Rapport Darkest Dungeon C

Yacine HAMADOUCHE

Liam GAVAU--PELLISSIER

Groupe TD 6 / TP 6

I/ Présentation du projet

L’objectif de ce projet était d’implémenter un jeu de stratégie tour par tour nommé "Darkest Dungeon" en C.  
   
Points clés du Projet :

* Gestion des listes chaînées : Utilisation de listes chaînées pour gérer les classes, personnages, accessoires et ennemis. Chaque élément est stocké dans une cellule faisant référence à une cellule suivante.
* Affichage : Fonctions d'affichage dans le terminal pour visualiser les éléments des listes, avec des formats spécifiques pour chaque type (classe, personnage, accessoire, ennemi).
* Combat : Les personnages et les ennemis s'affrontent dans des combats tour par tour. Les personnages peuvent attaquer, se défendre ou se restaurer. Les ennemis peuvent attaquer ou stresser les personnages.
* Gestion de la Santé et du Stress : Les personnages peuvent récupérer de la santé et réduire leur stress en utilisant la taverne et le sanitarium.
* Sauvegarde et Chargement : Sauvegarde et chargement de la progression à partir de fichiers CSV.
* Progression du Jeu : Le jeu progresse à travers une série de combats, avec des personnages gagnant des accessoires au fil des victoires.

Objectif du jeu :

* Gérer efficacement les ressources et les personnages pour survivre et vaincre les ennemis dans une série de combats jusqu’au boss final

Résultat du projet :

* Nous avons implémenté toutes les fonctionnalités de base proposée comme, les classes, les personnages, les ennemis, les accessoires, le sanitarium, la taverne, la roulotte, le déroulement d’un combat, et de plusieurs jusqu’à la victoire ou la défaite.

II/ Les structures mises en place

Nous avons fait le choix esthétique et pratique de séparer notre projet en plusieurs fichier, les structures sont dans un fichier nommé « Structures.h »  
Ce fichier contient les définitions des types énum et des structures utilisées pour représenter les classes, personnages, accessoires et ennemis dans le jeu.  
   
 Les structures définies sont :

* Statut : Énumération pour représenter l'état d'un personnage (attaquer, défendre, restaurer, mort, inutilisable).
* Accessoire : Structure représentant un accessoire avec ses attributs (nom, prix, bonus d'attaque, bonus de défense, bonus de PV, bonus de soin, réduction de stress).  
  Le principe est que l’accessoire soit relié à un personnage pour lui accorder des statistiques bonus.
* Classe : Structure représentant une classe de personnage avec ses attributs (nom, attaque, défense, PV max, restauration).
* Personnage : Structure représentant un personnage avec ses attributs (nom, classe, PV, stress, nombre de combats, accessoire, statut).  
  Les statistiques de base du personnage tel que ses points d’attaque, de défense, ses HP max etc... sont stockés dans sa classe.
* Ennemie : Structure représentant un ennemi avec ses attributs (nom, niveau, attaque, défense, PV, force d'attaque).
* Cellule : Structure représentant une cellule de liste chaînée pouvant contenir n'importe quel type de valeur (classe, personnage, accessoire, ennemi) et pointant vers la cellule suivante.  
  liste : Pointeur vers une cellule, représentant une liste chaînée.
* Type : Énumération pour identifier le type de chaque cellule (classe, personnage, accessoire, ennemi).

