Projet Info IMAC1 2021

Principe :

Jeu Pokémon

Objectif :

Battre ou capturer les pokémon présents sur le plateau

Déroulement du jeu :

Phase de déplacement, puis phase de combat si on rencontre un pokémon

Merci beaucoup à Enguerrand pour sa relecture et ses conseils, allez voir son doc sur l'affichage il est très complet !

En noir le minimum, en vert l'optionnel.

Exemple de plateau

0	1	2	3	4	5	6	7	8	9	10
1										
2										
3			P							
4								T		
5										
6						X				
7			B							
8										
9										
10										

```
X : Joueur
P : Pikachu
T : Togepi
B : Bulbizarre
```

Version minimale

Exemple d'écran de combat (minimum)

```
Pikachu                                          Bulbizarre
PV : 100                                         PV : 63

Que souhaitez-vous faire ?
1 : Attaquer
2 : Capturer
```

Eléments minimums

`struct Joueur {`
`  position pos;`
`  inventaire inv;`
`}`
	Les pokémon se déplacent d'une case dans une direction aléatoire en même temps que le joueur.
	Si le joueur entre sur une case avec un pokémon un combat commence, avec les choix d'action (l'exemple le plus simple est d'attaquer ou essayer de capturer).
	Combat de pokémon « neutre », pas de faiblesse/force, valeur d'attaque et dégâts identiques.
	Le pokémon retrouve toute sa vie en fin de combat.

`struct Pokemon {`
`  ...`
`}`	

`struct position{`
`  ...`
`}`	

`Struct inventaire{`
`  ...`
`}`	

Version avancée

Exemple d'écrans successifs de combat (avancé):

1) Attends l'entree utilisateur :

```
Bulbizarre rencontré ! Quel pokemon envoyer ?
1 : Joueur.equipe[0]
2 : Joueur.equipe[1]
…
n : Joueur.equipe[n-1]
```

2) Attends l'entree utilisateur :

```
Pikachu                                          Bulbizarre
[Elec]                                           [Plante]
PV : 100                                         PV : 63

Que souhaitez-vous faire ?
1 : Attaquer
2 : Capturer
3 : Se soigner
4 : Fuir
```

3) Pendant 2 secondes :

```
Felicitations ! Vous avez battu/capture Bulbizarre
```

Eléments avancés

```
struct Joueur {
     position pos;
     Pokemon* equipe;
}
```

```
struct Pokemon {
     ...
     ...
}
```

```
struct position{
     ...
}
```

```
Struct inventaire{
     ...
     ...
}
```

Les pokémon se déplacent d'une case dans une direction aléatoire en même temps que le joueur.
Si le joueur entre sur une case avec un pokémon un combat commence, avec différents choix d'action dont attaquer, essayer de capturer, se soigner ou fuir.
Les types d'attaque peuvent former un triangle de force/faiblesse du genre eau→feu→plante.
Le pokémon ne regagne pas sa vie, il faut la gérer pendant le combat.

On peut ajouter au principe minimal :
- Une puissance variée pour les pokémon
- Un type qui peut changer l'efficacité du genre triangle d'efficacité (à ajouter dans les pokémon et dans les attaques)
- Choisir le pokémon avec lequel on attaque
- Une liste de `Pokemon` pour faire l'équipe du joueur
- Choisir le pokémon à envoyer au début du combat
- Des cailloux ou autres objets sur le terrain
- Plus de types
- Des potions/un centre de soin
- Pokémon légendaire ?

Annexe : Documentation utile

Conseil principal : FAITES DES DESSINS.

Enfin, si vous voulez. Et des fois on ne sait pas trop comment dessiner des pointeurs...
Les dessins aident à voir quels éléments on manipule dans notre code, par quelles étapes passent notre algorithme, quelles valeurs prennent les variables, et comment les fonctions interagissent entre elles. En particulier, faites des dessins de quels fichiers sont inclus dans lesquels, cela peut éviter beaucoup d'oublis, et si vous faites un makefile cela peut éviter de se mélanger les pinceaux (ou de les démêler plus vite en cas de confusion).

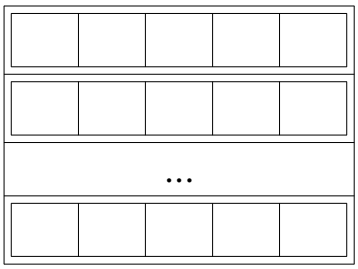
Tableau à double entrée :

Il est possible de réaliser un tableau à double entrée de plusieurs manières, un moyen est de déclarer des tableaux dans un tableau, en utilisant :

```
int nomDuTableau[nombreLignes][nombreColonnes];
```

On accède au contenu des cases en faisant `nomDuTableau[ligne][colonne]`.
Un petit schéma d'un tel tableau serait :

Ligne 0 contient un tableau →
Ligne 1 contient un tableau →
Ligne n contient un tableau →



Exemple :

Tableau 2 lignes, 5 colonnes

`tab[0][0]`	`tab[0][1]`	`tab[0][2]`	`tab[0][3]`	`tab[0][4]`
`tab[1][0]`	`tab[1][1]`	`tab[1][2]`	`tab[1][3]`	`tab[1][4]`

Évidemment c'est aussi utilisable avec des `char`, des `int*`, des `char*`, des `struct`, et autres idées plus ou moins utiles.

Un moyen autre est de représenter ça dans un tableau d'une seule ligne, en considérant que toutes les `n` cases on « change » de ligne.Dans ce cas pour un tableau dont les lignes possèdent `n` cases (autrement dit, c'est un tableau à `n` colonnes), au lieu d'utiliser `tab[l][c]`, il faut calculer soi-même, `tab[l][c]` devient donc `tab[l*n+c]`.

