Last Task

This assignment assesses your knowledge in STL, and general programming skills and program design. **You are NOT allowed to use primitive (C-style) arrays or "raw" pointers in this assignment**, except where it is necessary in interfacing with the C++ language/standard library, such as filenames, command-line arguments, STL functions expecting function pointers, etc. You are therefore strongly advised to use some kind of STL containers for any collection of objects you need to store. (You can use smart pointers or the STL "array" object if you want to.)

You will implement a very rudimentary "text adventure" game.

Text Adventure Games

In a text adventure game, players interact with the game using text commands, and what happens in the game is also described in text. Usually players need to navigate or explore some kind of unknown environment and to achieve certain objectives. Often the objective of the game, or even the permitted commands, are not stated clearly, and it is part of the game (and fun) for the player to discover them.

For an example of what these games are like, try the game "dunnet", which comes with the text editor Emacs. If you have Emacs installed, the game can be started from within the editor (type 'emacs' in the terminal to start it, then type 'Alt-x dunnet' to load the game). Or it can be started in "batch mode" directly from the linux terminal with the command "emacs -batch -l dunnet". Both work in the departmental linux system. (Try adding -Q to the list of options if it doesn't work.) There is also a web-based version. For details on the game and a guide, see here. Or you can try Zork, where a web-based version is in here.

Don't worry, you are not asked to produce something with that sort of complexity. In this assignment you are only asked to implement a very simple game, with a limited type of map, a very limited set of commands, and a simple goal to kill all "enemies".

Your Game

Part of the game data is stored in a separate map data file, to be read in by the program when in starts. It includes things such as details of rooms, objects, enemies and the player. This allows the same game to be played with different maps. The following description will often mention this "map data file". This map data file will be explained in more detail later.

Navigation

The player is in some kind of 2-dimensional layout, which for simplicity we refer to as a collection of rooms. Each room may have doors to some other rooms, only in the four directions: east, south, west, north. This is given in the map data file.

At any one point, the player is in one particular room. The initial room that the player is in when the game starts is specified in the map data file. Upon entering a new room, the game should print a description of the room, what objects are there, and what enemy (if any) is there.

The player can move to other rooms with the command "go east" (similarly for other directions). If there is a door to that direction, the player is moved to that room. Otherwise the program prints some kind of error message (and the player stays in that room).

Objects

Various objects are populated in the rooms initially (as specified in the map data file). Some of the objects are useful (needed to kill the enemies), but some may not be. The player can issue command "take xxx" to pick up the object xxx (and carry the object with him/her from that point onwards). Note that the object name (xxx) may actually contain spaces in it. The player can also issue the command "list items" to list all items picked.

Once picked up by the player, those objects follow the player and no longer stays in the room. For example, if a player picks up some objects from a certain room, move elsewhere, and later enter that

room again, those objects should not be listed to be in there when the room is described upon re-entering.

Enemies

There are some enemies in the game, Each enemy has a name ("monster", "zombie" etc), is in some initial room, and can be killed by some object or some combination of objects. All these are specified in the map data file.

While in a room with an enemy, if the player's command is "kill xxx" (where xxx is the name of the enemy - may also contain spaces), the program should check whether the player possesses all the objects required to kill this enemy. If so, the enemy is dead. (The player doesn't need to specify what objects to use, e.g. kill XXX with YYY; they will "automatically" be used. What set of objects is needed to kill the enemy is also not necessarily described to the player.) The dead enemy is removed from the game (if the player enter that room again it should not see the enemy still there). If the player issue the kill command but does not have the correct set of objects to kill that enemy, the enemy will attack the player and the player dies. The program should display some simple "game over" message and then quit.

If the player issue any other command instead (in a room with enemies), such as trying to leave the room, there is a 50% chance that the enemy will attack and so the player dies. In the other 50% chance the attack doesn't happen and the player's command is actioned. (See e.g. https://www.cplusplus.com/reference/cstdlib/rand/ on how to generate random numbers in C++.)

The above description assumes that at most one enemy is in a room. If there are more than one enemies, you can decide what happens, as long as the enemies are not "weaker" than as described above.

For simplicity, it is assumed those objects "used" to kill the enemy are still with the player after a successful kill.

The player wins when all the enemies are killed. When that happens, it should print some simple congratulatory message, then end the program.

