R1.01 Initiation au développement – SAÉ2

5 janvier 2025
Erwan Kerrien

World of IUT, une aventure textuelle

Rappel : si vous avez des questions sur cette SAÉ ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr

1 Consignes pour le rendu

Cette SAÉ est à faire par binômes. Elle doit prendre la forme d'un dépôt git sur gitlab.univ-lorraine.fr. **Vous devez inviter M. Kerrien comme administrateur : identifiant `@kerrien5`.**

Le dépôt contiendra le code source, ainsi qu'un rapport, et d'éventuels fichiers de test. L'état du dépôt le 5 janvier 2025 à 23h59 sera noté pour évaluer cette SAÉ. Aucun push ultérieur ne sera pris en compte.

- Le code source devra être correctement commenté.
- Vous fournirez un fichier `Makefile` pour compiler votre projet [1]. Un fichier `Makefile` de départ vous est fourni dans les ressources de la SAÉ, et vous aurez juste à le mettre à jour en fonction des fichiers que vous ajouterez. Pour quelques explications sur comment fonctionne la programmation modulaire et le rôle du `Makefile`, vous pouvez lire l'annexe en fin de document.
- Le rapport sera rendu au format pdf. Il fera au minimum 2 pages, mais n'excédera pas 5 pages. Il devra discuter et justifier du choix des structures de données utilisées. Il devra par ailleurs discuter des fonctionnalités et limitations du programme. Vous pourrez pour cela vous baser sur un ensemble de tests qu'il faudra alors décrire. Les fichiers nécessaires aux tests seront ajoutés au dépôt.
- Le rapport devra indiquer précisément comment le travail a été partagé, et clairement identifier la part de chacun dans chaque tâche.
- Pour celles et ceux qui aborderont la partie avancée, le fichier de description d'un jeu devra avoir un nom avec pour extension `.txt`.

Voici la liste exhaustive des fichiers attendus a minima :

- `Makefile`
- `WorldOfIUT.c`
- `game.c` et `game.h`
- `cmd.c` et `cmd.h`
- `exits.c` et `exits.h`
- `mobile.c` et `mobile.h`
- `location.c` et `location.h`
- `stack.c` et `stack.h`
- `rapport.pdf`

En option, vous pourrez avoir :

- un fichier `README.md` qui agrémente la page gitlab
- les fichiers `object.c` et `object.h`
- éventuellement les fichiers `stackobject.c` et `stackobject.h`
- et/ou des fichiers de description du jeu avec l'extension `.txt`
- et/ou des fichiers `test<x>.c` qui sont des versions de test de `WorldOfIUT.c`

Très important : le dépôt ne devra pas contenir de fichier exécutable, ni de fichier objet (`*.o`). Les seules extensions autorisées pour les fichiers sont donc : `.c, .h, .pdf, .txt`, auxquelles on ajoute le fichier `Makefile` et le fichier `README.md`. **Tout manquement à cette règle sera sanctionné.** [2]

1. ce fichier est exploité par la commande `make` qui n'est pas disponible par défaut mais s'installe au besoin par un simple `sudo apt install make` dans un terminal
2. Si vous voulez effacer tous les fichiers objets et executables générés par la compilation, il suffit de taper `make clean` (sous Linux) pour vous en débarasser.

2 Objectifs de la SAÉ

Vous devez créer le jeu *World of IUT*, une version très basique d'un jeu d'aventure textuelle. Ces jeux étaient très populaires dans les années 80 et 90 et sont les ancêtres des MMORPG actuels. L'un des tout premiers jeux d'aventure textuelle est The Colossal Cave Adventure (vous pouvez y jouer si vous voulez, mais en-dehors des cours !).

De nombreuses améliorations ont successivement été proposées, jusqu'à en faire des jeux en réseaux réunissant de nombreux joueurs, comme les MUD (*Multi-User Dungeon*) dont FranDUM [3]. La base cependant est la même : tout se fait de manière textuelle. Le joueur utilise des commandes pour interagir avec le monde. Les actions sont décrites par du texte, ainsi que les lieux, objets ou autres monstres rencontrés. Ces jeux ne nécessitent donc pas d'interface graphique et tout se déroule dans la fenêtre de commande.

Pour que cela fonctionne, le monde est découpé en cases dans lesquelles le joueur ou la joueuse peut se déplacer selon des directions prédéfinies. Les cases ont chacune un nom (description courte) et une description plus longue. L'objectif principal de la SAÉ est de réussir à avoir ce système de cases pour créer un espace que le joueur va pouvoir explorer. Atteindre cet objectif assurera déjà une note suffisante pour valider la SAÉ.

