The University of Queensland
School of Electrical Engineering and Computer Science

CSSE2310 – Semester 2, 2025
Assignment 4 (Version 1.1.1)

Marks: 75
Weighting: 15%
Due: 3:00pm Friday 31 October, 2025

This specification was created for the use of Gia Hung HUYNH (s4938484) only.
Do not share this document. Sharing this document may result in a misconduct penalty.
Specification changes since version 1.0 are shown in blue and are summarised at the end of the document.

Introduction

The goal of this assignment is to further develop your C programming skills and to demonstrate your understanding of networking and multithreaded programming. Your task is to write a networked card game. This will require two programs: a client (called `ratsclient`) and a server (called `ratsserver`). The server supports multiple simultaneously connected clients across multiple games. The client is to validate and send user-selected cards to the server.

Communication between the clients and `ratsserver` is over TCP using a defined communication protocol. Advanced functionality such as connection limiting, signal handling and statistics reporting is also required for full marks.

The assignment will also test your ability to code to a particular programming style guide and to use a revision control system appropriately.

Student Conduct

This section is unchanged from assignments one and three – but you should remind yourself of the referencing requirements. Remember that you can't copy code from websites and if you learn about how to use a library function from a resource other than course-provided material then you must reference it.

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 the discussion forum. 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 or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another person'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 actions will be initiated against you, and those you cheated with. If you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

You must follow the following code usage and referencing rules for **all code committed to your SVN repository** (not just the version that you submit):

Code Origin	Usage/Referencing
Code provided by teaching staff this semester Code provided to you in writing **this semester** by CSSE2310 teaching staff (e.g., code hosted on Blackboard, found in `/local/courses/csse2310/resources` on `moss`, posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class).	**Permitted** May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.)
Code you wrote this semester for this course Code you have personally written this semester for CSSE2310 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	**Permitted** May be used freely without reference. (This assumes that no reference was required for the original use.)
Unpublished code you wrote earlier Code you have personally written in a previous enrolment in this course or in another UQ course or for other reasons and where that code has not been shared with any other person or published in any way. **Code from `man` pages on `moss`** Code examples found in `man` pages on `moss`. (This does not apply to code from `man` pages found on other systems or websites unless that code is also in the `moss` `man` page.)	**Conditions apply, references required** May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
Code and learning from AI tools Code written by, modified by, debugged by, explained by, obtained from, or based on the output of, an artificial intelligence tool or other code generation tool that you alone personally have interacted with, without the assistance of another person. This includes code you wrote yourself but then modified or debugged because of your interaction with such a tool. It also includes code you wrote where you learned about the concepts or library functions etc. because of your interaction with such a tool. It also includes where comments are written by such a tool – comments are part of your code.	**Conditions apply, references & documentation req'd** May be used provided you understand the code AND the source of the code or learning is referenced in a comment adjacent to that code (in the required format – see the style guide) AND an ASCII text file (named `toolHistory.txt`) is included in your repository and with your submission that describes in detail how the tool was used. (All of your interactions with the tool must be captured.) The file must be committed to the repository at the same time as any code derived from such a tool. If such code is used without appropriate referencing and without inclusion of the `toolHistory.txt` file then this will be considered misconduct. See the detailed AI tool use documentation requirements on Blackboard – this tells you what must be in the `toolHistory.txt` file.
Code copied from sources not mentioned above Code, in any programming language: • copied from any website or forum (including StackOverflow and CSDN); • copied from any public or private repositories; • copied from textbooks, publications, videos, apps; • copied from code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); • written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member); • written by an AI tool that you did not personally and solely interact with; • written by you and available to other students; or • from any other source besides those mentioned in earlier table rows above.	**Prohibited** May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken. This prohibition includes code written in other programming languages that has been converted to C.
Code that you have learned from Examples, websites, discussions, videos, code (in any programming language), etc. that you have learned from or that you have taken inspiration from or based any part of your code on but have not copied or just converted from another programming language. This includes learning about the existence of and behaviour of library functions and system calls that are not covered in class.	**Conditions apply, references required** May be used provided you do not directly copy code AND you understand the code AND the source of the code or inspiration or learning is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.

You must not share this assignment specification with any person (other than course staff), organisation, website, etc. Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

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. Don't help another CSSE2310/7231 student with their code no matter how desperate they may be and no matter how close your relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: `https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct`.

The Rats Game

The Rats Casino, located in the fictional city of Brisvegas, is preparing a mitigation strategy for potential future COVID-related disruptions that may require patrons to remain at home during lockdowns. To maintain engagement during such scenarios, the casino plans to develop an online multiplayer card game that allows patrons to enjoy the casino experience remotely. As a freelance software developer, you have been commissioned by the Rats Casino to create a text-based proof-of-concept prototype consisting of two components: a client program named `ratsclient` and a server program named `ratsserver`. These programs will implement the casino's novel card game, "Rats", in an online environment.

A game requires four players in positions P1, P2, P3 and P4. The positions will be assigned by giving P1 to the player with the earliest name in lexicographic order (as determined by `strcmp()`). For example, if Harry, Hermione, Ron and Ginny connect to the game, then Ginny will be P1, Harry will be P2, Hermione will be P3 and Ron will be P4. Player names are not required to be unique, but if multiple players with the same name are in a game, the order they are placed in is up to you. The game is played in teams, so P1 and P3 form one team, while P2 and P4 form the other.

