The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 1, 2022
Assignment 3 (version 1.0)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 4:00pm Friday 13 May, 2022

Introduction

The goal of this assignment is to demonstrate your skills and ability in fundamental process management and communication concepts, and to further develop your C programming skills with a moderately complex program.

You are to create two programs – the first is called `sigcat` which is like the Unix utility `cat`, however it has enhanced signal handling functionality. The second, and major program, is called `hq`, and it is used to interactively spawn new processes, run programs, send and receive output and signals to those process, and manage their lifecycle. The assignment will also test your ability to code to a programming style guide, to use a revision control system appropriately, and document important design decisions (CSSE7231 only).

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 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 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 actions will be initiated against you, and those you cheated with. That's right, 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, and do not allow others to access your computer - you must keep your code secure.

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 these sites and they cooperate with us in misconduct investigations.

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

Code Origin	**Usage/Referencing**
Code provided to you in writing **this semester** by CSSE2310/7231 teaching staff (e.g. code hosted on Blackboard, posted on the discussion forum, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3)	May be used freely without reference. (This assumes that no reference was required for the original use.)
Code examples found in man pages on `moss`.	May be used provided the source of the code is referenced in a comment adjacent to that code.
Code you have personally written in a previous enrolment in this course or in another ITEE course and where that code has not been shared or published.	
Code (in any programming language) that you have taken inspiration from but have not copied.	
Other code – includes: code provided by teaching staff only in a previous offering of this course (e.g. previous A1 solution); code from websites; code from textbooks; any code written by someone else; and any code you have written that is available to other students.	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). Copied code without adjacent referencing will be considered misconduct and action will be taken.

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. If you are not able to adequately explain your code or the design of your solution 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.

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: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

Specification – `sigcat`

`sigcat` reads one line at a time from `stdin`, and immediate writes and flushes that line to an output stream. The output stream by default is `stdout`, however the output stream can be changed at runtime between `stdout` and `stderr` by sending `sigcat` the `SIGUSR1` and `SIGUSR2` signals.

Full details of the required behaviour are provided below.

Command Line Arguments

Your program (`sigcat`) accepts no commandline arguments.

```
./sigcat
```

Any arguments that are provided can be silently ignored.

`sigcat` basic behaviour

Upon starting, `sigcat` shall enter an infinite loop reading newline-terminated lines from standard input, and emitting them to an output stream. It is assumed that the data read from standard input is non-binary, i.e. does not contain null (`\0`) characters.

- At program start, the output stream is to be standard output `stdout`.
- `sigcat` shall remain in this loop until it receives end of file on `stdin`.
- Upon reaching EOF on `stdin`, `sigcat` shall exit with exit code 0.
- No further error checking is required in `sigcat`.

`sigcat` signal handling

`sigcat` shall register a handler or handlers for all signals of numeric value 1 through to 31 inclusive – except 9 (`KILL`) and 19 (`STOP`) which are not able to be caught.

Upon receiving a signal, `sigcat` is to emit, to the current output stream, the following text:

```
sigcat received <signal name>
```

where <signal name> is replaced with the signal name as reported by `strsignal()` (declared in <string.h>). Note that this line is terminated by a newline and `sigcat` must flush its output buffer after every emitted line of text.

Examples include:

```
sigcat received Quit
sigcat received Hangup
```

The signals `SIGUSR1` and `SIGUSR2` have special meaning to `sigcat`. After printing the output as specified above, upon receipt of either of these signals `sigcat` shall further

- set its output stream to standard output (`stdout`) if `SIGUSR1` is received
- set its output stream to standard error (`stderr`) if `SIGUSR2` is received

In this way, `sigcat` can be instructed to direct its output to either `stdout` or `stderr` by sending it the appropriate signals.

Specification – hq

hq reads commands from its standard input one line at a time. All of hq's output goes to stdout– and all commands are terminated by a single newline. The commands are documented below, and allow the user to run programs, send them signals, manage their input and output streams, report on their status and so on.

