TEXT-BASED ADVENTURE GAME

O ye CPSC 2720 students!

Harken back unto an era of yesteryear before such fancy contrivances as graphic cards and first-person shooters. A time when game playing required the use of imagination. When a game's interface was nothing but keyboard characters on a screen. A simpler time. A gentler time.

Ye are tasked with creating a new world for others to explore in a grand adventure. Ye may choose the world that you want to build, be it knights and dragons, aliens and spaceships, hard-boiled detectives and femme fatales, a haunted house inhabited by restless spirits, or a treasure hunter in the deep jungles of Africa. But choose ye wisely for many a CPSC 2720 student has been lost to such games and never returned!

TRANSLATION

In this project, your team will develop a text-based adventure game. The setting for your game will be decided by your team. Examples include:

- Fantasy (knights, dragons, elves, castles, etc.)
- Space (astronauts, aliens, spaceships, etc.)
- Contemporary (current society and locations)
- Nostalgic (deep-dark Africa explorers, 1920's hard-boiled detective, haunted house)

REQUIREMENTS

Your text-based adventure game is required to meet the following minimum requirements:

- **Multiple environments.** The game must present the character with different environments to traverse (e.g. a cave, a castle, a forest, or different types of rooms in a haunted house). To keep things manageable, and to prevent the game from getting too long, the environment should have no more than 25 'rooms' (i.e. a 5 x 5 grid).
- **Multiple characters**. The game will have supporting characters (non-playable characters) that the main character interacts with (e.g. A king that gives a knight a quest, a woman that hires detective, a bridge guardian that asks questions of a traveler)
- **Multiple actions.** The main character can interact with an environment or items in different ways (e.g. Movement, fighting, using items)
- **Multiple usable items.** A usable item allows the main character to interact with another object in some way (e.g. a key to open a door or a chest). Not all usable items need to be useful in the game.
- **A basic plot.** The basic plot of the game will involve an end goal achieve (e.g. Rescue a princess from a dragon, discover who stole a priceless statue, and defeat an alien invasion on a contested planet). To achieve the goal requires solving multiple puzzles (min. 5 puzzles, max. 10 puzzles). Examples of puzzles include finding a key to open a door, answering a simple

riddle, or deciphering a message. Finally, there must also be different ways (min. 5) that player can lose the game (e.g. eaten by a dragon, shot by the police, and didn't prevent an alien invasion in time).

- **The ability to save and restore progress.** Your game must allow a user to quit the game and then resume from where they left off (or as close to where they were in the game as is reasonable). In other words, develop a text file data format that can describes the current state of the game, can be written out to disk and can be read from disk to continue the game.

- **Provide means for getting help** when playing the game (e.g. If the user enters "help", a list of all possible actions is printed to the screen).

- **Error-checking to prevent program crashing** (e.g. Validating all input from the user, ignoring/warning about nonsensical actions)

PROJECT PHASES

The project will have a number of phases:

1. *Design* where the game will be designed and the project planned.
2. *Implementation* where the game is implemented.
3. *Testing* where your team will test a game developed by another team.
4. *Maintenance* where your team will address feedback from the team testing your game, or improve your game.

PROCESS CONSTRAINTS

To help keep the project manageable across the different teams, you will have the following process constraints:

USER INTERFACE

The game will be implemented as a text game (i.e. no GUI) with all interaction on the command-line. If you would like to use ASCII art, that is fine, **however** the art must be found in one or more separate files that are read in to keep the code readable (this is also good SE practice).

SOFTWARE TOOLS

1. The project is to be developed in **C++**.
2. A **Makefile** will be created for building your game, running tests, checking quality of code, and so forth.
3. All artifacts (source code, documentation, reports, and reported issues) will be kept in a provided repository on the department's **Gitlab** server.
4. **GTest** will be used as the unit testing framework.
5. **Cppcheck** will be used for static analysis
6. **Cpplint** will be used for checking programming style using the configuration provided with the assignments.
7. **Memcheck** (i.e. Valgrind) will be used for checking memory leaks.
8. **Gitlab Pipelines** will be used for continuous integration.

PLATFORM

1. The project will run in the Linux environment of the University of Lethbridge computer science labs.

Repository Organization

Your repository must be organized in a logical fashion (i.e. do not have all files at the top level!). Your repository is required to have at least the following top-level directories and files (so the grader can easily find the files and build your project), but you can add other directories according to your project needs:

- `Makefile` – a Makefile that has the following targets:
  - `compile` – compiles the project.
  - `test` – compiles and runs the unit test cases.
  - `memory` – runs `memcheck` on the project.
  - `coverage` – runs `lcov/gcov` on the project.
  - `style` – runs cpplint.py on the project.
  - `static` – runs cppcheck on the project.
  - `docs` – generates the code documentation using `doxygen`.
- Code::Block project files `(.cbp, .depend, .layout)` can be at the top/root level.
- `src` - the implementation files (`.cpp`)
  - `game` – contains the `main.cpp` for running the game
- `include` – the header files (`.h`)
- `test` – the `GTest` test files, including the `main.cpp` to run the tests
- `docs` – contains the project documentation, with the following sub-directories:
  - `design` - design document and UML diagram image files (`jpg` or `png`)
  - `code` - contains the `doxyfile`, and the generated `html` folder with the output of `doxygen`.
  - `user` - user manual
  - `testing` – testing and maintenance report
  - `team` – the team reports
    - `design` – report from the design phase
    - `implementation` – report from the implementation phase
    - `testing` – report from the testing phase
    - `maintenance` – report from the maintenance phase