Deux améliorations seront proposées pour celles et ceux qui veulent aller plus loin :

— étendre le jeu pour que des objets puissent être placés dans les cases. Ils pourront alors être ramassés ou déposés par le joueur ou la joueuse (notion d'inventaire).
— pouvoir charger les cases (et objets) du jeu depuis un fichier de description. Plusieurs mondes pourront alors être proposés sans avoir à recompiler le jeu.

Ajouter des monstres serait plus compliqué car il faudrait introduire la notion de temps (pour le déplacement des monstres ou encore pouvoir créer un système de combat). Aussi, nous nous contenterons de créer l'espace et éventuellement les objets, ce qui devrait déjà vous donner une bonne idée de la manière dont ces jeux sont programmés.

2.1 Monde minimal

Je demande un monde minimal décrit dans la figure 1. Les cases 11 et 12 sont bien respectivement dans les directions `UP` et `DOWN` par rapport à la case 5. Vous mettrez les descriptions que vous voulez, à condition qu'elles donnent les bonnes indications au joueur pour s'orienter.

3 Première étape : comprendre le matériau fourni

Pour commencer ce projet, récupérez le fichier d'archive fourni et décompressez-le dans un dossier. Vérifiez que vous avez les fichiers suivants :

— `Makefile`
— `WorldOfIUT.c`
— `game.c` et `game.h`
— `cmd.c` et `cmd.h`
— `exits.c` et `exits.h`
— `mobile.c` et `mobile.h`

3.1 Compilation et lancement du jeu

Ouvrez Visual Studio (ou VSCodium), et ouvrez le dossier contenant ces fichiers (`File` puis `Open folder...`).

Le fichier **Makefile** sert à compiler le projet. Ouvrez un terminal et tapez la commande `make` sous Linux.

Cette commande compile le projet. Vérifiez que cela a fonctionné en lançant le jeu par la commande : `./WorldOfIUT`

Les commandes sont toutes fonctionnelles, sauf `look` et `go` qui sont dans une version très basique (charge à vous de les terminer !). Vous pouvez les tester et les tapant après l'invite de commande (`->`). En voici la liste (qui reprend l'aide) :

— `help` : affiche l'aide
— `quit` : permet de quitter le jeu et de revenir au shell

3. entièrement écrit en C et dont j'ai eu la chance d'être administrateur pendant un an pendant mes études. Il est aujourd'hui tombé en désuétude et ne fonctionne plus...

— `look` : affiche une description de l'endroit, objet, personnage, direction passé en argument. Pour le moment, seule `look me` fonctionne. `look around` ou tout autre argument `look <arg>`, vous indique que vous êtes dans le vide et vous incite à coder les cases du jeu.
— `go` : permet de se déplacer dans la direction indiquée. Pour le moment, si l'argument `<arg>` est une direction valide, le message `You go <arg>` s'affiche. Sinon, cela affiche `You try to go <arg>, but you cannot go <arg> from nowhere. You (God) should create some locations...`, qui vous incite à nouveau travail de création du monde.

Testez les commandes suivantes : `help`, `look`, `look blabla`, `look me`, `go`, `go blabla`, `go south` **et** `quit`

3.2 Analyse des fichiers

Hormis le fichier `Makefile`, il y a deux sortes de fichiers. Un fichier de programme principal (`WorldOfIUT.c`) et des couples de fichiers avec les extensions .h et .c qui définissent chacun un module. Les modules sont `game`, `exits`, `cmd`, et `mobile`.

3.2.1 Le programme principal : `WorldOfIUT.c`

WorldOfIUT.c est le fichier du programme principal. C'est dans ce fichier qu'est définie la fonction `main` qui exécute le programme. Vous pouvez voir qu'il est simple : un message de bienvenue s'affiche (fonction `Intro`, puis une structure `Game` est initialisée (voir le module `game`), puis une boucle infinie (`while (1)`) est lancée qui appelle la fonction `processCommand` définie dans le module `cmd` : cette fonction affiche l'invite de commande, lit la commande de l'utilisateur et la traite. En particulier, si l'utilisateur entre la commande `quit`, le programme se terminera, sortant ainsi de la boucle infinie.

3.2.2 Définition d'un module

Un module est composé de deux fichiers de même nom, si ce n'est une extension différente. L'extension .h contient les déclarations. Ces déclarations doivent permettre au compilateur de vérifier que l'utilisation des structures ou fonctions ou autres constantes par une autre module ou fichier est correcte. On trouvera donc la définition des structures ainsi que la déclaration des fonctions par leur prototype : on ne donne que le type de retour, le nom de la fonction, ainsi que le type de chacun de ses paramètres (donc leur nombre).