Full details of the required behaviour are provided below.

Command Line Arguments

hq accepts no commandline arguments.

```
./hq
```

Any arguments that are provided may be silently ignored.

hq basic behaviour

hq prints and flushes a prompt "> " (greater than symbol followed by a single space) to stdout then waits to read a command from stdin.

The following table describes the commands that must be implemented by hq, and their syntax. Additional notes on each command will follow.

Command	Usage	Comments
spawn	spawn <program> [<arg1>] [<arg2>] ...	Run <program> in a new process, optionally the with arguments provided. Arguments or program names containing spaces may be quoted in double quotes. The new process's stdin and stdout must be connected to hq by pipes, so that they can be accessed with the send and rvc commands. The new process's stderr should be unchanged.
report	report [<jobid>]	Report on the status of <jobid> or all jobs if no argument provided
signal	signal <jobid> <signum>	Send the signal (<signum> – an integer) to the given job (<jobID>)
sleep	sleep <seconds>	Cause hq to sleep for the number of seconds specified. <seconds> may be fractional, e.g. sleep 1.5
send	send <jobid> <text>	Send <text> to the job. Strings containing spaces must be quoted with double quotes
rcv	rcv <jobid>	Attempt to read one line of text from the job specified and display it to hq's stdout
eof	eof <jobid>	Close the pipe connected to the specified job's stdin, thus causing that job to receive EOF on its next read attempt.
cleanup	cleanup	Terminate and reap all child processes spawned by hq by sending them signal 9 (SIGKILL).

The following apply to all command handling and inputs:

- Upon reaching EOF on stdin, hq shall terminate and clean up any jobs (see the cleanup command below), and exit with status 0.
- hq shall block or otherwise ignore SIGINT (Control-C).
- Any invalid commands provided to hq (i.e. a word at the start of a line is not one of the above) shall result in the following message to stdout:

  ```
  Error: Invalid command
  ```

  Note that empty input lines (user just presses return) shall just result in the prompt being printed again.
- All commands have a minimum number of arguments (possibly zero such as for report and cleanup). If this minimum number of arguments is not provided, the following error message shall be emitted to standard output:

  ```
  Error: Insufficient arguments
  ```

Extraneous arguments, if provided, shall be silently ignored.

- All numerical arguments, if present, must be complete and valid numbers. e.g. “`15`” is a valid integer, but “`15a`” is not. Similarly, “`2.7`” is a valid floating point value, but “`2.7foobar`” is not. Your program must correctly identify and report invalid numerical arguments (see details below for each command).
- Any text arguments, including strings and program names, may contain spaces if the argument is surrounded by double quotation marks, e.g. `"text with spaces"`. A helper function is provided to assist you with quote-delimited parsing, see the “Provided Library” section on page 9 for usage details.
- One or more spaces may be present before the command and there may be more than one space between arguments. The provided helper function will deal with this for you.
- Where a command takes a jobID argument then a *valid* jobID is the ID of any job created using `spawn`– even if the execution failed or the job has exited.

`spawn`

The `spawn` command shall cause `hq` to `fork()` a new process, setup pipes such that the child’s `stdin` and `stdout` are directed from/to `hq`, and `exec()` the requested program. The `$PATH` variable is to be searched for executable programs.

Each spawned process is to be allocated an integer job identifier (jobID), starting from zero. Jobs are created and job IDs should increment even if the `exec()` call fails.

`hq` should emit the following to stdout:

```
New Job ID [N] created
```

where `N` is replaced by the integer value, e.g.

```
New Job ID [34] created
```

If at least one argument is not provided (the program name), then `spawn` shall emit the following message:

```
Error: Insufficient arguments
```

If the `exec()` call fails then your child process is to exit with exit status 99.