The deck for the game consists of four suits: **S** (Spades), **D** (Diamonds), **C** (Clubs) and **H** (Hearts). Each suit has 13 ranks, with the order from highest to lowest rank being: **A** (Ace), **K** (King), **Q** (Queen), **J** (Jack), **T** (10), **9**, **8**, **7**, **6**, **5**, **4**, **3**, **2**. Individual cards are written rank first, e.g. **AD** (Ace of Diamonds), **9S** (9 of Spades) and **JH** (Jack of Hearts).

Definitions

Play a trick – where each player (in order) contributes one card.

Win a trick – to play a trick and contribute the highest card in it.

Lead – to play the first card in a trick. The suit of the card determines which suit the other players must play (if they can). To have the lead means that the user will play first.

Hand – a set of 13 cards which are then played in 13 tricks.

Following suit – to play a card which has the same suit as the card that was led. A card that does not follow suit cannot win the trick.

Game Flow

The game has the following stages:

1. Each player is dealt a hand of 13 cards.

2. Tricks: each player plays one card and the player with the highest card wins the trick.
3. Repeat 2 until all cards have been played.
4. The team that wins the most tricks wins the game.

Communication Protocol

All protocol messages from `ratsserver` to clients consist of single lines of text. The first character in the line determines the type of the message. See Table 1.

Table 1: The communication protocol (messages the server sends to clients).

First Character	Purpose	Format	Result
M	Send an "info" message from the server to client	`M%s`	Client prints "`Info: %s`"
H	Give player their hand of cards	`H%s`	
L	Lead a card	`L`	Client asks user for a card to play
P	Play a card following suit if possible	`P%c`	Client asks the user for a card. (The %c indicates the suit)
A	Informs the client that their lead or play was accepted	`A`	Client removes the card from their hand
O	Game is over	`O`	

Messages from the client to the server are in the form `%c%c\n`, indicating the card to play. If the client sends an invalid message to the server, the server will treat that message like it were an invalid play and ask the client to play a card again.

"M" messages relating to lead and play should be sent to all players **except** the one who caused the message. That is, the user does not hear messages about their own plays or disconnection. Announcements about winning tricks still go to all players.

Specification - `ratsclient`

The `ratsclient` program provides an interface that allows a user to interact with the server (`ratsserver`) as a client – connecting, sending cards and receiving responses from the server. Your `ratsclient` will not require multiple threads or processes. Error messages and exit statuses are summarised in Table 2 and explained throughout this specification.

Table 2: `ratsclient` exit statuses and messages.

Condition	Status	Message to `stderr`
Incorrect number of arguments	3	`Usage: ./ratsclient clientname game port`
Empty string for player name, game name or port	20	`ratsclient: invalid arguments`
Can't connect to the server	5	`ratsclient: unable to connect to the server`
Server doesn't follow protocol	7	`ratsclient: a protocol error occurred`
User EOF	17	`ratsclient: user has quit`
Game over	0	

Command Line Arguments

Your `ratsclient` program is to accept command line arguments as follows:

`./ratsclient` *`clientname game port`*

The *italics* indicate placeholders for user-supplied arguments. The *`clientname`* argument must always be the first argument, followed by *`game`* and then *`port`*.

Below is an example of how the program might be run [1]:

```
./ratsclient Harry ExplodingSnap 2310
```

The meaning of the arguments is as follows:

- *`clientname`* – this mandatory argument specifies the name of the player connecting to the server. There is no requirement that player names be unique, i.e. two clients could connect to the same game using the same name.
- *`game`* – this mandatory argument specifies the name of the game that the player wishes to join (or create if it hasn't been created).
- *`port`* – this mandatory argument specifies which `localhost` port the server is listening on – either numerical **or** the name of a service.

Prior to doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line, then it must print the (single line) message:

```
Usage: ./ratsclient clientname game port
```

to standard error (with a following newline), and exit with an exit status of 3.

Invalid command lines include the following:

- There are missing arguments.
- An unexpected argument is present.

Argument Checking

If any command line argument is the empty string, then `ratsclient` must print the message:

```
ratsclient: invalid arguments
```

to standard error (with a following newline), and exit with an exit status of 20.

Checking whether the *`port`* is a valid port or service name is not part of this checking (other than checking that it is not the empty string). The validity is checked in the section below.

Port Checking

If `ratsclient` is unable to connect to the server on the specified port (or service name) of `localhost`, it shall emit the following message (terminated by a newline) to `stderr` and exit with status 5:

```
ratsclient: unable to connect to the server
```

[1] This is a single example, not an exhaustive list. The example assumes that a `ratsserver` server is running on the given port.

Runtime Behaviour

Assuming that the checks above pass and your `ratsclient` can successfully connect to the server, then it is to perform the following actions.

`ratsclient` should send the user's name and game name as soon as it connects to `ratsserver`. The client informs the server of the user's name and requested game name by sending them to the server on individual lines, e.g. `Harry\nExplodingSnap\n`

At the beginning of the game, the client will print the hand after receiving it from the server (it can be assumed to be valid). Whenever the client displays the current hand, it should be in the following format:

- `S:` followed by the ranks of any **S** cards (in decreasing order) separated by spaces.
- `C:` followed by the ranks of any **C** cards (in decreasing order) separated by spaces.
- `D:` followed by the ranks of any **D** cards (in decreasing order) separated by spaces.
- `H:` followed by the ranks of any **H** cards (in decreasing order) separated by spaces.

