The University of Queensland
School of Electrical Engineering and Computer Science

CSSE7231 – Semester 1, 2025
Assignment 4 (Version 1.1)

Marks: 85
Weighting: 15%
Due: 3:00pm Friday 30 May, 2025

This specification was created for the use of Andrew WILSON (s4828041) only.
Do not share this document. Sharing this document may result in a misconduct penalty.
Note that line numbers vary from those in the CSSE2310 specification.
Specification changes since version 1.0 are shown in red 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. You are to create a server program (`uqfacedetect`) that supports multiple simultaneously connected clients and responds to requests from those clients to detect faces within a received image and replace those faces with another image (if received). You should also create a client program (`uqfaceclient`) that can send requests to the server and save responses.

Communication between the clients and `uqfacedetect` is over TCP using a defined communication protocol. Advanced functionality such as connection limiting, signal handling and statistics reporting are also required for full marks. CSSE7231 students are expected to implement some additional functionality using HTTP.

The assignment will also test your ability to code to a particular programming style guide and to use a revision control system appropriately. You will also need to use the Open Computer Vision (`OpenCV`) library.

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. 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. (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 CSSE7231 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 Origin	Usage/Referencing
Code you wrote this semester for this course Code you have personally written this semester for CSSE7231 (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.	**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 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.)	
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`.

Specification - `uqfaceclient`

The `uqfaceclient` program provides a command line interface that allows you to interact with the server (`uqfacedetect`) as a client – connecting, sending an image to detect faces, sending an (optional) image to replace faces with, receiving the modified image back from the server and saving it to a file.

Your `uqfaceclient` will not require multiple threads or processes – it will be able to construct a request based on the command line arguments, connect to the server, send the request, await a response, and then save the response to a file.

Command Line Arguments

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

```
./uqfaceclient port [--outputimage filename] [--replaceimage filename]
        [--detectfile filename]
```

The square brackets (`[]`) indicate optional groups of arguments. The *italics* indicate placeholders for user-supplied arguments. The ***port*** argument must always be the first argument. Option arguments can be in any order after the ***port*** argument.

Some examples of how the program might be run include the following[1]:

```
./uqfaceclient 1234 --detectfile detect.jpg --outputimage out.jpg
```

```
./uqfaceclient 3456 --detectfile detect.jpg --replaceimage replace.png \
--outputimage out.jpg
```

```
./uqfaceclient 8978 < detect.jpg > out.jpg
```

```
./uqfaceclient http --outputimage out.jpg --replaceimage replace.png < detect.jpg
```

```
./uqfaceclient 2310 --detectfile detect.jpg > out.jpg
```

The meaning of the arguments is as follows:

- ***port*** – this mandatory argument specifies which localhost port the server is listening on – either numerical or the name of a service.
- `--detectfile` – if specified, this option argument is followed by the name of a file containing an image. This is the image that will be sent to the server to be processed. If an input file is not specified on the command line then the image is to be read from `uqfaceclient`'s standard input.
- `--replaceimage` – if specified, this option argument is followed by the name of a file containing an image. This is the image that will be sent to the server to replace faces with.
- `--outputimage` – if specified, this option argument is followed by the name of a file where the output image will be saved. If an output file is not specified on the command line then the image will be sent to `uqfaceclient`'s standard output.

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: ./uqfaceclient port [--outputimage filename] [--replaceimage filename]
        [--detectfile filename]
```

to standard error (with a following newline), and exit with an exit status of 6.

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

- No arguments are present (i.e. there is no ***port*** argument)

[1] This is not an exhaustive list and does not show all possible combinations of arguments. The examples also assume that a `uqfacedetect` server is running on the listed ports.

- The `--detectfile`, `--replaceimage` or `--outputimage` option argument is given but it is not followed by a non-empty value argument string.
- Any of the option arguments is listed more than once.
- An unexpected argument is present.
- Any argument is the empty string.

Checking whether the *`port`* is a valid port or service name and/or whether the filename(s) are valid is not part of the usage checking (other than checking that their values are not empty). The validity of these values is checked after command line validity as described in the sections below – and in the same order as these sections are listed.

File Checking

Your client program must check that the input file(s) (if specified) can be opened for reading – first checking the `--detectfile` file and then, if applicable, the `--replaceimage` file. If not, then your program must print the message:

```
uqfaceclient: unable to open the input file "filename" for reading
```

to standard error (with a following newline) where *filename* is replaced by the name of the file from the command line. The double quotes must be present. Your program must then exit with an exit status of 11.

If the input file(s) can be opened, then your client program must check that the output file (if specified) can be opened for writing (creating the file if not present, truncating it if it is). If not, then your program must print the message:

```
uqfaceclient: cannot open the output file "filename" for writing
```

to standard error (with a following newline) where *filename* is replaced by the name of the file from the command line. The double quotes must be present. Your program must then exit with an exit status of 9. Note that this check will result in the creation/truncation of a file even if a check below fails and your program exits.

If your program is unable to open a file then it will exit immediately – the other files (if any) will not be checked.

Port Checking

If `uqfaceclient` 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 1:

```
uqfaceclient: unable to connect to the server on 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.

Runtime Behaviour

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

- read the input image(s) from the file(s) specified on the command line or from `stdin` if no face detection input file was specified on the command line. Note that this is binary data, not textual data (i.e. it will almost certainly contain null (0) bytes so should not be treated as a string).
- construct a request for the required image operation and send it to the server. See the Communication Protocol section later for details on how the request should be constructed.
- await a response from the server and then handle it appropriately as described below:
  - If the response follows the specified format and the operation is set to "output image", then the output image data must be saved to the output file specified on the command line (or sent to `stdout` if no output file is specified on the command line). Your program should then exit with exit status 0.
  - If the response follows the specified format and the operation is set to "error message", then your program must print the message:

    ```
    uqfaceclient: got the following error message: "message"
    ```

to `stderr` (with a following newline) where *`message`* is replaced by the message received from the server. The double quotes must be present. Your program must then exit with an exit status of 5.

– If `uqfaceclient` receives a response that does not follow the specified communication protocol or does not receive a response, e.g. because the network connection is closed (`uqfaceclient` detects EOF on the socket or receives SIGPIPE when writing to the socket), then your program must output the following message:

```
uqfaceclient: unexpected communication error
```

to `stderr` (terminated by a newline) and exit with status 16. Note that this also covers the case where the sending of the request failed since no response will be received.

Other requirements

Your `uqfaceclient` program must free all memory before exiting. If an output file is created, then the permissions for this file must include at least `rw` permissions for the owner.

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

Specification - `uqfacedetect`

`uqfacedetect` is a networked, multithreaded image processing server allowing clients to connect, send an image for manipulation (and an optional image to replace faces with), and then return a manipulated image to the client. All communication between clients and the server is over TCP using a message format that is described in the Communication Protocol section.

Command Line Arguments

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

`./uqfacedetect` *`clientlimit maxsize`* `[`*`portnumber`*`]`

The square brackets (`[]`) indicates 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:

```
./uqfacedetect 5 0
```

```
./uqfacedetect 0 10000000 2310
```

```
./uqfacedetect 100 0 1234
```

The meaning of the arguments is as follows:

- *`clientlimit`* – 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 we will not test).
- *`maxsize`* – this argument must be specified and is expected to be a 32-bit unsigned integer less than or equal to $2^{32} - 1$ specifying the maximum image size (in bytes) that will be accepted from clients. If this argument is zero, then the image size limit is $2^{32} - 1$ bytes.
- *`portnumber`* – if specified, this argument is a string which specifies which localhost port the 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 `uqfacedetect` is to use an ephemeral port.