`report`

The `report` command shall output information on all jobs spawned so far, with a header row and in the following format:

```
> report
[Job] cmd:status
[0] cat:running
[1] ls:exited(0)
[2] sleep:signalled(9)
...
```

The `cmd` field shall be the name of the job (the value of the first argument given to the `spawn` command). The `status` field shall be one of `running` – if the job has not yet exited; `exited` – if the job has terminated normally – with the exit status shown in parentheses; or `signalled` – if the job terminated due to a signal – with the signal number shown in parentheses. Jobs are to be reported in numerical order.

`report` may optionally take a single integer argument, which is a job ID. If provided, and valid, then only the status of that job shall be reported. (The header line is also output.)

```
> report 1
[Job] cmd:status
[1] ls:exited(0)
```

If an invalid job ID is provided (i.e. non-numerical value or job was never spawned), then an error message is to be emitted to standard output as demonstrated below:

```
> report 45
Error: Invalid job
> report abc
Error: Invalid job
```

`signal`

The `signal` command shall cause a signal to be sent to a job. Exactly two integer arguments must be specified – the target job ID, and the signal number. If fewer arguments are provided, the following error is emitted:

```
Error: Insufficient arguments
```

If the job ID is invalid, emit:

```
Error: Invalid job
```

If the signal number is invalid (non-numeric, less than 1 or greater than 31):

```
Error: Invalid signal
```

If all arguments are valid, the signal shall be sent to the targetted job. (There is no need to check whether the job is still running.)

`sleep`

The `sleep` command shall cause the `hq` program to sleep for the number of seconds specified in its single mandatory argument. Non-integer numbers, such as 1.5, are considered valid[1].

If no duration is provided, emit the error message:

```
Error: Insufficient arguments
```

If a negative or non-numerical duration is provided, emit the error message to `stdout`:

```
Error: Invalid sleep time
```

`send`

The `send` command shall send a single line of text, whose contents are the second argument to the command, to the identified job. Send requires exactly two arguments, the job ID and the string to be sent. Because arguments are separated by spaces, to send a line containing spaces, the text must be contained in double quotes. A helper function is provided to assist you with quote-delimited parsing, see see the "Provided Library" section on page 9 for usage details.

If less than two arguments are provided, `send` shall emit:

```
Error: Insufficient arguments
```

If an invalid job ID is provided, emit the error message:

```
Error: Invalid job
```

Example of usage:

```
> send 0 "hello job, quotes matter!"
> send 2 textwithoutspaces
> send 1 ""
> send 4 text with spaces but these extra words are all ignored
```

It is possible that the receiving process has exited – it is your job to manage any `SIGPIPE` signals so that your program does not terminate in this situation. You are not required to detect or report if a job specified in a `send` command is in this state – simply ensure that `hq` continues to run.

[1] The `strtod()` function may prove useful here

rcv

`rcv` shall attempt to read one line of text from the identified job, and print it to standard output. One mandatory argument is required – the job ID. If not specified, then emit

```
Error: Insufficient arguments
```

If an invalid job ID is specified, emit

```
Error: Invalid job
```

`rcv` must not block if there is no output available from the job. To facilitate this, a helper function `is_ready()` is provided – see the "Provided Library" section on page 9.

If there is no output available to read from the job, `rcv` shall emit

```
<no input>
```

If end-of-file is (or has previously been) received from the pipe connected to the job, then `rcv` shall emit

```
<EOF>
```

Otherwise, `rcv` shall emit to standard output, the line of text read from the job, e.g.

```
> rcv 0
Hello from the job!
```

eof

The `eof` command shall close the stdin of the given job. One mandatory argument is required – the job ID. If not specified, then emit

```
Error: Insufficient arguments
```

If an invalid job ID is specified, `eof` shall emit

```
Error: Invalid job
```

If the job ID is valid, `hq` does not output anything.

cleanup

The `cleanup` command shall send the `SIGKILL` (numerical value 9) signal to all jobs and `wait()` on them to reap zombies.