So a hand might be shown as:

```
S: A K Q J T
C: 9 8 7 6 2
D:
H: 8 7 6
```

In successive tricks, cards that have been played should be removed. Do not remove a card until the server has sent an "A" message.

When the game starts, the client will wait for the server to ask it to play. If it has the lead, then the hand should be displayed followed by the prompt:

```
Lead>
```

If the client does not have the lead, then it will display the hand followed by the prompt:

```
[Y] play>
```

where Y is the lead suit. All client prompts with ">" should have a space after ">", e.g. "`Lead> `". `ratsclient` keeps displaying the hand and prompt to the user until a valid card is chosen. Note that for a card to be valid, it must be in the user's hand **and** belong to the lead suit (if possible). `ratsclient` needs to perform these checks, rather than relying on the server to perform these checks. `ratsserver` will also need to perform these checks, and not assume that cards sent from clients are valid.

This sequence repeats until the server advises that the game is over. If the client receives an "O" message at any time, it should end the game. At that point, the client should exit with status 0.

If an unexpected message is received from the server, `ratsclient` is to print the following message to `stderr` and exit with status 7:

```
ratsclient: a protocol error occurred
```

Unexpected disconnects from the server or the server not ending the game when all cards have been played will be treated as the server not following protocol.

If `ratsclient` receives an EOF on `stdin` when expecting the user to play a card, `ratsclient` is to print the following message to `stderr` and exit with status 17:

```
ratsclient: user has quit
```

Messages are another aspect of the client interface. At various times, the client will expect a message from

the server. For example, a new hand. At any point where a message is expected, the server can send an info message. When this happens, `ratsclient` prints the following to `stdout`:

```
Info: <text of message>
```

and continues to wait for another message from the server. This mechanism is used by the server to notify clients of various events. The client must receive the initial greeting "M" message before any other messages. Apart from this, no "M" message should be "required" by the client. However, `ratsclient` should handle any valid "M" message regardless of when it arrives.

Other Requirements

Your `ratsclient` program must free all memory before exiting. For those aspects of the client behaviour not specified here, your program must exactly match the behaviour of `demo-ratsclient` on `moss`.

Specification - `ratsserver`

`ratsserver` is a networked, multithreaded Rats game server allowing clients to connect, send the user's name and game name, and play the Rats game. All communication between clients and the server is over TCP using a message format that is described in the Communication Protocol section. Error messages and exit statuses are summarised in Table 3 and explained throughout this specification. Note that `ratsserver` should not terminate under normal circumstances, nor should it block or otherwise attempt to handle `SIGINT`. `ratsserver` must not exit in response to a `SIGPIPE`.

Table 3: `ratsserver` exit statuses and messages.

Condition	Status	Message to `stderr`
~~Incorrect number of~~ Invalid arguments	16	`Usage: ./ratsserver maxconns greeting [portnum]`
~~Invalid port~~	~~1~~	~~`ratsserver: port invalid`~~
Error listening on the game server port	6	`ratsserver: unable to listen on given port "`*N*`"`
System error (this will not be tested)	3	`ratsserver: system error`

Command Line Arguments

Your `ratsserver` program is to accept command line arguments as follows:

`./ratsserver` *`maxconns greeting`* `[`*`portnum`*`]`

The square brackets (`[]`) indicate an optional argument. The *italics* indicate placeholders for user-supplied arguments. Arguments must appear in this order.

Some examples of how the program might be run include the following:

```
./ratsserver 5 Welcome
```

```
./ratsserver 0 "Welcome to Rats" 2310
```

```
./ratsserver 100 hello 1234
```

The meaning of the arguments is as follows:

- *maxconns* – this argument must be specified and is expected to be a non-negative integer less than or equal to 10,000 specifying the maximum number of simultaneous client connections to be permitted. If this is zero, then there is no limit to how many clients may connect (other than operating system limits, which will not be tested).

- *greeting* – this argument must be specified and is a string that the server will send to game clients that connect.

- *portnum* – if specified, this argument is a string which specifies which localhost port the game server is to listen on. This can be either numerical or the name of a service. If this is zero or this argument is absent, then `ratsserver` is to use an ephemeral port.

Important: Even if you do not implement the connection limiting functionality, your program must correctly handle command lines that include this argument (after which it can ignore any provided value – you will simply not receive any marks for that feature).

Prior to doing anything else, your program must check the command line arguments for validity. If the program receives an invalid command line, then it must print the (single line) message:

```
Usage: ./ratsserver maxconns greeting [portnum]
```

to standard error (with a following newline), and exit with an exit status of 16.

Invalid command lines include (but may not be limited to) any of the following:

- The *maxconns* argument is given but it is not a non-negative integer less than or equal to 10,000. A leading `+` sign is permitted (optional). Numbers with leading zeroes will not be tested, i.e. may be accepted or rejected.

- ~~The *portnum* option argument is given but it is not a non-empty string argument.~~

- An unexpected argument is present.

- Any argument is the empty string.

Checking whether *portnum* is a valid port or service name is not part of the usage checking (other than checking that the value is not empty). The validity of this value is checked after command line validity, as described below.

Port Checking

If `ratsserver` is unable to listen on the given port (or service name) of `localhost`, it shall output the following message (terminated by a newline) to `stderr` and exit with status 6:

```
ratsserver: unable to listen on given port "N"
```

where *N* should be replaced by the argument given on the command line. (This may be a non-numerical string.) The double quotes must be present. Being "unable to listen on a given port" includes the cases where the socket can't be created, the port string is invalid, the socket can't be bound to the address, or the socket can't be listened on. Note that we will not test the situation where `ratsserver` is unable to listen on an ephemeral port.

