The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310 - Semester 2 2020
Assignment 2 (v1.1)

Marks: 60
Weighting: 33% of your overall assignment mark
Due: 23:59 2 October, 2020

Student conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on Piazza. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students with the actual design and coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, **both of you are in trouble**.
You may use code provided to you by the CSSE2310 teaching staff **in this semester**.

Uploading or otherwise providing the assignment specification to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and they cooperate with us in misconduct investigations.

The course coordinator reserves the right to conduct interviews with students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. You should read and understand the statements on student misconduct in the course profile and on the school web-site:
`http://www.itee.uq.edu.au/about_ITEE/policies/student-misconduct.html`

Aim

Your task is to write a group of three programs to participate in a naval battle tournament. One of the programs will be called `2310hub` and will supervise the running of the tournament. The other two programs (`2310A` and `2310B`) will be players (referred to as agents).

The tournament is based on the `naval` game, which you developed for Assignment 1. Each competition round in the tournament is a battle between two agents. The first agent to successfully sink their opponent's ships wins their round.

The hub will be responsible for running the agent processes and communicating with them via pipes (created via `pipe()`). From the agents' point of view, communication will be via `stdin` and `stdout`. Agents will also produce output to `stderr`.

Other than the hub starting agent processes, there should not be any other parallelism used in this assignment. For example, no multi-threading nor non-blocking operations. Your programs are not permitted to start

any additional processes[1] nor are your programs permitted to create any files during their execution.

2310hub

The hub will take the following command line arguments (in order):

- the name of the rules file
- the name of the config file

Example usage: `./2310hub rules.txt config.txt`

When running agent processes, the hub must ensure that any output to `stderr` by agents is suppressed. When the hub receives a `SIGHUP`, it should kill and reap any remaining child processes. Note: We won't test the exit status for `2310hub` when it receives `SIGHUP`.

Note on file handling and formatting

Input files for marking this assignment will be well-formatted, exactly according to the description and examples in this specification. This assignment is about game logic and process management and communication, not the minutiae of parsing text files!

SPEC CLARIFICATION - The following paragraph was tweaked to make it clearer, the meaning is unchanged

In regards to message formats: most messages will consist of a certain number of comma-, colon- or space-separated fields. A message must contain a valid number of fields (based on its format). Each of these fields is to be processed verbatim and consist of a single token. This token can be surrounded by whitespace, but cannot contain whitespace within its body. For tokens representing/containing numeric values, the numeric component of token is considered valid provided that it can be converted successfully using standard C library functions. For example: " `GUESS D3` " consists of two tokens (`GUESS` and `D3`) separated by whitespace, neither token contains any internal whitespace and all of the numeric components of `D3` can be converted successfully using C library functions. Similarly "`RULES 15, 20, 5` " consists of four tokens (`RULES`, `15`, `20` and `5`) separated by commas, none of the tokens contain internal whitespace and all of the numeric tokens can be converted successfully.

Because your assignment and its components are auto-marked by exact text matching, you must adhere to the output formats precisely.

Rules file

The rules file describes the board and the rules for the game. The file format is identical to that from Assignment 1

- Lines beginning with the `#` character are comments, and must be ignored
- Empty lines are ignored
- The first line consists of two positive integers (>0) giving the width and height of the board
- The next line consists of a single integer giving the number of ships for each player. You may assume that there will not be more than 15 ships per player.

[1] other than those specified on the hub command line or configuration file

- Subsequent lines define the length of the ships starting from ship 1

```
# 8x8 board
8 8
# 5 ships per player
5
# Here follow the ship lengths
5
4
3
2
1
```

Listing 1: Format of rules file

Config file

The config file details the tournament competition rounds.
Each line describes a competition round by specifying

1. the agent programs participating in the round, and
2. the file paths to the agents' map files.

The format for the config file is as follows:

- Lines beginning with the `#` character are comments, and must be ignored
- Empty lines are ignored
- All other lines consist of four comma separated fields and have the following form (ending in `\n` only):
  `agent1,filename1,agent2,filename2`
  where
  - `agent1` and `filename1` are the program path and map filename of the first agent (id = 1) competing in the round and,
  - `agent2` and `filename2` are the program path and map filename of the second agent (id = 2) competing in the round.