3.2.3 Mobule `game`

Le module `game` définit la structure `Game` et les fonctions disponibles pour la manipuler depuis l'extérieur. Ceci est décrit dans le fichier `game.h`. Cette structure a pour vocation de rassembler les références vers tous les objets (au sens large : joueur, cases, objets éventuels) du jeu. Pour l'instant elle ne contient donc qu'un pointeur vers le joueur qui est un `Mobile` (voir le module `mobile`). On trouve ensuite la fonction `GameInit` qui crée la structure (en allouant la mémoire nécessaire, d'où le type pointeur renvoyé), et la fonction `GameShutdown` qui la détruit, en réalisant toutes les libérations de mémoire nécessaires. Le fichier `game.c` contient la définition de ces fonctions.

3.2.4 Module `mobile`

Le module `mobile` définit la structure `Mobile` et les fonctions disponibles pour la manipuler. Cette structure décrit un personnage et ne sera utilisée que pour le joueur. La structure `Game` contient donc une référence vers ce mobile. Un mobile est décrit par un nom (chaîne de caractères `name`) ainsi qu'une description plus longue (chaîne de caractères `desc`). La fonction `MobileNew` permet de créer une telle structure (en utilisant la fonction standard `strdup` pour à la fois allouer et recopier une chaîne de caractères) et la fonction `MobileDelete` permet de libérer la mémoire associée à cette structure (notamment appel à `free` pour libérer la mémoire allouée via `strdup`). Enfin la fonction `MobilePrint` affiche une description d'un objet de ce type, avec une première ligne qui contient le nom, suivi sur les autres lignes, de sa description. **Vous devrez modifier cette structure pour indiquer une case pour le mobile, et écrire une fonction permettant de bouger le mobile.**

3.2.5 Module `exits`

Le module `exits` définit le type `Direction` et les fonctions associées (voir le fichier `exits.h`). On utilise ici une nouveau mot-clé `enum` qui permet de définir un type par l'ensemble des valeurs acceptées. Le langage C utilise de manière

sous-jacente le type `int` et la notation employée permet de spécifier que la valeur `WRONGDIR` de type `Direction` sera traduite par -1 si on souhaite l'utiliser comme entier, puis `NORTH` vaudra 0 et implicitement on incrémente de 1 pour les valeurs suivantes (donc `EAST`=1, `SOUTH`=2, `WEST`=3, `UP`=4 et `DOWN`=5). Ainsi qu'indiqué dans le commentaire, on peut ainsi réaliser une boucle sur toutes les directions en faisant un `for(int i=NORTH; i<=DOWN; i++)...`. Le module met ensuite à disposition deux fonctions permettant de transformer une chaîne de caractères en valeur de type `Direction` (fonction `strtodir`) et inversement traduire une valeur de `Direction` en chaîne de caractères (fonction `dirtostr`). Ces fonctions sont définies dans le fichier `exits.c`.

3.2.6 Module `cmd`

Le module `cmd` gère l'interaction utilisateur via les commandes qu'il ou elle rentre. On voit dans le fichier `cmd.h` qu'il ne rend disponible qu'une seule fonction `processCommand` qui prend en paramètre les informations sur le jeu (variable de type `Game`). Cette fonction affiche l'invite de commande, lit une commande sur l'entrée standard et la traite. Si vous allez regarder le fichier `cmd.c`, vous verrez que ce traitement est fait par plusieurs fonctions :

- la fonction `getInput` affiche l'invite de commande puis lit une ligne entrée par l'utilisateur grâce à la fonction `getline` (non disponible sous Windows)
- puis cette ligne est traitée par la fonction `parseAndExecute` qui extrait le premier mot de la commande (via la fonction `strtok` de la bibliothèque standard), et en fonction de ce mot va appeler la fonction correspondant à la commande, et à défaut afficher un message d'erreur.
- si la commande est "help", alors la fonction `cmdHelp` est appelée : celle-ci affiche simplement un message d'aide avec `printf`
- si la commande est "quit", alors la fonction `cmdQuit` est appelée, avec le jeu en paramètre : celle-ci libère proprement la mémoire occupée par le jeu (via `GameShutdown`) puis quitte le programme en appelant `exit`. Le 0 passé en paramètre est la valeur renvoyée par l'exécutable au shell et indique que cette fin de programme est normale.
- si la commande est "look", alors la fonction `cmdLook` est appelée avec en paramètre le jeu ainsi que le deuxième mot sur la ligne de commande (aussi extrait via `strtok` et stocké dans la variable `args`). Cette fonction vérifie que la commande a été correctement émise et ne réagit correctement pour l'instant que si le second mot est "me", auquel cas elle affiche les informations sur le joueur en appelant la fonction `MobilePrint` avec en paramètre le joueur stocké dans la structure `Game` (voir le module `mobile`). Sinon, elle affiche un message standard. **Vous devrez enrichir cette fonction** `cmdLook`
- si la commande est "go", alors la fonction `cmdGo` est appelée avec le jeu en paramètre ainsi que le deuxième mot sur la ligne de commande (voir "look" ci-dessus pour voir comment il est extrait). Cette fonction fait un peu comme `cmdLook` pour le moment et vérifie que la syntaxe est correcte puis regarde si le second mot est une direction valide (via la fonction `strtodir` qui renvoie dans ce cas autre chose que `WRONGDIR`), auquel cas elle affiche "You try to go <dir>, but you cannot go <dir> from nowhere. You (God) should create some locations...", sinon, elle affiche un message d'erreur. **Vous devrez enrichir et compléter cette commande.**