Runtime Behaviour

Once the port is opened for listening, `ratsserver` shall print to `stderr` the port number (not the service name) followed by a single newline character, and then flush the output. **In the case of ephemeral ports, the actual port number obtained shall be printed, not zero**.

Upon receiving an incoming client connection on the port, `ratsserver` shall spawn a new thread to handle that client. The thread sends a greeting and waits for the client to send the user's name and the name of the

game they wish to play in. `ratsserver` should send the greeting as soon as each client connects. Therefore, both client and server should send their initial info before attempting to read. If the game does not exist, it is created and the client is added to the game. If the game exists, the client is added to the game. If a game does not yet have four players, the thread will terminate.

When a game fills up, the thread that handled the most recently connected player should run that game (see Figure 1). For the sequence of actions of a game thread, see Figure 2. This thread **must** call the function `get_random_deck()` to get a random deck of cards (see libcsse2310a4). A deck is represented by a line of 104 characters. It will indicate the order the cards are to be dealt. The first card will be dealt to P1, the second to P2, the third to P3, the fourth to P4, the fifth to P1, etc.

At the beginning of the game, **P1** will lead.

The server sends informational messages to clients using "M" messages in the following circumstances:

- Send the welcome message to a client when it connects.
- The game begins (send messages to all clients in this order).
  - The first message text is: "`Team 1: %s, %s`" (substitute the names of P1 and P3).
  - The second message text is: "`Team 2: %s, %s`" (substitute the names of P2 and P4).
  - After dealing cards to the players via an "H" message, the server then sends all players: "`Starting the game`".
- When a player wins a trick: "`%s won`" (substitute player name).
- When a game ends (13 tricks have been played) without any clients disconnecting early, the winning team (most tricks won) will be announced: "`Winner is Team %d (%d tricks won)`". For example: "Winner is Team 2 (10 tricks won)". After this, an "O" message is sent to all players in the game.
- When another player disconnects early: "`%s` disconnected early". For example: "Harry disconnected early". After this, an "O" message is sent to all players in the game.
- When another player (legally) plays a card: "`%s plays %c%c`" (substitute player name and card).

Information messages should be sent to clients before the next move or lead message. No messages should be sent to clients after they have been sent a game over "O" message. On receiving an "O" message, the client will disconnect.

Client disconnections will be handled as follows. Regardless of when the server detects the disconnection, it should not be acted upon until the server needs to read from that client. That is, when a read call involving that client fails. So if the server is waiting for P3 to send back a move and P1 disconnects first, then P3 disconnects, P3 will be regarded as the disconnecting player. Other clients will be sent the message specified above, then a game over message. After that, all clients in the game will be disconnected and the client handler thread will clean up and terminate. Other client threads and the `ratsserver` program itself must continue uninterrupted. Note that this happens even if clients disconnect before the game starts. Their positions do not get filled by later connections.

If connection limiting behaviour is implemented, then `ratsserver` must keep track of how many active client connections exist, and must not let that number exceed the ***`maxconns`*** parameter. See below for more details on how this limit is to be implemented.

Note that your `ratsserver` must be able to deal with any clients using the correct communication protocol, not just the client programs specified for the assignment. Testing with `netcat` is highly recommended.

Since multiple clients can connect to the server at the same time, your structure(s) containing information about each game must be protected with a semaphore or mutex to ensure that they do not get corrupted.