Example `hq` Sessions

```
$ ./hq
> spawn cat /etc/resolv.conf
New Job ID [0] created
> report
[Job] cmd:status
[0] cat:exited(0)
> rcv 0
# Generated by NetworkManager
> rcv 0
search labs.eait.uq.edu.au eait.uq.edu.au
> rcv 0
nameserver 130.102.71.160
> rcv 0
nameserver 130.102.71.161
> rcv 0
<EOF>
```

```
> send 0 foobar
> send 0 "anybody there?"
> rcv 0
<EOF>
>
```

Even though the `cat` process has exited almost immediately, its output has been buffered in the connecting pipe and can still be read. Note also that the `send` command is sending down a pipe that has nothing listening on the other end. The kernel will be sending `SIGPIPE` to `hq` but these are being handled / ignored.

In the next example, we spawn a process running `cat` which will be expecting input on `stdin`, and sending it back to `stdout`. `hq` can be used to manage this with the `send` and `rcv` commands:

```
$ ./hq
> spawn cat
New Job ID [0] created
> report
[Job] cmd:status
[0] cat:running
> rcv 0
<no input>
> send 0 "hello world"
> rcv 0
hello world
> report
[Job] cmd:status
[0] cat:running
> send 0 "line 1" extra args
> send 0 "line 2"
> rcv 0
line 1
> rcv 0
line 2
>
```

Next we illustrate the use and effects of the `signal` command:

```
$ ./hq
> spawn cat
New Job ID [0] created
> report
[Job] cmd:status
[0] cat:running
> signal 0 9
> report
[Job] cmd:status
[0] cat:signalled(9)
```

All of these examples had only a single job, however `hq` must be able to keep an arbitrary number of jobs in flight at once:

```
$ ./hq
> spawn cat /etc/services
New Job ID [0] created
> spawn cat /etc/passwd
New Job ID [1] created
> spawn cat
New Job ID [2] created
> report
[Job] cmd:status
[0] cat:running
[1] cat:exited(0)
```

```
[2] cat:running
> rcv 0
# /etc/services:
> send 2 "hello world"
> rcv 1
root:x:0:0:root:/root:/bin/bash
> rcv 2
hello world
> rcv 2
<no input>
> eof 2
> rcv 2
<EOF>
> report
[Job] cmd:status
[0] cat:running
[1] cat:exited(0)
[2] cat:exited(0)
```

In this example, jobs 0 and 1 were simply catting files. Job 1 ran and terminated almost immediately, but job 0 did not because the file being `cat`'ed was larger than the pipe buffer so `cat` was blocked on output. Job 2 was a cat blocking on standard input, and we interacted with it using `send` and `rcv`. We then send job 2 an end-of-file with the `eof` command, and confirmed using `report` that job 2 had now also exited, having come to end of file on input.

Here is an example of `hq` and `sigcat` interacting:

```
1  > spawn ./sigcat
2  New Job ID [0] created
3  > send 0 "Hello sigcat"
4  > rcv 0
5  Hello sigcat
6  > signal 0 1
7  > rcv 0
8  sigcat received Hangup
9  > signal 0 12
10 > signal 0 1
11 > sigcat received Hangup
12 rcv 0
13 sigcat received User defined signal 2
14 > report
15 [Job] cmd:status
16 [0] ./sigcat:running
```

There are some very important subtleties demonstrated in this example:

- We spawn the `sigcat` process (line 1), send it some text (line 3) and then read it back (lines 4 & 5).
- We then send it signal 1 (`SIGHUP`) on line 6, and read back the output triggered by that signal (lines 7 & 8). Remember that by default, `sigcat` emits its output to `stdout`, which is captured by `hq` and accessible only via the `rcv` command.
- on line 9, we send signal 12 (`SIGUSR2`), which causes `sigcat` to emit the signal message to `stdout`, but then switch its output stream to `stderr` .
- When we then resend signal 1 (line 10), the output message from sigcat (underlined here) appears immediately on the console – because it's emitted over `stderr` and this stream is not captured by `hq`.
- There is no prompt before the input on line 12 because the prompt is shown on line 11 (it was output before the message to `stderr`).