Ces commandes "look" et "go" sont la raison pour laquelle l'anglais est la langue utilisée dans le jeu : la syntaxe anglaise permet de simplifier grandement la gestion des messages face à la diversité possible des arguments (par exemple, pas de genre...)

3.3 Récapitulatif et précisions des objectifs

Aucun lieu n'est défini pour le moment, l'objectif est de rajouter un espace que le joueur ou la joueuse peut explorer (commande "look") et dans lequel il ou elle peut se déplacer (commande "go"). Vous devrez donc, comme objectif principal :

- avoir défini une structure `Location` qui permet de stocker une case de l'espace. Cette structure sera associée à des fonctions de création `LocationNew` et de destruction `LocationDelete`, ainsi qu'à des fonctions d'affichage `LocationPrint` ainsi que plus tard `LocationPrintShort`.
- S'y ajoutera une fonction de création de toutes les cases du jeu `LocationInit` couplée à une fonction de destruction de toutes ces cases `LocationDestroy`. Ces fonctions devront gérer intelligemment la mémoire, ce qui nécessitera une structure de pile.
- Tout ceci sera défini dans un nouveau module `location` et donc deux fichiers `location.h` et `location.c`, ainsi qu'uu nouveau module `stack` (et donc les fichiers `stack.c` et `stack.h`) que vous devrez écrire.
- modifier la structure `Mobile` pour y ajouter un pointeur vers une `Location` et ainsi placer le joueur dans le jeu. Une nouvelle fonction `MobileMove` sera écrite pour placer le joueur dans un nouvel endroit.

— modifier les fonctions `cmdLook` et `cmdGo` afin de pouvoir bénéficier des commandes "look around" (regarder l'endroit où est couramment le joueur), "look <dir>" (pour regarder quel endroit se trouve dans la direction <dir> indiquée, et "go <dir>" pour déplacer le joueur dans la direction indiquée.
— modifier la structure `Game` afin que la mémoire soit correctement gérée (allocation et libération).

4 Deuxième étape : le module `Location`

4.1 Création du module

En prenant exemple sur le module `mobile`, créez un nouveau module `location` qui définit la structure `Location`, similaire à la structure `Mobile` et qui contiendra aussi un champ `name` pour son nom et un champ `desc` pour sa description, tous deux des chaînes de caractères. Associez-lui

— une fonction `LocationNew`, similaire à `MobileNew` qui alloue une nouvelle structure et en renvoie le pointeur
— une fonction `LocationDelete`, similaire à `MobileDelete`, et qui libère la mémoire allouée par la fonction précédente
— une fonction `LocationPrint`, similaire à `MobilePrint` qui affiche le nom de la `Location` sur une première ligne, puis sa description sur les lignes suivantes.

Ces diverses déclaractions et définitions seront faites dans deux fichiers `location.h` et `location.c`

4.2 Intégration au projet

4.2.1 Impact sur le `Makefile`

Modifiez le `Makefile` pour que le nouveau module `location` soit compilé avec le projet. Dans ce qui suit, modifiez à chaque fois le `Makefile` de façon à indiquer la dépendance éventuelle d'un module au fichier `Location.h`.

4.2.2 Impact sur le `Mobile`

Modifiez la structure `Mobile` pour y ajouter une référence vers sa localisation stockée dans une `Location`. Ajoutez au module une fonction `MobileMove` qui prendra en paramètre un pointeur vers un `Mobile`, ainsi qu'un pointeur vers une `Location` et stockera ce dernier dans le nouveau champ.

4.2.3 Impact sur le `Game`

Modifiez la fonction `GameInit` pour y ajouter la création d'une `Location` et son stockage dans le `Mobile` du joueur. On pourra utiliser la fonction `MobileMove` pour réaliser ce lien. On pourra utiliser par exemple "On the road" comme nom à stocker dans la `Location` et "The road continues north and south. You can see a house on the west." comme description.