Exemple :

Tableau 2 lignes, 5 colonnes

Ligne 1					Ligne 2				
tab[0]	tab[1]	tab[2]	tab[3]	tab[4]	tab[5]	tab[6]	tab[7]	tab[8]	tab[9]

tab[0]	tab[1]	tab[2]	tab[3]	tab[4]
tab[5]	tab[6]	tab[7]	tab[8]	tab[9]

tab[0*5+0]	tab[0*5+1]	tab[0*5+2]	tab[0*5+3]	tab[0*5+4]
tab[1*5+0]	tab[1*5+1]	tab[1*5+2]	tab[1*5+3]	tab[1*5+4]

Il est aussi possible d'utiliser nos chers amis les pointeurs (que sont déjà les `tab[]`) en passant notamment par `malloc`, qui permet une allocation dynamique. La logique est la même qu'au dessus, seulement la syntaxe pour créer un tableau devient :

```
int* tab = (int*)malloc(sizeof(int)*NombreDeCases);
```

Dans le cas d'un tableau à double entrée, un seul `malloc` ne marche pas, il faut donc trouver un moyen de contourner le problème.

Bien que l'allocation avec `malloc` semble inutile « puisque le truc avec les crochets marche très bien », l'allocation dynamique est en fait prépondérante dans la programmation en général, savoir la manipuler est réellement utile pour ne pas avoir à le faire sur des projets plus compliqués. Ceci dit le sujet précise que la méthode est libre, il semblerait injuste de vous enlever des points si vous faites autrement.

Découpage du travail :

Si vous ne savez pas par où partir, séparez le projet en plusieurs morceaux, qui doivent chacun fonctionner avant de passer à la suite. Si vous bricolez votre code alors que vous ne le comprenez pas dès le début, le code spaghetti arrive rapidement et ça va souvent plus vite de tout refaire.

Il faut finir avec un jeu de plateau sur lequel vous pouvez déplacer votre joueur et capturer ou combattre des créatures (hélas) imaginaires. Si on analyse cet objectif on voit qu'il faudra commencer par les composants du jeu, les « ingrédients » de la recette :

- Un plateau, il faut donc réfléchir à sa représentation en C++, une représentation facile à manipuler, car le plateau est fondamental dans le projet. Il faudra aussi trouver un moyen d'afficher le contenu du plateau.
- Un joueur, qu'il faut pouvoir placer dans le plateau. Si on se rend compte que la représentation du plateau n'est pas du tout pratique on peut encore la changer. Il faut pouvoir le déplacer, comment ? (Attention, il ne doit pas pouvoir sortir du plateau !)
- Des pokémon, qui se déplaceront aussi dans un tableau, probablement le même que celui du joueur.

Ceci constitue les éléments de base, tous les composants nécessaires au jeu sont présents, il faut maintenant programmer le reste, découpable de la même manière.

Amélioration de l'expérience de jeu :

L'expérience de jeu repose sur la qualité du jeu lui-même, mais aussi sur la beauté de l'interface, et des options qu'il propose. Pour améliorer cette expérience, un affichage de début et de fin de partie est un élément non négligeable qui laisse de bons souvenirs. Des paramètres comme la taille du plateau à choisir au début de partie (amélioration proposée dans le sujet) sont aussi intéressants.

Bonnes pratiques et astuces :

Séparer les fonctions en modules est un excellent moyen de mieux se repérer dans son code. Travailler avec des fichiers courts permet de très vite retrouver une fonction spécifique, surtout si les modules sont intelligemment organisés. Par exemple séparer dans des fichiers .cpp respectifs (et .h associés) les fonctions et struct utilisés pour un type d'objet, par exemple un duo .cpp/.h pour le plateau, sa création et son affichage. Si vous avez plein de petits fichiers c'est bon signe, ensuite les makefile sont là pour aider la compilation.

Commentez vos fonctions, une courte description de l'action effectuée, des argument pris en entrée et de la sortie, juste après la définition de la fonction. Cela évite de (re)comprendre le code à chaque fois qu'on a fait de l'audiovisuel entre deux sessions.

Sans écrire que `cin` récupère une entrée utilisateur, vous pouvez noter ce que font les petits groupes de lignes dans vos fonctions, en plus de la description succincte cela accélère le débuggage au niveau logique, les problèmes de comportement non voulu.

Affichage en console :

Le document d'Enguerrand en parle beaucoup plus précisément et présente une aide à l'affichage qu'il a concoctée spécialement pour vous !

Il est possible d'afficher un tableau de manière « jolie » avec un code aussi simple :

```
for (int i=0; i<n; i++){
     cout << "| " << tab[i];
}
cout << "|" << endl;
```

A vous d'étendre le concept !

Pour finir, souvenez vous qu'un bon programmeur est surtout quelqu'un qui sait chercher sur Google (et StackOverflow).

Comme le dit Enguerrand, je reste avec votre disposition jours et nuits et pendant les vacances pour répondre à vos questions et vous aider si besoin !

Sources extérieures :

- https://www.cplusplus.com/doc/tutorial/ - Documentation C++

- https://github.com/dsmtE/ConsoleUtils – Exemples de la lib d’affichage d’Enguerrand

- https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/16595-lallocation-dynamique
(C’est en C, mais la théorie est la même, seule la syntaxe change avec les new)

- https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c
(Idem, du C mais tout y est sur les #include, la compilation par modules, les **pointeurs** et les struct)

- https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/
(La version 4 présentée sur le site est un peu tricky, Enguerrand vous conseille de ne pas l'utiliser)