Important: Even if you do not implement the connection limiting functionality, your program must correctly handle command lines which 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: ./uqfacedetect clientlimit maxsize [portnumber]
```

to standard error (with a following newline), and exit with an exit status of 6.

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

- The *clientlimit* 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 *portnumber* 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 *portnumber* 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.

Temporary Image File Checking

The OpenCV library will get images to process by reading them from a file and will generate output by writing to a file. Your uqfacedetect server will therefore need to save images from clients to a file before requesting OpenCV to load that file, and similarly, OpenCV processing output will be saved to a file, before being read by your server and sent back to clients. In the interests of minimising disk usage, a single file (/tmp/imagefile.jpg[2]) will be used for all transfers (which uqfacedetect will need to suitably protect with a semaphore or mutex to ensure one client doesn't interfere with the operation of another).

After validating the command line, your server program must check that the file /tmp/imagefile.jpg can be opened for writing, truncate the file, then close it. If any of these steps fail, then your program must print the message:

```
uqfacedetect: unable to open the image file for writing
```

to standard error (with a following newline) and then exit with an exit status of 9. Note that this check will result in the creation/truncation of the file even if a check below fails and your program exits. Note that .jpg is at the end of the filename to tell the OpenCV library function cvSaveImage() to save output images in the JPEG format. cvLoadImage(), which opens and reads an image file, will determine the image type from the contents of the file, not the name. This means that images sent from clients can be in any format (e.g. JPEG, PNG, TIFF, etc.), so, for example, a PNG image received from a client can be saved as /tmp/imagefile.jpg and will still be treated as a PNG image when loaded by OpenCV.

Cascade Classifier File Checking

Your server program must then check that both of the following two files can be loaded using the cvLoad() function:

```
/local/courses/csse2310/resources/a4/haarcascade_frontalface_alt2.xml
/local/courses/csse2310/resources/a4/haarcascade_eye_tree_eyeglasses.xml
```

If not, then your program must print the message:

```
uqfacedetect: unable to load a cascade classifier
```

to standard error (with a following newline) and then exit with an exit status of 14.

The returned CvHaarClassifierCascade structures are used to detect faces and eyes, respectively, when a pointer is passed as an argument to the cvHaarDetectObjects() function (refer to the examples in the The OpenCV library section below). Since multiple clients can connect to the server and send face detection and replacement requests at the same time, these structures must be protected with a semaphore or mutex.

Port Checking

If uqfacedetect 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 19:

```
uqfacedetect: 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, and the socket can't be listened on. Note that we will not test the situation where uqfacedetect is unable to listen on an ephemeral port.

[2]Note that all users on moss have their own /tmp directory, so you will not be interfering with servers being run by other users.

Runtime Behaviour

Once the port is opened for listening, `uqfacedetect` 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, `uqfacedetect` shall spawn a new thread to handle that client (see below for client thread handling).

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

The `uqfacedetect` program should not terminate under normal circumstances, nor should it block or otherwise attempt to handle `SIGINT`.

Note that your `uqfacedetect` 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.

Client handling threads

A client handler thread is spawned for each incoming connection. This client thread must then wait for a request from the client over the socket. The exact format of the requests is described in the Communication Protocol section below. When a request is received, it should be handled and a response returned to the requestor. The client thread must then wait for another request.

Due to the simultaneous nature of the multiple client connections, your `uqfacedetect` will need to ensure mutual exclusion around any shared data structure(s) such as the `CvHaarClassifierCascade` structures to ensure that these do not get corrupted.

Once a client disconnects or there is a communication error on the socket (e.g. a `read()` or equivalent from the client returns `EOF`, or a badly formed request is received, or a `write()` or equivalent fails) then the client handler thread is to close the connection, clean up and terminate. Other client threads and the `uqfacedetect` program itself must continue uninterrupted. `uqfacedetect` must not exit in response to a `SIGPIPE`.