4.2.4 Impact sur la commande "look"

Modifiez la fonction `cmdLook` afin d'afficher les informations sur la localisation courante du joueur si le deuxième mot est "around". On pourra utiliser la fonction `LocationPrint`.

5 Troisième étape : créer et connecter les `Location`

5.1 Mise à jour du module `location`

Modifiez la structure `Location` afin de pouvoir y stocker des références vers des `Location` situées dans chaque direction valide. On pourra utiliser un tableau de pointeurs (un pointeur par direction). L'absence d'issue dans une direction sera indiquée par le pointeur `NULL`.

Ajoutez une fonction `LocationInit` qui crée toutes les cases du monde et établit les connections entre elles. Les noms et descriptions seront entrés en dur à ce stade (voir la description du monde minimal de la figure 1). La fonction renverra la première case créée. Pour plus de lisibilité, on pourra écrire une fonction (non exposée à l'extérieur dans le .h) `LocationSetExit` qui prend en paramètre un pointeur vers une `Location`, une `Direction` et un autre pointeur vers une `Location` et qui stocke ce dernier pointeur comme issue dans la direction indiquée pour la première `Location`.

5.2 Intégration du projet

5.2.1 Impact sur le `Game`

Modifiez la fonction `GameInit` pour ne plus créér une seule `Location` mais toutes les `Location`s par un appel à `LocationInit`. Déplacez le joueur dans la bonne case, dans cette fonction.

5.2.2 Impact sur la commande "go"

Modifiez la fonction `cmdGo` pour déplacer le joueur dans la direction indiquée si celle-ci est valide et si la case actuelle a bien une issue dans cette direction. Sinon, affichez un message approprié.

5.2.3 Impact sur la commande "look"

Modifiez la fonction `cmdLook` pour que le nom de la case attenante s'affiche quand on indique une direction valide, et correspondant à une issue, en deuxième mot de la commande.

6 Quatrième étape : une gestion correcte de la mémoire

6.1 Analyse du problème

6.1.1 Description du problème

Il faut pouvoir libérer la mémoire allouée pour les différentes `Locations`. On ne peut cependant pas utiliser la même stratégie que pour une liste chaînée car il peut y avoir des cycles dans le monde. Si vous regardez le monde minimal de la figure 1, vous voyez le cycle : 5-6-7-8-9-10-5...

En procédant comme une liste chaînée, on libérerait la mémoire de la case 5, puis celle de ses voisines, dont 6, puis on libérerait successivement 7, 8, 9, 10, ... puis 5 qui est déjà libérée, ce qui provoquerait un `segmentation fault`.

6.1.2 Une solution : une pile de stockage

Cette situation est délicate à gérer. Mais on peut s'en sortir avec une structure de pile : à chaque fois qu'on crée une `Location`, on pousse son pointeur sur une pile. Dans ce cas, tous les pointeurs alloués seront dans cette pile, une et une seule fois, et il suffira donc de libérer successivement chaque pointeur de cette pile, sans se soucier des `Location`s voisines. Par exemple, dans notre pile, on aura le pointeur vers 10, puis 9, puis 8, puis 7, 6, et 5. On libère 10, mais sans appel récursif de libération sur les voisines, puis on fait de même avec 9, 8,7,6 et 5 : à la fin toutes les `Location`s ont bien été libérées une et une seule fois.

6.1.3 Création d'un module `stack`

En vous inspirant du TP sur les listes chaînées ou du TP sur les piles, créez un module `stack` (fichiers `stack.h` et `stack.c`) qui définit un type `Stack` implémentant une pile. Chaque cellule de la pile stockera un pointeur vers une `Location`, ainsi qu'un pointeur vers la cellule suivante. Vous définirez :

- la constante `empty_stack`.
- la fonction `StackIsEmpty` qui détermine si un `Stack` est `empty_stack`
- la fonction `StackPush` qui insère un pointeur sur une `Location` sur la pile, et renvoie la nouvelle pile. Ce pointeur est le résultat d'un appel à `LocationNew`.
- la fonction `StackHead` qui renvoie le pointeur vers la `Location` stockée sur la pile. Pour détruire cette `Location`, on pourra appeler `LocationDelete` sur ce pointeur.
- la fonction `StackPop` qui détruit la première cellule de la pile et renvoie le reste de la pile. Cette fonction ne fait pas appel à `LocationDelete` (faire un appel à `StackHead` avant)

6.2 Intégration du projet

6.2.1 Impact sur le `Makefile`

Modifiez le `Makefile` pour que le nouveau module `stack` soit compilé, et reportez les dépendances des différents fichiers C au nouveau fichier `stack.h` au fur et à mesure des modifications.