  The program path is a relative or absolute path to the executable that will be used as an AI for that agent.

Furthermore, note the following:

- The relevant agent program (`2310A`, `2310B` or otherwise) must be `exec()`ed from the current working directory of the hub program.
- The agent listed first in a round, goes first. For example in the above line, `agent1` will go first.
- map filenames may contain directory slashes (`/`) to indicate the path to the file, however these names may not contain commas or any other special characters.

Each competition round is referred to by a round number. Round numbering starts at 0, and is determined by the round's location in the config file (i.e. the round described by the first (non-comment) line in the file is given the number 0, with the round number incremented for each subsequent (non-comment) line).

A basic config file representing a single round would look something like:

```
# Round 0 - Agent 1 is './2310A', Agent 2 is './2310B',
# ./2310A goes first, map*.txt are in the current directory
./2310A,map1.txt,./2310B,map2.txt
```

Listing 2: Single round tournament config file

while a more complex tournament would look something like:

```
# Round 0 - Agent 1 is './2310A', Agent 2 is './2310B',
# ./2310A goes first, map*.txt are in the current directory
./2310A,map1.txt,./2310B,map2.txt
# Round 1 - ./2310A vs. ./2310A, maps are in a subdirectory
./2310A,tests/map3.txt,./2310A,tests/map4.txt
# Round 2 - ./2310B vs. ./2310B
./2310B,tests/map3.txt,./2310B,map3.txt
```

Listing 3: Multiple round tournament config file

Hub operation

Game startup

1. Load and check the rules file
2. Load and check the config file
3. Create child processes and communication pipes for all agents, `exec()`ing the agent programs as required with the appropriate agent command-line arguments

The child processes should be started in the order they are listed in the configuration file. For a given round in the file, the first player should be spawned first, followed by the second player.

After the hub has spawned all of the child processes, it should then send each agent a `RULES` message, and await a `MAP` message in reply. These messages should be received/sent to agents in the order they were started.

If any of the following conditions occur

- the hub is unable to start an agent process,
- the hub receives a malformed `MAP` message from an agent, or
- the hub receives a `MAP` message that describes ships overlapping or out of bounds

then the hub should immediately kill both agents participating in that round. Any other rounds should proceed if they can.

If no rounds can be started, the agent should exit with an "Error starting agents" error (see the 'Hub Errors' section for details and exit code).

Once the hub has started up all of the agent processes, gameplay will begin.

Gameplay

The hub uses round robin scheduling to manage competition rounds (i.e. it will circle through the rounds, prompting agents for one move each time, starting from the first competition round).

SPEC CHANGE - Some of the sequence numbers where wrong in the original version

The following describes how the hub will operate:

1. The hub will print ten asterisk characters to stdout followed by newline
   `**********`
2. The hub will print the current competition round number to `stdout` as follows:
   `ROUND <r>` (newline character at end)
   where `<r>` is the round number.
3. The hub will display the boards of the two agents competing in the current round to `stdout`, with all ships/hits/misses revealed. See below for display details.
4. The hub will prompt the first competing agent for their move (using the `YT` message). The hub will continue prompting the player until a valid move is received. Note that a valid move in this case means a well-formed message with a target which is on the board which has not been targeted before.
5. After receiving a *valid* move from the agent, the hub replies to that agent with the `OK` message, and then sends either a `HIT`, `SUNK` or `MISS` message to **both agents** reflecting the result of the move.
   The hub will also print information about the agent's move to `stdout` as follows:
   `<result> - player <p> guessed <pos>` (newline character at end)
   where
   - `<result>` is either `HIT`, `MISS` or `SHIP SUNK`,
   - `<pos>` is the coordinate of the agent's guess (e.g. `D3`), and
   - `<p>` is the guessing agent id (either `1` or `2`).
6. If an agent's last ship has been sunk the hub should proceed to step 8).
7. The hub will repeat steps 4 - 6 for the second competing agent in the round.
8. If game over is reached, the hub should send the `DONE` message to both agents in the current round and print
   `GAME OVER - player <id> wins` (newline character at end)
   where `<id>` is the id of the winning agent (`1` or `2`).
9. The hub should repeats steps 1-8 for each remaining competition round. Rounds that have reached the Game Over state should do steps 1 - 3, but skip the move/gameplay for that round.