Provided Library: libcsse2310a3

A library has been provided to you with the following functions which your program may use. See the man pages on moss for more details on these library functions.

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

The function attempts to read a line of text from the specified stream, allocating memory for it, and returning the buffer.

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

This function will detect if there is any data available to read on the specified file descriptor, returning 0 (no input) or 1 (input available) accordingly. You will need to use this to prevent your `rcv` command blocking.

```
char** split_space_not_quote(char *input, int *numTokens);
```

This function takes an input string and tokenises it according to spaces, but will treat text within double quotes as a single token.

To use the library, you will need to add `#include <csse2310a3.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing this function. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a3`.

Style

Your program must follow version 2.2.0 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site.

Hints

1. You **may** wish to consider the use of the standard library functions `strtod()`, `strtol()`, `strsignal()` and `usleep()` or `nanosleep()`.
2. While not mandatory, the provided library functions will make your life a lot easier – use them!
3. The standard Unix `tee` command behaves like `cat`, but also writes whatever it receives on `stdin` to a file. This, combined with `watch -n 1 cat <filename>` in another terminal window, may be very helpful when trying to figure out if you are setting up and using your pipes correctly.
4. Review the lectures/contacts from weeks 5 and 6. These cover the basic concepts needed for this assignment and the code samples may be useful.

Suggested Approach

It is suggested that you write your program using the following steps. Test your program at each stage and commit to your SVN repository frequently. Note that the specification text above is the definitive description of the expected program behaviour. The list below does not cover all required functionality.

1. Write `sigcat` first. It will be useful for testing `hq` later.
2. Write small test programs to figure out the correct usage of the system calls required for each `hq` command – i.e. how to connect both `stdin` and `stdout` of a child process to pipes and manage access to them from the parent.
3. Prototype the overall command loop of `hq` first, and work out how to parse input lines into tokens. You can then implement each command (`spawn`, `report` etc) incrementally, using knowledge you gained from step 2 above.

Forbidden Functions

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

- `goto`

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

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` files and compiled programs).

Your programs (named `sigcat` and `hq`) must build on `moss.labs.eait.uq.edu.au` with:
`make`

Your programs must be compiled with `gcc` with at least the following options:
`-pedantic -Wall -std=gnu99`

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.

The default target of your Makefile must cause both programs to be built[2].

If any errors result from the `make` command (i.e. no executable is 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

`https://source.eait.uq.edu.au/svn/csse2310-sem1-sXXXXXXX/trunk/a3`

where sXXXXXXX is your moss/UQ login ID. 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/a3` 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 a3`

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

`sXXXXXXX_csse2310_a3_`*`timestamp`*`.zip`

where `sXXXXXXX` is replaced by your moss/UQ login ID and *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 in order to check out your submission from your repository.

You must not create the zip file using some other mechanism and you must not modify the zip file prior to 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 creation of your submission zip file.

We will mark 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 CSSE2310/7231 course profile for details.

[2] If you only submit an attempt at one program then it is acceptable for just that single program to be built when running `make`.

[3] You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck in order 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.

Marks

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

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. 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 create a child process (job) then we can not test your communication with that job. If your program takes longer than 15 seconds to run any test, then it will be terminated and you will earn no marks for the functionality associated with that test. 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.