6.2.2 Impact sur la `Location`

Modifiez la fonction `LocationInit` pour qu'elle stocke chaque `Location` créée dans un `Stack`. Ce `Stack` sera renvoyé par la fonction.

Ajoutez une fonction `LocationDestroy` qui prend en paramètre un `Stack` et détruit proprement chaque `Location` qui est stockée avant de détuire chaque cellule de la pile. Cette fonction renverra `empty_stack`. L'algorithme général sera

```
Stack tmps=s
Tant que StackIsEmpty(tmps) == Faux faire
|   LocationDestroy(StackHead(tmps))
|   tmps <- StackPop(tmps)
FinTantQue
```

6.2.3 Impact sur le `Game`

Modifiez la structure `Game` pour y ajouter un champ de type `Stack` qui recueillera la liste des `Location`s du jeu. Modifiez la fonction `GameInit` pour que le résultat de l'appel à `LocationInit` soit stocké dans ce champ de type `Stack`. Déplacez le joueur dans la bonne case, dans cette fonction, en tenant compte de ce nouveau champ.

Modifiez la fonction `GameShutdown` afin de détruire proprement le `Stack` de `Location`s par un appel à `LocationDestroy`.

7 Extensions possibles et optionnelles

7.1 Nouvelle commande

Créez une nouvelle commande "exits" qui donne la liste des issues pour la case courante.

7.2 Gestion d'objets

Vous ajouterez des objets : les objets peuvent se trouver dans des cases et/ou être portés par le joueur. Par conséquent, on ajoutera aussi les commandes "get <object>", pour prendre un objet présent dans la case courante, "drop <object>" pour déposer dans la case courante un objet que l'on porte, "look <object>" pour regarder un objet que l'on porte où qui se trouve dans la case courante, et "inventory" pour lister les objets que l'on porte sur soi.

On pourra s'inspirer du module `location` pour faire un module `object`. On définira de plus un autre module `stackobject` qui instancie une pile d'objets. Cette pile sera ajoutée au `Game` pour gérer la mémoire. On utilisera aussi ce genre de pile pour stocker les objets présents dans une case (ajout d'un champ de type `StackObject` dans la structure `Location`), et également pour lister les objets portés par le joueur (ajout d'un champ de type `StackObject` dans la structure `Mobile`).

Une alternative, meilleure mais un peu plus avancée, consiste à ne pas définir de nouveau module `stackobject` mais plutôt à modifier la structure `Stack` afin de la rendre plus générale. Pour cela, il suffit de remplacer le pointeur vers une `Location` qu'elle contient, par un pointeur de type `void*` (pointeur générique). Bien entendu, il faudra aussi mettre à jour les fonctions du module. On pourra alors utiliser un `Stack` pour stocker des `Location*` aussi bien que des `Object*`, à condition de faire la bonne conversion de type.

7.3 Format de fichier monde

Dans une autre extension, au lieu de définir en dur les cases du jeu dans la fonction `LocationInit`, on pourra lire ces cases depuis un fichier texte. Le format suggéré décrit toutes les cases les unes après les autres, avec le format suivant :

— première ligne : "@room <n>" où <n> est le numéro de la case (numéro unique)

— deuxième ligne : description courte (nom) de la case
— troisième ligne : 6 entiers indiquant les 6 issues possibles dans l'ordre N,E,S,W,U,D. -1 indique une issue impossible, sinon c'est le numéro de case vers laquelle mène cette issue (voir la description de la première ligne)
— lignes suivantes (jusqu'au prochain @room) : description longue de la case

Si les objets sont implémentés, on utilisera un format similaire, par objet :

— première ligne : "@object <n>" où <n> est le numéro de l'objet (numéro unique)
— deuxième ligne : nom de l'objet
— troisième ligne : un entier indiquant soit le numéro de la case où se trouve l'objet, soit -1 s'il est sur le joueur
— lignes suivantes (jusqu'au prochain @object ou @room) : description longue de l'objet

8 Annexe : quelques éléments sur la compilation de modules et `make`

Nous employons une séparation du code en modules, ce qui est une bonne pratique dans le développement logiciel : cela permet d'avoir une bonne lisibilité du code avec des blocs de traitements bien définis, souvent autour de structures (ou d'objets comme vous le verrez en programmation objet), ce qui facilite aussi la phase de debug et permet éventuellement une réutilisation du code. Un exemple ici est le module `stack` que vous pouvez très bien imaginer réutiliser dans un autre contexte (à condition de le généraliser un peu...). Cette structuration nécessite cependant de bien identifier et séparer les 3 phases de compilations, ce qui est facilité par l'outil `make`.