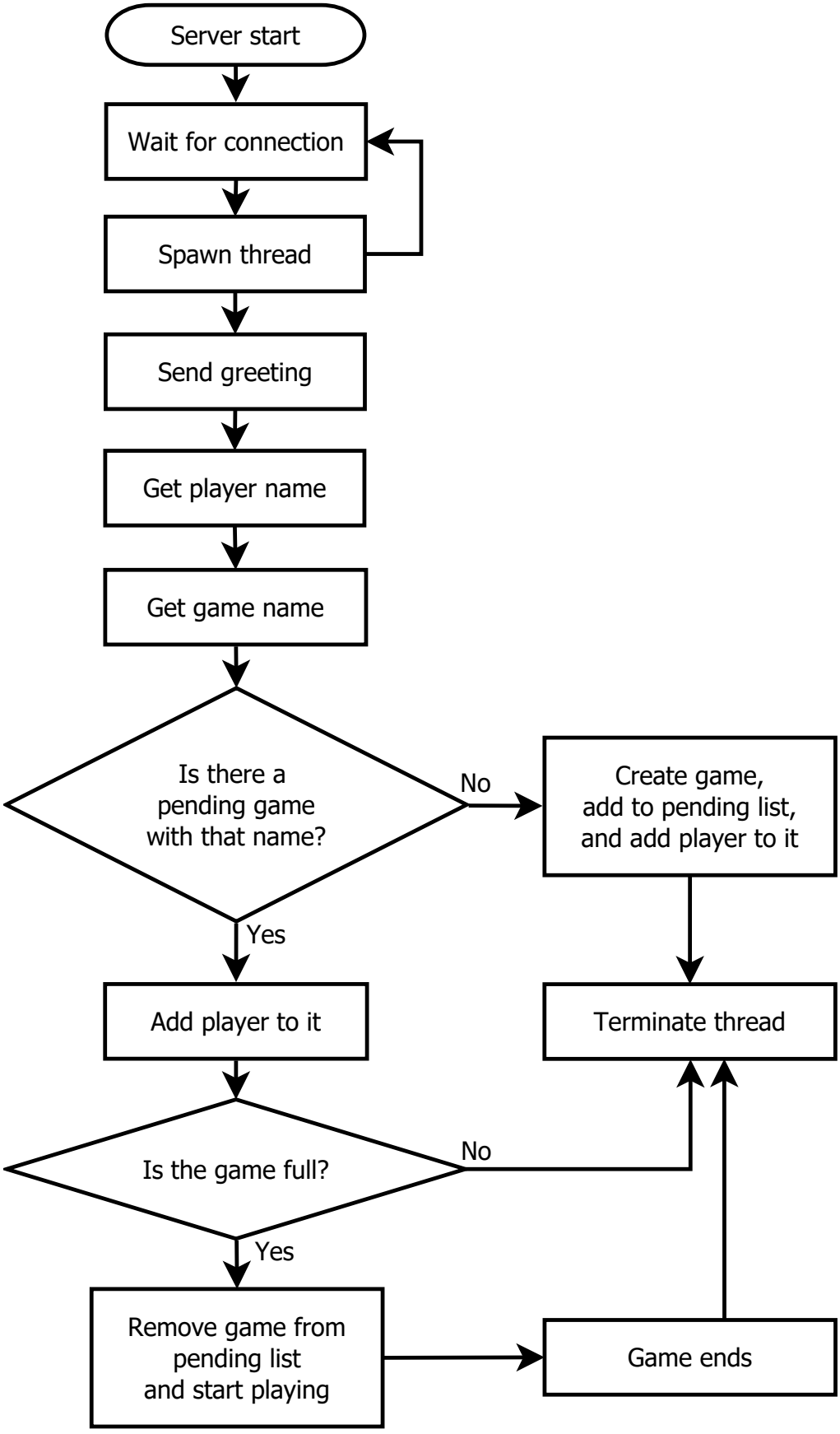

Figure 1: Server activities on connection.

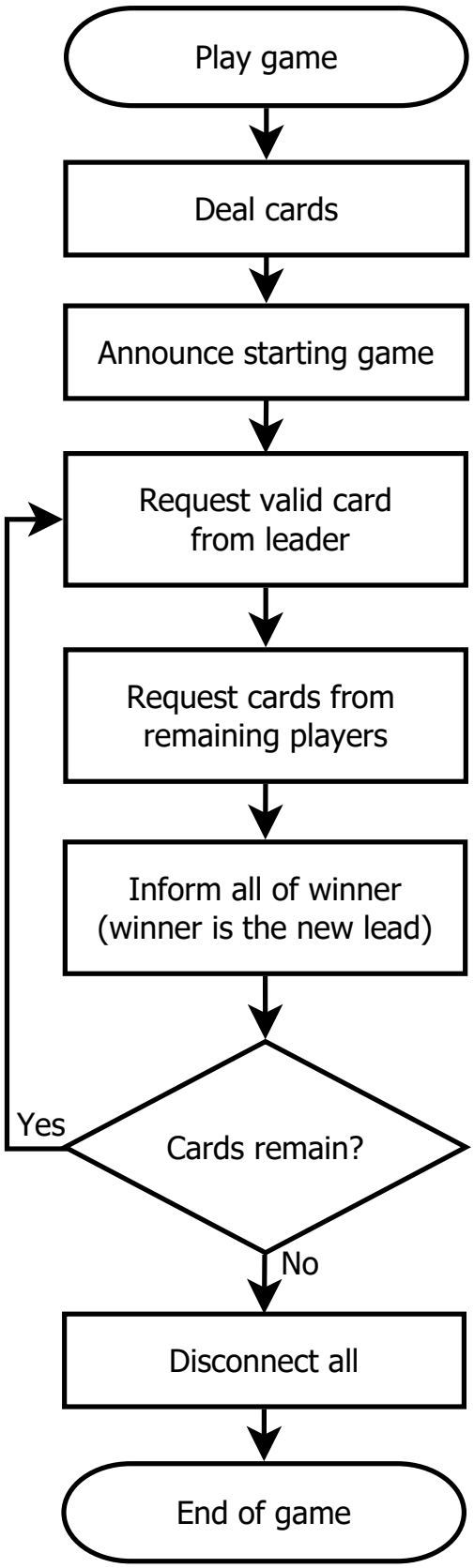

Figure 2: Flow chart for server thread managing a single game.

SIGHUP Handling (Advanced)

Upon receiving `SIGHUP`, `ratsserver` is to output (and flush) to `stderr` statistics reflecting the program's operation to date, specifically:

- The number of clients connected (at this instant)
- The total number of clients that have connected since the program start
- The number of games running (at this instant)
- The number of games that ran to completion
- The number of games that terminated early
- The total number of tricks played across all games

The required statistics format is illustrated below. Each of the six lines is terminated by a single newline. You can assume that all numbers will fit in a 32-bit unsigned integer, i.e. you do not have to consider numbers larger than 4,294,967,295.

Example 1: `ratsserver` SIGHUP `stderr` output sample

```
Connected players: 20
Total num players connected: 60
Num games running: 5
Games completed: 4
Games terminated: 6
Total tricks played: 67
```

Note that to accurately gather these statistics and avoid race conditions, you will need some sort of mutual exclusion that protects the variables holding these statistics from corruption.