The entire sequence above is repeated until all battles are completed, the hub receives `SIGHUP`, or some other error occurs. If the hub receives `SIGHUP`, it should terminate all child processes and exit.

All of the above hub output is to `stdout`.

Hub output example

The following shows sample output from the hub for a single round tournament.

```
**********
Round 0
   ABCDEFGH
 1 ..11111.
 2 ........
 3 ....5...
 4 ..3.....
 5 2.3.....
 6 2.3.44..
 7 2.......
 8 2.......
===
   ABCDEFGH
 1 1......5
 2 12......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
 8 ........
HIT player 1 guessed A1
SHIP SUNK player 2 guessed E3
**********
Round 0
   ABCDEFGH
 1 ..11111.
 2 ........
 3 ....*...
 4 ..3.....
 5 2.3.....
 6 2.3.44..
 7 2.......
 8 2.......
===
   ABCDEFGH
 1 *......5
 2 12......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
 8 ........
MISS player 1 guessed B1
MISS player 2 guessed H1
```

```
**********
Round 0
   ABCDEFGH
 1 ..11111/
 2 ........
 3 ....*...
 4 ..3.....
 5 2.3.....
 6 2.3.44..
 7 2.......
 8 2.......
===
   ABCDEFGH
 1 */.....5
 2 12......
 3 12..333.
 4 12......
 5 12...4..
 6 .....4..
 7 ........
 8 ........
MISS player 1 guessed B1
MISS player 2 guessed H1

.... some time later ...

**********
Round 0
   ABCDEFGH
 1 ..*****/
 2 ....///.
 3 ....*...
 4 ..*.....
 5 *.*.....
 6 *.*.*4..
 7 *///....
 8 *...//..
===
   ABCDEFGH
 1 *//////*
 2 *2......
 3 *2..333.
 4 *2.////.
 5 *2...4..
 6 .....4..
 7 .//.....
 8 .../////.
MISS player 1 guessed H8
SHIP SUNK player 2 guessed F6
```

```
GAME OVER - player 2 wins
```

Listing 4: Sample hub output, single round tournament

Hub errors

All error output to be sent to `stderr`. Check the following conditions in order:

Exit	Condition	Message
0	Normal end	
1	Incorrect number of args	`Usage: 2310hub rules config`
2	Invalid rules file or contents	`Error reading rules`
3	Invalid config file or contents	`Error reading config`
4	Error starting agents	`Error starting agents`
5	Communications error	`Communications error`
6	Received `SIGHUP`	`Caught SIGHUP`

Any communication error encountered by the hub results in all competition rounds being ended early. This includes an unexpected `EOF` condition on any pipes read by the hub.

Agents

All agent processes take the following command line arguments (in order).

- This agent's id (either `1` or `2`)
- The filename of the agent's map file
- An integer representing the random seed for this agent (ignored by agent `2310A`, but required by agent `2310B`)

Once the agent has checked its command line arguments, it should:

1. Read the board dimensions from `stdin` (parsing the `RULES` message)
2. Print the positions of its ships to `stdout` (using the `MAP` message)

Map file

The map file describes the locations and directions of an agent's ships. It has the same format as Assignment 1. For example:

```
A1 S
B2 S
E3 E
F6 N
H1 W
```

Listing 5: Sample map file

Notes on positions and rows/columns

Alphanumeric positions are the same from assignment 1, e.g. `A1` is the top left most square. Columns are labels `A` through `Z`, rows are numbered `1` through `26`.
Numerical coordinates are zero based, in the form *`col, row`*, with zero-based indexing. So, `A1` is equivalent to `(0,0)`, `C4` is equivalent to `(2,3)` and so on.