Je commence par la **dernière et troisième phase** qui est la *phase d'édition des liens* : elle consiste à combiner des fichiers objets (compilés dans la phase 2) afin de former un exécutable. Dans cette phase, il faut que chaque fonction soit bien définie : pour chaque symbole de fonction, il faut trouver un code permettant de l'exécuter. Cette phase est assurée dans le `Makefile` par la ligne :

```
WorldOfIUT: WorldOfIUT.o cmd.o mobile.o game.o exits.o
```

Il faut donc indiquer tous les fichiers objets (extension .o) qu'il faut assembler pour faire l'exécutable `WorldOfIUT`. Vous aurez à ajouter `location.o` et `stack.o` après les avoir écrits.

La deuxième phase consiste en la *compilation* des fichiers C (extension .c) en fichiers objets (extension .o). Chaque fichier C est compilé séparément. Dans cette phase, on s'assure que chaque variable est bien déclarée, chaque type est bien défini (par exemple via une structure), et que l'appel de chaque fonction est correct, ce qui se fait par l'indication du prototype de chaque fonction. Cette phase ne vérifie pas la présence du code pour la fonction, ce qui sera fait par la troisième phase vue ci-dessus (édition de liens). Comme chaque fichier C est compilé séparément, il faut potentiellement réécrire toutes les structures, et tous les prototypes des fonctions utilisées, ce qui peut être fastidieux et source d'erreur. On passe donc par l'inclusion de fichiers qui contiennent ces informations : les fichiers *header* (extension .h). Ces fichiers contiennent la définition des types dont on a besoin ainsi que le prototype des fonctions qu'on va utiliser pour écrire le code C du fichier courant. L'inclusion des fichiers se fait par des directives de préprocesseur `#include` (voir la première phase ci-dessous). Comme ces fichiers sont inclus dans le code source, il faut indiquer que le fichier objet doit être regénéré si le fichier C change mais également si un des fichiers header inclus change. Le compilateur ne sait pas en faire la liste, qu'il faut donc indiquer dans le `Makefile`. C'est ce qui est fait par exemple pour générer le fichier `cmd.o` : par défaut il est régénéré quand le fichier `cmd.c` change et on ajoute la liste des fichiers header inclus, grâce à la ligne du `Makefile` :

```
cmd.o: cmd.h game.h exits.h mobile.h
```

Si on regarde effectivement le fichier `cmd.c`, il inclut le fichier `cmd.h` ainsi que `mobile.h` et `exits.h`. Les autres fichiers inclus sont hors projet et ne seront donc pas modifiés. Mais `cmd.h` inclut lui-même `game.h` qui inclut `mobile.h`, déjà pris en compte. `mobile.h` n'inclut aucun fichier supplémentaire, ni `exits.h`. La liste des fichiers header à prendre en compte s'arrête donc là. Il faudra ajouter les fichiers `location.h` et `stack.h` que vous allez créer. A vous de traquer les fichiers qui dépendent d'eux.

Dans ces fichiers header, le protoype d'une fonction ne reprend que la première ligne de sa définition (type de sortie, nom, type et nombre de variables d'entrée), en la faisant débuter par le mot clé `extern` et terminer par un point-virgule. Par exemple, la fonction `MobileNew` qu'on trouve définie dans `mobile.c` est déclarée par son prototype dans `mobile.h` par la ligne :

```
extern Mobile *MobileNew(char *name, char *desc);
```

On peut remarquer le `extern` initial et surtout le ; final. Les noms des variables d'entrée ne sont pas nécessaires et n'ont qu'un titre informatif pour indiquer le rôle de ces variables. Le prototype aurait tout aussi bien pu s'écrire :

```
extern Mobile *MobileNew(char *, char *);
```

Vous voyez que le compilateur a juste besoin de connaître le nom de la fonction ainsi que le type de ses variables d'entrée (dans l'ordre) et son type de sortie pour vérifier la viabilité d'un appel de cette fonction lors de la deuxième phase de génération du code objet. Vous aurez à ajouter une ligne pour la compilation du fichier `location.o`, et une autre pour le fichier `stack.o` une fois que vous aurez écrit ces modules.

