ADVENTURE GAME (TEXT-BASED)

In this project, your team will develop a text-based adventure game. Your team will decide the setting for your game. Examples include:

- Fantasy (knights, dragons, elves, castles)
- Space (astronauts, aliens, spaceships)
- 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 prevent the game from getting too long, the environment should have no less than 9 locations (e.g. 3x3 grid) and no more than 25 locations (e.g. 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 a 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. The expected actions are grouped into the following categories (with examples):
    - **Movement –** forward, backwards, turn left, turn right, run, walk.
    - **Environment** – look, touch, examine, dig, climb.
    - **Interaction**:
        - *Objects* – pick up, put down, throw.
        - *Other characters* – speak, attack, persuade.

- **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 helpful in the game.
- **A basic plot.** The game's basic plot will involve an end goal (e.g. Rescue a princess from a dragon, discover who stole a priceless statue, defeat an alien invasion on a contested planet). Achieving the goal requires solving multiple puzzles (min. five puzzles, max. ten 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. three) that the player can lose the game (e.g. eaten by a dragon, shot by the police, or didn't prevent an alien invasion in time).
- **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 four phases:

1. *Design* where the game is designed, and the project planned:
    1. *Design Draft* – A preliminary design submitted for feedback.
    2. *Final Design* – The final design.
2. *Implementation* where the game is implemented:
    1. *Basics* – The player can:
        i. Move between two adjacent locations.
        ii. Interact with the environment in one way (e.g. examine a room)
        iii. Interact with an object in two ways (e.g. pick up and put down)
        iv. Unit tests for movement and interacting with objects actions (see below for examples)
    2. *Completed* – The remaining requirements are completed (e.g. all locations, interact with other characters, get help)

UNIT TESTING

You are expected to create unit tests to verify the correctness of your software, specifically the player's actions and environmental items that affect them. Examples of such tests include:

- Is the player's location attribute updated correctly when the player moves from one location to another?
- Does a player's inventory increase by one when they pick up an item?
- Does a player's inventory decrease by one when they drop an item?
- Does a player's health increase when they eat food?
- Does a player fall asleep when they eat a poisoned apple?
- Does a player's health decrease when they touch a flame?
- Does a player die when a bullet hits them?

The code coverage report for the final implementation should show that most, if not all, of the code for player's actions, are covered. Code that cannot be reasonably tested using automated testing, such as printing to the screen or getting user input, is not expected to be covered by unit testing.

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 interactions 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** created to build your game, run tests, check code quality.
3. An example file will be provided, but you may need to modify it for your specific software.
4. All artifacts (source code, documentation, reports, and reported issues) will be kept in a provided repository on the department's **Gitlab** server.
5. **GTest** will be used as the unit testing framework.
6. **Cppcheck** will be used for static analysis
7. **Cpplint** will be used for checking programming style using the configuration provided with the assignments.
8. **Memcheck** (i.e. Valgrind) will be used for checking memory leaks.
9. **Gitlab Pipelines** will be used for continuous integration.

PLATFORM

The project must run in the Linux environment of the University of Lethbridge cs.uleth.ca network.

REPOSITORY ORGANIZATION

Your repository must be organized logically (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). However, 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.
  - memcheck – runs valgrind on the project.
  - coverage – runs lcov/gcov on the project.
  - style – runs cpplint.py on the project.
  - static – run cppcheck on the project.
  - docs – generates the code documentation using Doxygen.
- 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 (design.pdf) and UML diagram image files (jpg, png or dia). The images must appear in the design document; it is not enough to only have them in the directory and then refer to the files within the design document.
  - code – contains the doxyfile, and the generated html folder with the output of doxygen.

NOTE

No files that can be generated when compiling and running appear in the repository, including executable files, code coverage reports, the documentation generated by Doxygen, or files created by VSCode/Git Bash/other software.