III/ Les fonctions créée

* int taille\_liste(liste lst) :   
  Calcule et renvoie la taille d’une liste chaînée.   
  Parcourt chaque cellule (nœud) de la liste et incrémente un compteur pour retourner le nombre total d’éléments.
* void afficher(liste lst) :  
  Affiche les éléments de la liste en fonction du type (classe, personnage, accessoire, ennemie). Selon le type détecté dans la cellule, le code utilise un format de tableau adapté pour afficher toutes les informations du type correspondant (par ex. pour un personnage : Nom, Classe, Attaque, etc.).
* void ajouter\_classe(liste \*lst\_classe, const char \*nom, int att, int def, int hpmax, int rest, type type) :  
  Alloue et initialise une structure de type classe, puis l’insère en tête de la liste lst\_classe. Sert à créer une classe avec ses attributs (attaque, défense, etc.).
* void ajouter\_personnage(liste \*lst\_personnage, liste \*lst\_classe, const char \*nom\_perso, const char \*nom\_classe, type type) :  
  Crée un nouveau personnage et lui associe une classe en recherchant cette dernière dans la liste lst\_classe. Si la classe est trouvée, le personnage est initialisé (HP, stress, statut, etc.) et inséré en tête de la liste lst\_personnage.   
  Gère aussi le cas où la classe n’est pas trouvée.
* void ajouter\_accessoire(liste \*lst\_accessoire, const char \*nom, int prix, int attbonus, int defbonus, int hpbonus, int heal\_bonus, int strred, type type) :  
  Crée un objet de type accessoire et l’insère en tête de la liste des accessoires. Chaque accessoire a un prix et divers bonus (attaque, défense, etc.).
* void ajouter\_ennemie(liste \*lst\_ennemie, const char \*nom, int niveau, int att, int def, int hp, int attstr, type type) :  
  Crée et initialise un ennemi (niveau, attaque, défense, etc.) avant de l’insérer en tête de la liste lst\_ennemie.
* void ajouter\_cellule(liste \*liste, cellule \*c) :  
  Ajoute une cellule déjà existante à la fin d’une liste. Parcourt la liste jusqu’à la dernière cellule, puis insère la nouvelle cellule à la suite.
* Cellule \*supprimer\_num(liste \*liste, int num) :  
  Supprime la cellule d’indice num dans la liste (0 pour la tête, 1 pour la cellule suivante, etc.) et renvoie un pointeur vers cette cellule détachée. Gère les cas d’indice hors limites et de liste vide. Le pointeur retourné permet d’effectuer des switch entre liste.
* int fin\_combat(liste lst\_personnage\_actif, liste lst\_ennemie\_actif) :  
  Vérifie si le combat est terminé en testant :   
  S’il n’y a plus de personnages actifs (perdu) renvoie 1.   
  S’il n’y a plus d’ennemis actifs (gagné) renvoie 2.   
  Sinon (combat toujours en cours) renvoie 0.
* void attaque\_personnage(personnage \*perso, ennemie \*ennemi) :  
  Calcule les dégâts infligés à un ennemi par un personnage (attaque de base + bonus d’accessoire - défense ennemie).   
  Met à jour les HP de l’ennemi et affiche le résultat de l’action.
* void attaque\_ennemie(personnage \*perso, ennemie \*ennemi, int type) :  
  Permet à l’ennemi d’attaquer ou de stresser un personnage en fonction de la valeur de type.   
  Si l’ennemi stresse, modifie le stress du personnage.   
  Si le personnage est trop stressé (≥100), il devient inutilisable.   
  Si l’ennemi attaque, calcule les dégâts et met à jour les HP du personnage. Marque le personnage comme mort s’il n’a plus de points de vie.
* void restaurer(personnage \*perso) :  
  Regénère une partie des points de vie du personnage en combinant la rest de la classe et le bonus de soins apporté par l’accessoire éventuel.  
  Les HP du personnage sont alors mis à jour sans dépasser le maximum.
* void taverne(liste taverne) :  
  Réduit de 25 le stress de chaque personnage se trouvant dans la liste taverne.   
  Si le stress devient négatif, il est remis à 0.   
  Le personnage est remis en état d’attaque (réinitialisation du statut).
* void sanitarium(liste sanitarium) :   
  Soigne chaque personnage en lui rajoutant 7 HP, sans dépasser son maximum de vie.
* void lire\_csv\_accessoire(liste \*lst\_accessoire),   
  void lire\_csv\_classe(liste \*lst\_classe),   
  void lire\_csv\_ennemi(liste \*lst\_ennemi),   
  void lire\_csv\_personnage(liste \*lst\_personnage, liste \*lst\_classe) :  
  Lisent des fichiers CSV (respectivement pour les accessoires, les classes, les ennemis et les personnages) et chargent les données dans les listes associées en appelant les fonctions d’ajout.  
  Permettent d’initialiser le jeu avec des données préconfigurées.
* const char \*statut\_to\_string(statut statut)   
  et statut string\_to\_statut(const char \*statut) :  
  Font la conversion entre un statut (ex. ATTAQUER, DÉFENDRE, etc.) et sa représentation en texte.   
  Sont utilisées pour l’enregistrement et le chargement depuis un fichier, afin de stocker le statut sous forme de chaîne de caractères.
* void save(...) :   
  Sauvegarde l’état global du jeu dans un fichier CSV.   
  Parcourt les listes (personnages, accessoires, etc.) et écrit les attributs de chaque élément dans le fichier.   
  Sauvegarde également des informations comme la quantité d’or, le numéro du combat, et la limite du nombre de personnages.
* Et void chargement(…) :  
  Charge la sauvegarde depuis un fichier CSV.  
  Parcourt les sections (PERSONNAGE, ACCESSOIRE\_ACQUIS, etc.), puis recrée les éléments dans les listes appropriées. Gère la conversion du statut sous forme de chaîne vers le type statut.