Global variables are NOT to be used to implement signal handling (or for any other purposes in this assignment). See the Hints section below for how you can implement signal handling.

Client Connection Limiting (Advanced)

If the *`maxconns`* feature is implemented and a non-zero command line argument is provided, then `ratsserver` must not permit more than that number of simultaneous client connections to the server. If a client beyond that limit attempts to connect, `ratsserver` shall block, indefinitely if required, until another client leaves and this new client's connection request can be `accept()`ed. Clients in this waiting state are not to be counted in statistics reporting – they are only counted once they have properly connected.

Other Requirements

Other than error messages, the listening port number, and `SIGHUP`-initiated statistics output, `ratsserver` is not to emit any output to `stdout` or `stderr`.

Your server must not busy wait. If a thread has nothing to do, then it must be blocking, e.g. in `accept()`, and not sleeping or busy waiting.

Your server must free memory that is no longer needed, i.e., not allow memory use to grow over time (excluding memory allocated within that you have no control over).

For those aspects of the server behaviour not specified here, your program must exactly match the behaviour of `demo-ratsserver` on `moss`.

Provided Library

libcsse2310a4

The following functions are declared in `/local/courses/csse2310/include/csse2310a4.h` on `moss` and their behaviour and required compiler flags are described in man pages on `moss`.

This function **must** be used by `ratsserver` to get a random deck of cards when each new game is started.

```
char* get_random_deck(void);
```

These functions may be used to read a line from a given stream or file descriptor.

```
char* read_line_stream(FILE* stream);
```

```
char* read_line_fd(int fd);
```

Testing

As well as testing your client with the demo server (`demo-ratsserver`) and your server with the demo client (`demo-ratsclient`) on `moss`, remember that you can use `netcat` to simulate and capture requests/responses. In addition to this, you may find `2310netproxy` useful for this purpose (refer to the man page on `moss`).

A test script will be provided on `moss` that will test your program against a subset of the functionality requirements – approximately 50% of the available functionality marks. The script will be made available about 7 to 10 days before the assignment deadline and can be used to give you some confidence that you're on the right track. The "public tests" in this test script will not test all functionality and you should be sure to conduct your own tests based on this specification. The "public tests" will be used in marking, along with a set of "private tests" that you will not see.

The Gradescope submission site will also be made available about 7 to 10 days prior to the assignment deadline. Gradescope will run the test suite immediately after you submit. When this is complete[2] you will be able to see the results of the "public tests". You should check these test results to make sure your program is working as expected. Behaviour differences between `moss` and Gradescope may be due to memory initialisation assumptions in your code, so you should allow enough time to check (and possibly fix) any issues after submission.

Style

Your program must follow version 3.06 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site. Your submission must also comply with the *Documentation required for the use of AI tools* if applicable.

Hints

1. The multithreaded network server example from the lectures can form the basis of `ratsserver`.
2. Review the lectures and sample code related to network clients, threads and synchronisation, and multithreaded network servers. This assignment builds on all of these concepts.

[2] Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality and efficiency of your code.

3. You can test `ratsserver` and `ratsclient` independently using `netcat`. You can also use the provided demo programs `demo-ratsclient` and `demo-ratsserver` to (1) understand the expected functionality and communication protocol, and (2) test your client with the demo server and vice versa. Use `netcat` as a server or client to see what messages the demo client and server send. `2310netproxy` can also be used (refer to the man page on `moss`).

4. Use the provided library functions and example code (see above).

5. Consider a dedicated signal handling thread for `SIGHUP`. `pthread_sigmask()` can be used to mask signal delivery to threads, and `sigwait()` can be used in a thread to block until a signal is received. You will need to do some research and experimentation to get this working. Be sure to properly reference any code samples or inspiration you use.

6. Remember to `fflush()` output that you `printf()`, `fprintf()` or `fwrite()`. Output to network sockets via `FILE*` handles is not newline buffered.

7. Some functions which **may** be useful include: `qsort()`, `dprintf()`, `fflush()`, `pthread_sigmask()`, `sigemptyset()`, `sigaddset()`, `exit()`, `read()` and `write()`. You should consult the man pages of these functions.

8. Since both `ratsserver` and `ratsclient` use the same communication protocol and are required to perform similar checks on cards played, functions related to the protocol and card checking should be implemented in a common `.c` (and `.h`) file and not duplicated in the client and server.

Possible Approach

1. Try implementing `ratsclient` first. (The programs are independent so this is not a requirement, but when you test it with `demo-ratsserver`, it may give you a better understanding of how `ratsserver` works.)

2. For `ratsserver`, start with the multithreaded network server example from the lectures, gradually adding functionality for supported operations.

3. Code your server to run a single game at a time but put as much code/data in functions and structs as possible so that you can handle multiple games later by spawning multiple threads.

Forbidden Functions

You must not use any of the following C statements/directives/etc. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `#pragma`
- `gcc` attributes (other than the possible use of `__attribute__((unused))` as described in the style guide)

You must not use any of the following C functions. If you do so, you will get zero (0) marks for any test case that calls the function.

- `longjmp()` and equivalent functions
- `system()`
- `mkfifo()` or `mkfifoat()`

- `fork()`, `pipe()`, `popen()`, `execl()`, `execvp()` and other `exec` family members.
- `pthread_cancel()`
- `sleep()`, `usleep()`, `nanosleep()` or any other function that involves a sleep, alarm or timeout.
- `signal()`, `sigpending()`, `sigqueue()`, `sigtimedwait()`, `sigwaitinfo()`, `sigsuspend()`
- Functions described in the man page as nonstandard, e.g. `strcasestr()`. Standard functions will conform to a POSIX standard – often listed in the "CONFORMING TO" section of a man page. Note that `getopt_long()` and `getopt_long_only()` are an exception to this – these functions are permitted if desired.