Agent A strategy

Agent A makes guesses according to the following logic:

- Locate the topmost row which has cells in it that have not yet been guessed.
- If the numerical row number (zero-based) is even, then guess the leftmost cell in the row, which has not already been guessed.
- If the row number is odd, then guess the rightmost cell in the row, which has not already been guessed.

Agent B strategy

Agent B is more complex in that it switches between two phases ('search' mode and 'attack' mode) when selecting its next attack. The agent starts a game in search mode.

Search Mode: The player will first choose the row it will target and will then choose the column. The numerical (zero-based) row number will be calculated as

$$r = r_1 \mod h$$

while the numerical (zero-based) column will be calculated as

$$c = r_2 \mod w$$

where h is the board height, w is the board width, and r_1,r_2 are consecutive random numbers generated by the C `rand()` function after the random number generator has been appropriately seeded using the agent's `seed` command-line argument. The agent should continue to generate positions until it produces a position which has not been guessed yet.

If the agent's guess is a hit, it will switch into attack mode for its next guess. Otherwise, the next guess will be randomly selected using the same search strategy.

Attack Mode: Once the agent first switches into attack mode, it will start keeping track of a list of positions to target next. The first positions it will target are the valid, non-targeted, non-diagonal, adjacent positions directly surrounding the position where the hit which switched it into attack mode occurred. These positions should be targeted in the following order: `N, E, S, W`.

While there are still positions to target, the agent will remain in attack mode and will choose its next attack in the order in which they were being kept track of. If an attack is a hit, it will perform the same tracking operation as explained above (i.e. targeting adjacent positions surrounding the hit position). Note that locations which have been, or are currently being tracked should not be tracked again. Once there are no more positions to target, the agent will switch back to search mode.

Notes on random numbers

The random number generator behind `rand()` is psuedo-random, and must be initialised with a seed value. Any given seed value will result in a predictable sequence of pseudo-random numbers being produced. This is is how we can get you to use randomness in your assignment while still having known output to mark against.

However, this means **it is critical that your agent correctly call `srand()`, and only call `rand()` in the way and sequence described above**. Any extra calls to `rand()` will disrupt this sequence and cause your tests to fail.

Here's some pseudocode to help you out (although you are required to error check the argument, not shown here):

```
#include <stdlib.h>     // Needed for rand() and friends

int main(argc, argv[]) {
   ...
   // Seed the random number generator with argv[3] - the seed
   srand(atoi(argv[3]));
   ...
```

Listing 6: Pseudocode for agent random seed initialisation

When the hub starts agent processes it should use the following algorithm to determine the value of the `seed` command-line argument:

$seed = 2 * round\_num + agent\_id$

For example:

- Agent 1 in round 0 gets seed value 1 $(2 * 0 + 1)$
- Agent 2 in round 2 gets seed value 6 $(2 * 2 + 2)$ item and so on ...

Agent output

All of the following output is to `stderr`.

- The agent should print its board with ships/hits/misses revealed, followed by it's opponenents board, with hits/misses shown, separated by a line with three equals '`===`' signs, before it sends or receives any move information in the current turn
- Every time a `HIT/MISS/SUNK` message is received by the agent, information should be printed about the move:
  `<result>` `- player` *`<p>`* `guessed` *`<pos>`* (newline character at end)
  as described in the "Hub operation" section.

- Upon receiving a `DONE` message from the hub, information should be printed about game over:
  `GAME OVER - agent <id> wins`
  as described in the "Hub operation" section and the agent should exit normally.