La première phase est assurée par le *préprocesseur*. Cette phase est simplement une phase de remplacement purement textuel et elle ne s'intéresse qu'aux directives de préprocesseur, ces fameuses lignes qui commencent par # (par exemple tous les `#include` que nous avons déjà vus). Un `#include` va simplement insérer **tel quel** le fichier passé en argument. Les premières lignes avec `#ifndef`, puis `#define`, se terminent par le `#endif` final[4] : elles permettent de s'assurer que le fichier ne sera pas inclus plusieurs fois dans le même fichier C en définissant une variable lors de la première inclusion et n'autorisant l'inclusion que si cette variable n'a pas été définie (d'où le n dans `#ifndef`, la direction `#ifdef` existe aussi, et elle se termine aussi par un `#endif`). Voir la section de la documentation de gcc réservée à cette question.

4. cette directive ne prend pas d'argument et j'ai donc l'habitude de rajouter en commentaire le nom de la variable dont on a testé la définition (ou non-définition) dans le `#if` correspondant afin de faciliter la lecture de ces directives

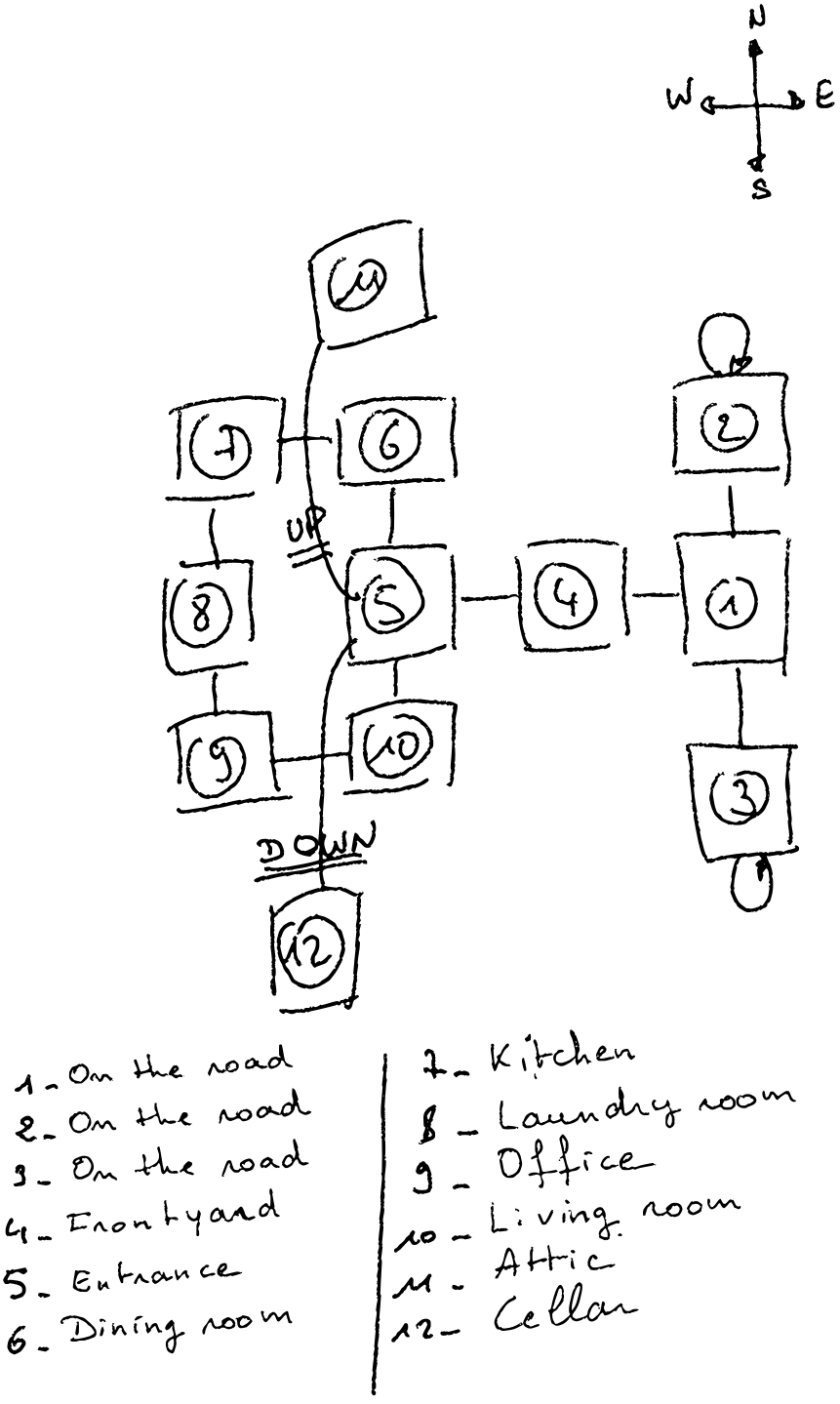

FIGURE 1 – Monde minimal à instancier dans `LocationInit`. Les noms sont à utiliser, mais les descriptions sont laissées à votre imagination.