Map data, JSON files and parsers

The program must read map data from a file supplied in the command line argument, such as:

```
./main map1.json
```

Three sample map files are given to you. They are in JSON format. Please read the contents of those sample files; they should be pretty self-explanatory.

You may add or change the structure of these JSON files, for example if you need to add extra attributes to objects to "enhance" the game.

You are encouraged to use this JSON parser: https://github.com/nlohmann/json. It has the advantage that only one header file is needed; a copy is included here. You can read the documentation from the above link; basically you only need to read the section on "STL-like access". Some simpler explanations/examples on how to use it with our sample map json file is given here.

However, if you want to, you can use other existing JSON parsers, but they must not require installation of any libraries; it must run out-of-the-box on the departmental linux platform. You can also write your own code for reading JSON files, if you so wish.

You do not even need to use JSON format; if you wish, you can design any file format and write code that read them accordingly. But if you do so, you must supply three map data files that contain "equivalent" information as the three sample maps. For convenience you should still name them with .json suffix even if they are not actually JSON files.

Your program can assume the map data files have no errors; if the map data file does not conform to these formats, the program

behaviour is undefined. Similarly, you can assume that the files represent a legal input, e.g. room IDs in the n/e/s/w fields actually correspond to some room IDs. Otherwise, the program behaviour is again undefined.

Further functionalities

As you can see, this is hardly a fun "game", so you should try to "enhance" the game. In fact you will need to do that if you want to get to the highest range of marks (see marking criteria). Some examples of what could be done include:

- Maybe the player doesn't die immediately after one attack, but have their "health points" reduced, and their health points can be replenished somehow;
- Some doors need keys to open (which the player picks up as objects somewhere);
- Some objects are reusable (guns) but some objects are perished after used once (grenade);
- Enemies can move, either randomly or somehow "sense" where the player is and go after them;
- Load/save game;
- More complex natural language processing ("open door with key" and so on);
- ...

Any extra functionalities must remain "compatible" with what is stated above, in other words the three sample files (or their equivalences) must still be playable with the same set of input commands.

When designing features you may be faced with a question of "Should I hard-code this command/object/message/... in the C++ code itself or should I allow this to be specified in the map?" Basically, you can do it in any way you like as long as it doesn't contradict anything specified here.

If you do implement extra features you must document them in a readme.txt file, and (if needed) supply extra map files that showcase them, or I may miss them.

Criteria and Testing

This task is unlike all the previous programming tasks; you have to develop all your program from scratch. No partially written .h or .cpp file will be provided. It is up to you to decide how to structure your program, what data structures to use, what classes to create (if any), or how the source code is split into different files (if you choose to do so). There is no requirement that the program must be very "object-oriented"; you can write the whole program in one .cpp file without defining any classes, if you so wish. Even though the marking criteria will give considerations to these design choices, it does not mean that there is only one "correct" or "best" design. Remember, C++ supports a number of programming paradigms.

You can also name your file(s) in any way. However, you must supply a makefile so that when typing 'make' the program will be compiled into an executable file called 'main'. Please refer to an earlier tutorial on how to write makefiles.

There are no runnable test suites as in previous assignments, although there is a sample completed program and three sample map files (see next section). As usual, your program will only be tested on the departmental linux system, and this time with your makefile.

Files Provided

- Main (beta, version 0.1)
- Main.exe (not ready yet)

These are linux and Windows executable files showing an example implementation. This is just an example; you do not have to follow exactly the input/output, for example.

- Map1.json
- Map2.json
- Map3.json

These are three sample map data files, in json format.

Submission Instructions

You should package all files in your submission in a zip archive. The file MUST be called finaltask (lowercase). At least the makefile should be in the top level folder in your archive; furthermore you are strongly encouraged to put everything in only the top level folder (no subfolders). Mac users: please avoid zipping on a Mac if at all possible. It creates extra folders making automatic extraction difficult. Or it may help if the folder containing your code (before zipping)

Your zip file should include any source files needed to compile your program. In particular, please remember the makefile, and readme.txt if you have extra features. Do not include anything unnecessary, such as the entire project folder of whatever IDE you are using.

The recommended JSON parser file json.hpp and the three sample map files will be added to the test folder before testing, if you do not submit them. Hence there is no need to submit them if you do not make any changes to them. You can submit modified and/or extra map files if you wish, but the first three must be "equivalent" to the three sample ones and they must be named sequentially map1.json, map2.json,... and so on.