SIGHUP handling (Advanced)

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

- Total number of clients connected (at this instant)
- The total number of clients that have connected and disconnected since program start
- The number of valid face detection requests (from all clients since server start up) that have been responded to successfully (a response with an output image was sent)
- The number of valid face replacement requests (from all clients since server start up) that have been responded to successfully (a response with an output image was sent)
- The number of times a badly formed request is received – which is the same as the number of times the contents of the response file was sent (see the Badly formed requests section below)

The required statistics format is illustrated below. Each of the five 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: `uqfacedetect` SIGHUP `stderr` output sample

```
Clients connected: 6
Num clients completed: 34
Face detect requests: 5
Face replacement requests: 10
Invalid requests: 3
```

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

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 *clientlimit* feature is implemented and a non-zero command line argument is provided, then `uqfacedetect` must not permit more than that number of simultaneous client connections to the server. If a client beyond that limit attempts to connect, `uqfacedetect` 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.

Statistics HTTP reporting (CSSE7231 students only)

Upon startup, `uqfacedetect` shall check the value of the environment variable `A4_STATS_PORT`. If set, then `uqfacedetect` shall also listen for HTTP connections on that port number (or service name) – using a separate thread.

If that environment variable is set and `uqfacedetect` is unable to listen on that port or service name then it shall output the following message to `stderr` and terminate with exit status 12:

```
uqfacedetect: unable to listen on given statistics port "N"
```

The ability to listen on this port is checked immediately after the ability to listen on the "main" port (i.e. the one given on the command line – or an ephemeral port). If the `A4_STATS_PORT` environment variable is not set then `uqfacedetect` shall not listen on any additional ports and shall not handle these HTTP connections.

The communication protocol uses HTTP. The connecting program (e.g. `netcat`, or a web browser) shall send HTTP requests and `uqfacedetect` shall send HTTP responses as described below. The connection between client and server is kept alive between requests. Multiple stats clients may be connected simultaneously.

Additional HTTP header lines beyond those specified may be present in requests or responses and must be ignored by the respective server/client. Note that interaction between a client on the stats port and `uqfacedetect` is *synchronous* – any one client can only have a single request underway at any one time.

The only supported request method is a `GET` request in the following format:

- Request: `GET /stats HTTP/1.1`
  - Description: the client is requesting statistics from `uqfacedetect`
  - Request
    - Request Headers: none expected, any headers present will be ignored by `uqfacedetect`.
    - Request Body: none, any request body will be ignored
  - Response
    - Response Status: 200 (OK).
    - Response Headers: the `Content-Length` header with correct value is required (number of bytes in the response body), other headers are optional.
    - Response Body: the ASCII text containing the same `uqfacedetect` statistics information described in the `SIGHUP` handling section above.

If `uqfacedetect` receives an invalid HTTP stats request then it should close the connection to that requestor (and terminate the thread associated with that connection). No response is to be sent. Similarly, if a stats HTTP client disconnects, `uqfacedetect` should handle this gracefully and terminate the thread associated with the connection. Stats clients are not included in the client count statistics and are not subject to client connection limiting.

Other Requirements

Other than error messages, the listening port number, and `SIGHUP`-initiated statistics output, `uqfacedetect` 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()`, or `get_HTTP_request()` (see libcsse2310a4 later) 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 `OpenCV` 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-uqfacedetect` on `moss`.

Communication Protocol

Clients and `uqfacedetect` must communicate using the image processing communication protocol (see Table 1). Valid requests and responses are described below. Note that the `uqfaceclient` described earlier is limited to requesting a single image operation. The server must support clients that send multiple requests over one connection.

Table 1: **The image processing communication protocol.** Each message will consist of a prefix, operation type, image size and image data. Multi-byte numbers are transmitted in little-endian format (i.e. the same format used for in-memory representation on `moss`).

Number of Bytes	Data Type	Description
4	32-bit unsigned integer	**Prefix** – `0x23107231` – this fixed number at the start of the message indicates that this is an image processing protocol message.
1	8-bit unsigned integer	**Operation type** – this integer indicates the type of message. The number must be either 0 (face detection request), 1 (face replacement request), 2 (output image) or 3 (error message).
4	32-bit unsigned integer	**Image 1 size** – number of bytes (M) of image 1, which is the image to detect faces or the output image, depending on the specified operation. If the operation type is 3, then this is the number of bytes of the error message.
M	Bytes	**Image 1 data** – the data for image 1. The bytes may have any value. If the operation type is 3, then this is the data for the error message.
4	32-bit unsigned integer	**Image 2 size (only present in face replacement requests)** – number of bytes (N) of image 2, which is the image to replace faces with.
N	Bytes	**Image 2 data (only present in face replacement requests)** – the data for image 2. The bytes may have any value.