Example agent output (to `stderr`)

The following shows possible output from an agent which is in the player 1 slot.

```
   ABCDEFGH
 1 ..11111.
 2 ........
 3 ....5...
 4 ..3.....
 5 2.3.....
 6 2.3.44..
 7 2.......
 8 2.......
===
   ABCDEFGH
 1 ........
 2 ........
 3 ........
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........
HIT player 1 guessed A1
SHIP SUNK player 2 guessed E3
   ABCDEFGH
 1 ..11111.
 2 ........
 3 ....*...
 4 ..3.....
 5 2.3.....
 6 2.3.44..
 7 2.......
 8 2.......
===
   ABCDEFGH
 1 *.......
 2 ........
 3 ........
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........
MISS player 1 guessed B1
MISS player 2 guessed H1
```

```
   ABCDEFGH
 1 ..11111/
 2 ........
 3 ....*...
 4 ..3.....
 5 2.3.....
 6 2.3.44..
 7 2.......
 8 2.......
===
   ABCDEFGH
 1 */......
 2 ........
 3 ........
 4 ........
 5 ........
 6 ........
 7 ........
 8 ........

 ... some time later ...

   ABCDEFGH
 1 ..*****/
 2 ....///.
 3 ....*...
 4 ..*.....
 5 *.*.....
 6 *.*.*4..
 7 *///....
 8 *...//..
===
   ABCDEFGH
 1 *//////*
 2 *.......
 3 *.......
 4 *..////.
 5 *.......
 6 ........
 7 .//.....
 8 .../////.
MISS player 1 guessed H8
SHIP SUNK player 2 guessed F6
GAME OVER - player 2 wins
```

Listing 7: Sample agent output to `stderr`

Agent errors

All error output should be sent to `stderr`. Check the following conditions in order:

Exit	Condition	Message
0	Normal end	
1	Incorrect number of args	`Usage: agent id map seed`
2	Player id is invalid	`Invalid player id`
3	Invalid or missing map file	`Invalid map file`
4	Invalid seed (e.g. non-integer)	`Invalid seed`
5	Communications error	`Communications error`

Communication errors from the agent's point of view include:

- unexpected EOF on `stdin`
- messages not being sent in the correct format or sequence

Agent/Hub inconsistency

The hub is the ultimate arbiter of the game state. We will not test cases where the agent and hub can have inconsistent beliefs about the game state, and the agent is not required to check for this possibility.

Messages

Note that broadcast messages (i.e. messages to all) are sent to both agents in a given round.

Direction	Format	Detail
hub → agent	`YT`	Agent should send back a move
hub → agent	`OK`	Hub acknowledges agent's move
agent → hub	`GUESS` *pos*	An agent's guess *pos*=position e.g. `"GUESS D3"`
hub → all	`HIT id,pos`	Inform agents that an agent hit a ship. *id*=guessing agent ID *pos*=position e.g. `HIT 1,A1`
hub → all	`SUNK id,pos`	Inform agents that an agent sank a ship. *id*=guessing agent ID *pos*=position e.g. `SUNK 1,A1`
hub → all	`MISS id,pos`	Inform agents that an agent's guess missed. *id*=guessing agent ID *pos*=position e.g. `MISS 1,A1`
hub → agent	`RULES` w`,`h`,`n`,`l_1`,`l_2`,...,`l_n	The board dimensions w=width of board h=height of board n=number of ships per player l_i = the length of the i_{th} ship e.g. `"RULES 8,8,5,5,4,3,2,1"`
agent → hub	`MAP` pos_1`,`d_1`:`pos_2`,`d_2`:...:`pos_n`,`d_n	The positions of an agent's n ships pos_n=position of n_{th} ship d_n=direction of n_{th} ship e.g. `"MAP A1,E:D3,S:G8,W:E3,N"`
hub → all	`EARLY`	Game ended early due to comms error
hub → all	`DONE <id>`	Game ended normally, agent *id* won

Message Sequencing

Figure 1 demonstrates the communication protocol sequence between the Hub and two agents (i.e. one round). For multiple rounds, the Hub does a single move sequence with Agent 1 and Agent 2 in each round, then proceeds to the next round.

Style

You must follow version 2.0.4 of the CSSE2310/COMP7306 C programming style guide available on the course BlackBoard site.

Submission

Your submission must include all source and any other required files (in particular you must submit a Makefile). Do not submit compiled files (eg .o, compiled programs) or rules or map files. You may not create any subdirectories in your submission. Such subdirectories will be removed prior to marking.