1. `sigcat` correctly copies its input to `stdout` (3 marks)
2. `sigcat` correctly outputs messages upon receiving signals (3 marks)
3. `sigcat` correctly changes output streams upon receipt of `SIGUSR1` and `SIGUSR2` (4 marks)
4. `hq` correctly rejects invalid commands (3 marks)
5. `hq` correctly implements `spawn` command (6 marks)
6. `hq` correctly implements `report` command (7 marks)
7. `hq` correctly implements `sleep` command (4 marks)
8. `hq` correctly implements `send` command (6 marks)
9. `hq` correctly implements `eof` command (3 marks)
10. `hq` correctly implements `rcv` command (6 marks)
11. `hq` correctly implements `signal` command (4 marks)
12. `hq` correctly implements `cleanup` command (4 marks)
13. `hq` correctly handles `SIGPIPE` and `SIGINT` as appropriate (2 marks)
14. `hq` frees all allocated memory prior to exit (5 marks)

Some functionality may be assessed in multiple categories, e.g. the ability to tokenise strings containing spaces and within quotes.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.2 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.

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 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 are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission. This 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). (Automated style marking can only be undertaken on code that compiles. The provided `style.sh` script checks this for you.)

If your code does compile 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 `style.sh` when it is run over each of your `.c` and `.h` files individually[6].

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. Note that in some cases `style.sh` may erroneously report style violations when correct style has been followed. If you believe that you have been penalised incorrectly then please bring this to the attention of the course coordinator and your mark can be updated if this is the case. Note also that when `style.sh` is run for marking purposes it may detect style errors not picked up when you run `style.sh` on moss. This will not be considered a marking error – it is your responsibility to ensure that all of your code follows the style guide, even if styling errors are not detected in some runs of `style.sh`.

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 "other". The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide. Note that functions longer than 50 lines will be penalised in the automated style marking. Functions that are also longer than 100 lines will be further penalised here.

Comments (2.5 marks)

Mark	Description
0	The majority (50%+) of comments present are inappropriate OR there are many required comments missing
0.5	The majority of comments present are appropriate AND the majority of required comments are present
1.0	The vast majority (80%+) of comments present are appropriate AND there are at most a few missing comments
1.5	All or almost all comments present are appropriate AND there are at most a few missing comments
2.0	Almost all comments present are appropriate AND there are no missing comments
2.5	All comments present are appropriate AND there are no missing comments

Naming (1 mark)

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

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

[6] 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.

Other (1.5 marks)

Mark	Description
0	One or more functions is longer than 100 lines of code OR there is more than one global/static variable present inappropriately OR there is a global struct variable present inappropriately OR there are more than a few instances of poor modularity (e.g. repeated code)
0.5	All functions are 100 lines or shorter AND there is at most one inappropriate non-struct global/static variable AND there are at most a few instances of poor modularity
1.0	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no or very limited use of magic numbers AND there is at most one instance or poor modularity
1.5	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no use of magic numbers AND there are no instances of poor modularity

SVN commit history assessment (5 marks)

Markers will review your SVN commit history for your assignment up to your submission 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)
- 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) 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:

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

Design Documentation (10 marks) – for CSSE7231 students only

CSSE7231 students must submit a PDF document containing a written overview of the architecture and design of your program. This must be submitted via the Turnitin submission link on Blackboard.

Please refer to the grading criteria available on BlackBoard under "Assessment" for a detailed breakdown of how these submissions will be marked. Note that your submission time for the whole assignment will be considered to be the later of your submission times for your zip file and your PDF design document. Any late penalty will be based on this submission time and apply to your whole assignment mark.
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. It must meet the following formatting requirements:

- Maximum two A4 pages in 12 point font
- Diagrams are permitted up to 25% of the page area. The diagram(s) 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 will be asked to explain why.

Total Mark

Let

- F be the functionality mark for your assignment (out of 60).
- S be the automated style mark for your assignment (out of 5).
- H be the human style mark for your assignment (out of 5).
- C be the SVN commit history mark (out of 5).
- D be the documentation mark for your assignment (out of 10 for CSSE7231 students) – or 0 for CSSE2310 students.

Your total mark for the assignment will be:

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

out of 75 (for CSSE2310 students) or 85 (for CSSE7231 students).

In other words, you can't get more marks for style or SVN commit history 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.

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 or emailed to `csse2310@uq.edu.au`.