IV/ Les étapes importantes du main()

* Initialisation des listes et chargement des données via CSV :  
  Le programme commence par déclarer et initialiser toutes les listes nécessaires (liste de classes, de personnages, d’ennemis, d’accessoires, etc.)  
  Les fichiers CSV (classe.csv, accessoire.csv, ennemi.csv, personnage.csv) sont ensuite lus pour charger les données de base dans chaque liste correspondante (fonctions lire\_csv\_xxx)  
  Le programme demande à l’utilisateur s’il souhaite charger une sauvegarde existante  
  En cas de réponse positive, il demande un nom de fichier et tente de charger les données (personnages, accessoires, or, numéro de combat, etc.) via la fonction chargement().   
  Si la sauvegarde n’est pas chargée, le joueur a l’option de créer une nouvelle sauvegarde avant de commencer la partie.
* Boucle de combats (jusqu’à 10) :   
  Une boucle while (numéro\_combat < 10) représente une suite de combats successifs  
  À chaque nouveau combat : Si le numéro de combat est égal à 5 le nombre max de personnage actifs augmente.  
  Le joueur sélectionne quels personnages vont participer au combat (jusqu’à perso\_max)  
  Un ennemi est choisi (tiré de la liste des ennemis par niveau) et placé dans la liste d’ennemis actifs.   
  Le joueur peut acheter des accessoires dans la “roulotte” (liste des accessoires disponibles) et les équiper à ses personnages actifs.  
  Déroulement d’un combat :  
  Tant que la condition de fin de combat n’est pas atteinte (vérifiée par fin\_combat()), on alterne :   
  Le tour des personnages (chacun choisit son action : attaquer, se défendre ou se restaurer).   
  Le tour de l’ennemi (attaque ou inflige du stress aléatoirement à un des personnages). Si tous les personnages actifs sont éliminés, la partie est perdue.   
  Si tous les ennemis actifs sont éliminés, la victoire est déclarée.
* Fin de combat et récompenses :  
  Lorsque l’ennemi est vaincu, le joueur reçoit de l’or (10 pièces) et gagne un accessoire aléatoire depuis la roulotte. Les personnages actifs sont replacés dans la liste globale des personnages, leur nombre de combats (NBcombat) est incrémenté et leurs éventuels accessoires sont remis dans la liste des accessoires acquis.   
    
  Gestion des Sanitarium et Taverne :  
  Après chaque combat, la fonction sanitarium() soigne automatiquement les personnages qui y séjournent, et la fonction taverne() réduit le stress de ceux qui s’y reposent.   
  Le joueur peut décider de faire sortir un personnage du Sanitarium ou de la Taverne, et d’en faire entrer de nouveaux, dans la limite des places disponibles.   
    
  Incrémentation du numéro de combat et poursuite jusqu’à la fin :  
   Le compteur numero\_combat est augmenté de 1 à chaque victoire, jusqu’à atteindre 10. Lorsque numero\_combat atteint 10, le jeu s’arrête, et le programme affiche un récapitulatif (liste des personnages toujours présents, or total, accessoire acquis, etc.).