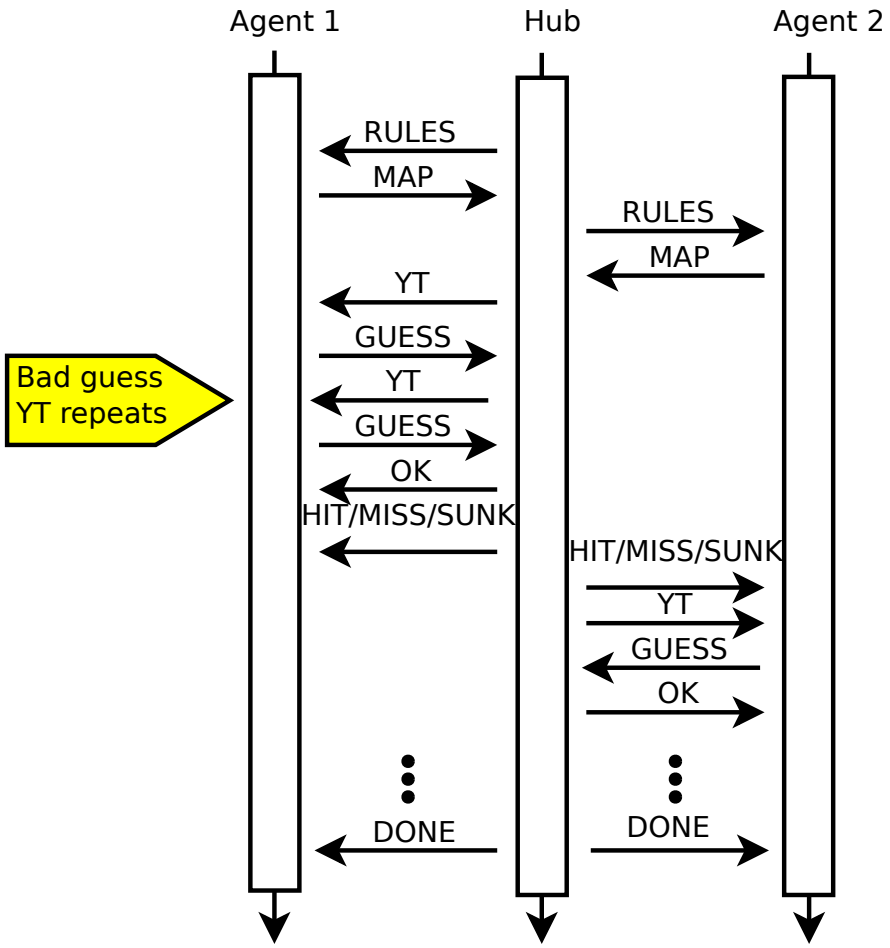

Figure 1: Hub / Agent communication protocol sequence

Your program must compile with the `make` command. Your program must be compiled under gcc with at least the following switches: `-pedantic -Wall -std=gnu99`. You are not permitted to disable warning or use pragmas to hide them.

If any errors result from the `make` command (i.e. an executable cannot be created), then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality). Your program must not invoke other programs or use non-standard headers/libraries.

The marking tests will run `make` in a clean SVN checkout of your repository, and will expect to find the following three programs

- `2310hub`
- `2310A`
- `2310B`

Your assignment submission must be commited to your subversion repository under

`https://source.eait.uq.edu.au/svn/csse2310-sXXXXXXX/trunk/ass2`

Note: `sXXXXXXX` is your moss/UQ login ID.

Your submission will be assessed by the contents of your SVN repository at the due date/time. Any subversion commits after the due date will be ignored. You must ensure that all files needed to compile and use your assignment (include a makefile) are committed and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes. The late submission policy in the CSSE2310 Electronic Course Profile applies. Be familiar with it!

Marks

Marks will be awarded for functionality, style and documentation.

Functionality (42 marks)

Provided your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program can detect hits correctly. The markers will make no alterations to your code (other than to remove code without academic merit).

