GO FISH

OVERVIEW

In this project, your team will develop a text-based game that allows the user to play the card game *Go Fish*.

REQUIREMENTS

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

- **Play Go Fish.** The game will support playing Go Fish.
- **A deck of cards.** The game will support the standard deck of 52 playing cards (four suits of 13 cards).
- **Rule enforcement.** The software will enforce the rules of Go Fish. See https://bicyclecards.com/how-to-play/go-fish/ for the rules.
- **Simple Computational Intelligence**. The user will play the game against two (2) C.I.s (i.e. computer opponent). The C.I. are not be sophisticated:
  - One will ask for the lowest card in their hand (that they have not yet asked).
  - One will ask for a random card from their hand.
- **Help.** Provide a means for getting help when playing the game (e.g. `help` prints out the rules of game, explains the interface/commands, or suggests a card to ask).
- **Robust.** Error-checking to prevent program crashing (i.e. validating all input from the user).

PROJECT PHASES

The project will have two phases:

1. *Design* where the game will be designed and the project planned.
2. *Implementation* where the game is implemented.

PROCESS CONSTRAINTS

To help keep the project manageable, 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. **Cppcheck** will be used for static analysis
5. **Cpplint** will be used for checking programming style using the configuration provided with the assignments.
6. **Memcheck** (i.e. Valgrind) will be used for checking memory leaks.
7. **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.
  - `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`.

GRADING

You will be graded based on your demonstrated understanding of the good software engineering practices covered so far in the course.

Examples of items the marker will be looking for include (but are not limited to):

- Appropriate software design for this point in the course (e.g. the program should not be a single class with one large method). This also include appropriate functional separation across the class methods (e.g. separating the I/O from the game mechanics).
- Evidence of unit testing of public methods.
    - Not all public methods need to be tested, but you do need to show some effort towards unit testing.
    - Do not unit tests methods that require user input – these will hang the CI server!
- Use of equivalence partitioning in the creation of test cases. At minimum you should be able to unit test for the following two situations:
    - When the player gets a card (either by drawing a card or asking) and completes a set.
    - When the player gets a card (either by drawing a card or asking) and does not complete a set.
- Version control history shows an iterative progression in completing the project.
    - You are expected to have a **minimum of five commits** in your repository.
- Version control repository contains no files that are generated by tools (e.g. object files, binary files, documentation files)
- Public methods are appropriately documented.
    - The generated HTML files from running `doxygen` on your project will be examined.
- Memory leak checking (both interactive and automated unit testing), static analysis and style analysis show no problems with your code.

SUBMISSION

There is no need to submit anything, as project repositories were created for your team and te grading will be done on the contents of these.

- Creating **a different project repository will result in an automatic 0 (zero)** as the marker will not be able to find it.