The use of comments to control the behaviour of `clang-format` and/or `clang-tidy` (e.g., to disable style checking) will result in zero marks for automatically marked style.

Submission

Your submission must include all source and any other required files (in particular, you must submit a `Makefile`). Do not submit compiled files (e.g. `.o`, compiled programs) or test input files.

Your programs (named `ratsclient` and `ratsserver`) must build on `moss.labs.eait.uq.edu.au` and in the Gradescope environment with:

```
make
```

Make sure your default target builds both programs! Your program must be compiled with `gcc` with at least the following options:

```
-Wall -Wextra -pedantic -std=gnu99
```

You may of course use additional flags (e.g. `-pthread`). You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

If any errors result from the `make` command (e.g. an executable can not 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.

Your assignment submission must be committed to your Subversion repository under

```
svn+ssh://source.eait.uq.edu.au/csse2310-2025-sem2/csse2310-s4938484/trunk/a4
```

Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a4` directory in your repository (and not within a subdirectory) 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.

To submit your assignment, you must run the command

```
2310createzip a4
```

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made

available in the Assessment area on the CSSE2310/7231 Blackboard site)[3]. The zip file will be named

`s4938484_csse2310_a4_`*`timestamp`*`.zip`

where *`timestamp`* is replaced by a timestamp indicating the time that the zip file was created.

The `2310createzip` tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command '`make`', and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process to check out your submission from your repository. You will be asked to confirm references in your code and also to confirm your use (or not) of AI tools to help you.

You must not create the zip file using some other mechanism and you must not modify the zip file before submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of the creation of your submission zip file.

Multiple submissions to Gradescope are permitted. We will mark whichever submission you choose to "activate" – which by default will be your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline[4] will incur a late penalty – see the CSSE2310 course profile for details.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you attend an interview about your assignment and you are unable to adequately respond to questions – see the CSSE2310 Student Interviews section below.

Functionality (60 Marks)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has been generated correctly and has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Not all features are of equal difficulty. Partial marks will be awarded for partially meeting the functionality requirements. A number of tests will be run for each marking category listed below. Your mark in that category will be proportional (or approximately proportional) to the number of tests passed in that category.

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 client can never create a connection to a server, then we can not determine whether it sends the correct messages to the server. Your tests must run in a reasonable time frame, which could be as short as a few seconds for usage checking to many tens of seconds in some cases. If your program takes too long to respond, then it will be terminated and you will earn no marks for the functionality associated with that test.

Exact text matching of files and output (`stdout` and `stderr`) and communication messages is used for functionality marking. Strict adherence to the formats in this specification is critical to earn functionality marks.

The markers will make no alterations to your code (other than to remove code without academic merit). Note that your client and server will be tested independently. Note that later categories may contain earlier categories. For example, "multiple games" categories assume that you can handle early disconnects.

Marks will be assigned in the following categories. There are 20 marks for `ratsclient` and 40 marks for

[3] You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck to download the zip file to your local computer and then upload it to the submission site.

[4] or your extended deadline if you are granted an extension.

`ratsserver`.

1. `ratsclient` correctly handles invalid command lines (2 marks)
2. `ratsclient` connects to the server and also handles inability to connect to the server (2 marks)
3. `ratsclient` handles valid command line arguments by sending correct messages to the server (2 marks)
4. `ratsclient` correctly handles input from the user and responses received from the server (5 marks)
5. `ratsclient` plays a single game (5 marks)
6. `ratsclient` correctly handles communication failure with server (includes handling `SIGPIPE` when writing to the socket) (2 marks)
7. `ratsclient` frees all memory before exiting (2 marks)

8. `ratsserver` correctly handles invalid command lines (2 marks)
9. `ratsserver` correctly listens for connections and reports the port (including inability to listen for connections) (2 marks)
10. `ratsserver` correctly plays a single game (6 marks)
11. `ratsserver` correctly plays a single game with invalid plays (5 marks)
12. `ratsserver` correctly plays multiple games sequentially (5 marks)
13. `ratsserver` correctly plays concurrent games (including protecting data structures with mutexes or semaphores) (6 marks)
14. `ratsserver` correctly handles disconnecting clients and communication failure (including not exiting due to `SIGPIPE`) (3 marks)
15. `ratsserver` correctly implements client connection limiting (3 marks)
16. `ratsserver` correctly implements `SIGHUP` statistics reporting (including protecting data structures with mutexes or semaphores) (6 marks)
17. `ratsserver` does not leak memory and does not busy wait (2 marks)

Some functionality may be assessed in multiple categories. The ability to support multiple simultaneous clients will be covered in multiple categories, as will the ability to support multiple requests per client. Multiple categories will include checks that the correct number of threads are created in handling requests (one additional thread per connected client).

Style Marking

> Text below this point is unchanged from assignment three (other than any specification updates at the end). You should still make sure that you are familiar with all of the requirements below.

Style marking is based on the number of style guide violations, i.e. the number of violations of version 3.06 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below. Your style marks 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 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.

You are encouraged to use the `2310reformat.sh` and `2310stylecheck.sh` tools installed on `moss` to correct and/or check your code style before submission. The `2310stylecheck.sh` tool does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All `.c` and `.h` files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not[5].

Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your `.c` and `.h` files as follows. If any of your submitted `.c` and/or `.h` files are unable to be compiled by themselves then your automated style mark will be zero (0). If your code uses comments to control the behaviour of `clang-format` and/or `clang-tidy` then your automated style mark will be zero. If any of your source files contain C functions longer than 100 lines of code[6] then your automated and human style marks will both be zero. If you use any global variables then your automated and human style marks will both be zero.

If your code does compile and does not contain any C functions longer than 100 lines and does not use any global variables and does not interfere with the expected behaviour of `clang-format` and/or `clang-tidy` then your automated style mark will be determined as follows: Let

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)
- A be the total number of style violations detected by `2310stylecheck.sh` when it is run over each of your `.c` and `.h` files individually[7].

Your automated style mark S will be

$$S = 5 - (W + A)$$

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. If you believe that `2310stylecheck.sh` is behaving incorrectly or inconsistently then please bring this to the attention of the course coordinator prior to submission, e.g., it is possible the style checker may report different issues on moss than it does in the Gradescope environment. Your automated style mark can be updated if this is deemed to be appropriate. You can check the result of Gradescope style marking soon after your Gradescope submission – when the test suite completes running.

Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for "comments", "naming" and "modularity". Note that if your code contains any functions longer than 100 lines or uses a global variable then your human style mark is zero and the criteria/standards below are not relevant.

The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide.

[5] Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

[6] Note that the style guide requires functions to be 50 lines of code or fewer. Code that contains functions whose length is 51 to 100 lines will be penalised somewhat – one style violation (i.e. one mark) per function. Code that contains functions longer than 100 lines will be penalised very heavily – no marks will be awarded for human style or automatically marked style.

[7] Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file. You can check that a header file compiles by itself by running `gcc -c` *`filename.h`* – with any other `gcc` arguments as required.

Comments (3 marks)

Mark	Description
0	25% or more of the comments that are present are inappropriate AND/OR at least 50% of the required comments are missing
1	At least 50% of the required comments are present AND the vast majority (75%+) of comments present are appropriate AND the requirements for a higher mark are not met
2	All or almost all required comments are present AND all or almost all comments present are appropriate AND the requirements for a mark of 3 are not met
3	All required comments are present AND all comments present are appropriate AND additional comments are present as appropriate to ensure clarity

Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1	All names used are appropriate

Modularity (1 mark)

Mark	Description
0	There are two or more instances of poor modularity (e.g. repeated code blocks)
0.5	There is one instance of poor modularity (e.g. a block of code repeated once)
1	There are no instances of poor modularity

SVN Commit History Marking (5 marks)

Markers will review your SVN commit history for your assignment up to your zip file creation time. 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). Progressive development is expected, i.e., no large commits with multiple features in them.
- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality, not in terms of specific numbered test cases in the test suite) and/or why the change has been made and will be usually be more detailed for significant changes.).

The standards expected are outlined in the following rubric. The mark awarded will be the highest for which the relevant standard is met.

Mark (out of 5)	Description
0	Minimal commit history – only one or two commits OR all commit messages are meaningless.
1	Some progressive development evident (three or more commits) AND at least one commit message is meaningful.
2	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful
3	Multiple commits that show progressive development of almost all or all functionality AND at least two-thirds of the commit messages are meaningful.
4	Multiple commits that show progressive development of ALL functionality AND meaningful messages for all but one or two of the commits OR Multiple commits that show progressive development of almost all functionality AND meaningful messages for ALL commits
5	Multiple commits that show progressive development of ALL functionality AND meaningful messages for ALL commits.

Total Mark

Let

- F be the functionality mark for your assignment (out of 60 for CSSE2310 students).
- S be the automated style mark for your assignment (out of 5).
- $A = F + \min\{F, S\}$ (the automatically determined mark for your assignment).
- H be the human style mark for your assignment (out of 5).
- C be the SVN commit history mark (out of 5).
- V be the scaling factor (0 to 1) determined after interview (if applicable – see the CSSE2310 Student Interviews section below) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

$$M = (A + \min\{A, H\} + \min\{A, C\}) \times V$$

out of 75 (for CSSE2310 students)

In other words, you can't get more marks for automated style than you do for functionality. Similarly, you can't get more marks for human style or SVN commit history than you do for functionality and automated style combined. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

CSSE2310 Student Interviews

This section is unchanged from assignments one and three.

The teaching staff will conduct interviews with a subset of CSSE2310 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. If you legitimately use code from other sources (following the usage/referencing requirements outlined in this assignment, the style guide, and the AI tool use documentation requirements) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment unless there are documented exceptional circumstances that prevent you from attending.

Students will be selected for interview based on a number of factors that may include (but are not limited to):

- Feedback from course staff based on observations in class, on the discussion forum, and during marking;
- An unusual commit history (versions and/or messages), e.g. limited evidence of progressive development;
- Variation of student performance, code style, etc. over time;
- Use of unusual or uncommon code structure/functions etc.;
- Referencing, or lack of referencing, present in code;
- Use of, or suspicion of undocumented use of, artificial intelligence or other code generation tools; and
- Reports from students or others about student work.

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 the discussion forum.

Version 1.1

- Clarified the error conditions that `ratsserver` should handle and the exit statuses and error messages that should be output – see Table 3.

Version 1.1.1

- Added five more forbidden functions – see line 359.
- Removed redundant line – see line 195.