Marks will be assigned in the following categories. Your hub and agent programs will be tested independently, so even if your hub doesn't work you can still get marks for the agents.

Hub (22 marks)

1. Program correctly handles invalid command lines and input files (2 marks)
2. Play a single round between two agents [13 total]
    - Sub category marks to be advised
3. Play multiple rounds between multiple agents [7 total]
    - sub-category marks to be advised

Agent A - scan (8 marks)

- sub-category marks to be advised

Agent B - search and attack (12 marks)

- sub-category marks to be advised

Style (8 marks)

Style marks will be calculated as follows: Let

- A be the number of style violations detected by `style.sh` plus the number of compilation warnings.
- H be the number of **additional** style violations detected by human markers. Violations will not be counted twice.

If $A > 10$ (i.e. you have more than 10 automatic style violations), then your style mark is 0 and H will not be calculated. Otherwise, your style mark is broken into two parts:

- an automatic mark ($M_A = 4 \times 0.8^A$), and
- a human marker mark ($M_H = M_A - 0.5 \times H$) (i.e. your human mark is capped by your automatic mark and you lose 0.5 marks for each style violation detected by a human marker)

Your final style mark will then be

- $S = M_A + \max\{0, M_H\}$

The number of compilation warnings will be the total number of distinct warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 2.0.4 of the CSSE2310 C Programming Style Guide.

A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name).

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final - it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark - this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission, however the marker has ultimate discretion – the tool is only a guide.

Documentation (10 marks)

PDF document containing a written overview of the architecture and design of your program (7 marks)

This document should describe, at a general level, the functional decomposition of the program, the key design decisions you made and why you made them.

- Submitted via Blackboard/TurnItIn prior to the due date/time
- Maximum 2 A4 pages in 12 point font

- Diagrams are permitted up to 25% of the page area. The diagram must be discussed in the text, it is not ok to just include a figure without explanatory discussion.

Don't overthink this! The purpose is to demonstrate that you can communicate important design decisions, and write in a meaningful way about your code. To be clear, this document is not a restatement of the program specification - it is a discussion of your design and your code.

If your documentation obviously does not match your code, you will get zero for this component, and be asked to explain why.

SVN commit history assessment (3 marks)

Markers will review your SVN commit history for your assignment from the time of handout up to the due date.

This element will be graded according to the following principles:

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit

Again, don't overthink this. We understand that you are just getting to know subversion, and you won't be penalised for a few "test commit" type messages. However, the markers must get a sense from your commit logs that you are practising and developing sound software engineering practices by documenting your changes as you go. In general, tiny changes deserve small comments - larger changes deserve more detailed commentary.

Total mark

Let

- F be the functionality mark for your assignment.
- S be the style mark for your assignment.
- D be the documentation mark for your assignment.

Your total mark for the assignment will be:

$$M = F + \min\{F, S\} + \min\{F, D\}$$

In other words, you can't get more marks for style or documentation than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Hints

- Use appropriate modularisation techniques to share common code between the hub and the agent programs
- Study and experiment with the example code provided in Weeks 4, 5 and 6 on `fork()`, `exec()`, `wait()`, `pipe()`, `dup2()` etc. Don't start writing the hub until you understand these examples!
- Draw yourself a diagram that shows the relationships between the hub and agent processes, and the communication channels between them.

- Draw a timeline that shows the possible communication sequences between the hub and each agent. This will help you manage the state in agents and hub.
- Start with the simple agent
- Consider starting with the simple case of just one round, before trying to scale it out to multiple rounds.
- The agents are pretty stupid, they just need to generate guesses and track/display the information they receive from the hub.
- The Unix `tee` command can be used to duplicate input - for debugging purposes you can wrap your agent inside a shell script and use `tee` to save the communications messages to a file. We'll talk about this in the pracs.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on Piazza or emailed to `csse2310@helpdesk.eait.uq.edu.au`.

v1.1

- Correct sequence number references in Hub Operation / gameplay description
- Add protocol diagram
- Clarify handling of numeric values
- Misc typo fixups