoi(argv[1]);
        switch ( mode )
        {
            case 1:
            case 2:
                std::cout << "Game␣mode␣:␣" << mode << std::endl;
                break;
            default:
                std::cerr << "Unknown␣game␣mode." << std::endl;
                return 0;
        }
    } else {
        std::cerr << "Not␣enough␣arguments.␣Aborting..." << std::endl;
        return 0;
    }
    return 0;
}
```

Bien entendu, il faudrait traiter les exceptions éventuellement levées par `std::stoi`...