V/ Les difficultés auxquelles vous avez été confronté  
  
Nous avons rencontrés 4 difficulté majeures spécifique :

1. La première était vis a vis de la sauvegarde du statut, nous voulions sauvegarder directement le type statut (ATTAQUER, DEFENDRE, etc.) dans le fichier .csv, mais cela n’a pas fonctionné pour des raisons qu’on n’a pas préférer approfondir.   
   Il a donc fallu créer deux fonctions spécifiques pour convertir un statut en chaîne de caractères et inversement, à savoir statut\_to\_string(statut) et string\_to\_statut(const char \*) qui n’ont pas forcément été techniques à implémenter mais qui ont requis du travail supplémentaire. Ainsi, lors de la sauvegarde, nous pouvons écrire la valeur sous forme de texte dans le .csv, et lors du chargement, la retransformer en type statut.
2. La deuxième difficulté a été rencontré lors de la création initiale du projet, lorsque nous avons dû créer toutes les structures, nous avons du bien réfléchir et relire plusieurs fois le sujet pour avoir la meilleure disposition de structure pour avoir des fonctions génériques.   
   Afin d’avoir un code lisible, mais pour autant des fonctions simples nécessitant pas plusieurs dizaines de ligne.  
   Surtout qu’au début nous avions fait le choix de mettre l’attribut « suivant » directement dans la structure (ennemie, personnage etc..). Le choix complexifiait par forcément le code, mais le rendais extrêmement lourd et nous forçait à faire une fonction par structure pour chaque action (suppression, ajout, etc..).  
   On a finalement opté pour le format actuel, après nous être rendu compte des bénéfices que ce dernier apportait vis a vis de la lisibilité du code et de la taille.
3. La troisième mais pas des moindres, était lié à une décision qu’on a prise, celle de traiter toutes les erreurs d’entrées potentielles du joueur, et d’éviter tout crash du programme.   
   Mais cela nous a demandé beaucoup de test, d’ajustement et de temps.
4. La quatrième était un peu plus syntaxique et compréhensible partiellement, en effet, à chaque fois que nous souhaitions accéder aux champs spécifiques d’une cellule (par exemple, le prix d’un accessoire), il fallait préalablement connaître précisément le type stocké et faire le « cast » adéquat, comme ((accessoire \*)tmp->valeur)->prix.   
   Cela signifie que la variable tmp->valeur est convertie (cast) en un pointeur de type accessoire \*.   
   En d’autres termes, on dit au compilateur : « Oh ! Interprète la zone mémoire à laquelle tmp->valeur pointe comme si elle était un objet de type accessoire. »   
   Une fois convertie, on peut accéder aux champs de la structure accessoire, par exemple prix.   
   Sans ce cast, le compilateur ne saurait pas qu’il doit traiter tmp->valeur comme un accessoire, et ne permettrait donc pas l’accès à la variable prix.  
   Cette obligation de « deviner » ou d’anticiper le type de la valeur avant d’y accéder nous as fait perdre pas mal de temps puisque nous ne comprenions pas d’où venait l’erreur initialement, en effet nous n’avions jamais utilisé les listes chaînées comme cela.

VI/ Une réflexion sur le thème de l'utilisation de listes chaînées

1. Est-ce qu'il y aurait une solution plus simple ?  
     
   Non nous ne pensons pas qu’il y aurait une solution plus simple, probablement une différente comme d’utiliser des tableaux pour stocker les objets des structures, cette solution ne serait pas forcément plus simple ni plus dure simplement différentes puisqu’elle nous forcera à aborder la gestion de la mémoire d’un nouvel angle avec de nouvelles perspectives
2. Est-ce que les listes chaînées offrent tout de même des avantages ici ?  
     
   Dans l’architecture actuelle du code, les listes chaînées offrent plusieurs avantages certains, elle nous permettent tout d’abord, d’ajouter ou de supprimer des éléments facilement, sans avoir besoin de déplacer les données comme avec un tableau,   
   elles permettent aussi de mieux gérer la mémoire, car chaque élément est alloué individuellement, ce qui évite d'avoir des espaces inutilisés ou de devoir trouver un grand bloc de mémoire libre.  
   Elles sont aussi très flexibles, car leur taille peut s’adapter en fonction des besoins, même si on ne sait pas à l’avance combien d’éléments seront nécessaires.   
   Et elles permettent, d’avoir des opérations d’insertion et de suppression très rapides et potentiellement plus simple, contrairement aux tableaux où ces opérations peuvent prendre plus de temps.

VII/ Lancer le jeu

Après la compilation du fichier main.c, le programme guide très bien à travers différentes étapes du jeu. Il suffit d'interagir avec le terminal en suivant les instructions affichées à l'écran. Le programme demande des choix à l'utilisateur, tels que sélectionner des personnages, acheter des accessoires, et décider des actions à entreprendre pendant les combats. Les messages et les menus sont clairs et permettent une navigation facile tout au long du jeu.