Badly formed requests

If the first four bytes received by the server are not the expected prefix (`0x23107231`), then the server must send the contents of the `moss` file `/local/courses/csse2310/resources/a4/responsefile` then terminate the connection with the client. The communication protocol above is NOT used to send this response – the file contents are just returned as is. This file may change over the course of the assignment so you must not hardwire the contents into your program, but your program may just read it once if desired (e.g. on startup or when it receives the first of these requests). Alternatively, you may read and send back the contents of the file for every such request. (It can be assumed that reading of this file will succeed. Your program does not have to deal with the failure to read this file and may behave in any way you like if such reading fails.) Note that the file may contain binary data (including null characters) so should not be treated as a string. The contents of this file will allow you to test your server using a web browser on your local machine. See the Testing section later for details.

Face detection and replacement requests and responses

If the expected prefix (`0x23107231`) is received by the server, then the operation type that follows the prefix must either indicate a face detection request or face replacement request. For a face detection request, the client handling thread is to save the image it receives from the client to `/tmp/imagefile.jpg`, and then use the `cvLoadImage()` function to load the image into memory. If the thread can load the image sent from the client, it can be assumed that the image processing will be successful[3]. The thread will then use the `OpenCV` library to detect faces and eyes and draw ellipses around the detected faces and eyes (refer to The `OpenCV` library section below) and save the image to `/tmp/imagefile.jpg`. Finally, the thread will send the image to the client using the communication protocol with an operation type of "output image" and then wait for a new request from the same client.

The procedure for a face replacement request is similar to a face detection request, except that the client handling thread expects two images to be sent from the client. The first image is the image to detect faces and the second image is to be used to replace faces with. The first image is to be saved to `/tmp/imagefile.jpg`

[3] "Successful" does not mean faces will be found – just that `OpenCV` will be able to perform operations on the image.

then loaded into memory. The second image should then be saved to the same file and then loaded into memory. The output image should also be saved to the same file and then sent to the client.

The client handling thread is to send an error message (not null terminated) to the client using the communication protocol, close the connection to the client, clean up and terminate if it:

1. Receives an invalid communication protocol message. The error message should be: "invalid message".
2. Does not receive a valid operation type, i.e. is not a face detection or replacement request. The error message should be: "invalid operation type".
3. Receives an image that is 0 bytes in size. The error message should be: "image is 0 bytes".
4. Receives an image that exceeds the maximum image size (as specified on the command line). The error message should be: "image too large".
5. Cannot load an image (using `cvLoadImage()`) that a client has sent. The error message should be: "invalid image".
6. Does not detect any faces in the received face detection image. The error message should be: "no faces detected in image".

Remember to protect access to `/tmp/imagefile.jpg` and shared data structures to prevent threads simultaneously reading and writing to the same file or data structure!

Provided Libraries

libcsse2310a4

Several HTTP-related functions have been provided to aid in parsing/construction of HTTP requests/responses. The functions in this library are:

```
int get_HTTP_request(FILE *f, char **method, char **address,
        HttpHeader ***headers, unsigned char **body, unsigned long* bodySize);
```

```
unsigned char* construct_HTTP_response(int status, char* statusExplanation,
        HttpHeader** headers, const unsigned char* body,
        unsigned long bodySize, unsigned long* len);
```

```
void free_header(HttpHeader* header);
```

```
void free_array_of_headers(HttpHeader** headers);
```

These functions and the `HttpHeader` type 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`.

The `OpenCV` library

Open Computer Vision (`OpenCV`) is a library for image processing. You will be using the C API and only the server will need to use this library. There a few steps to using `OpenCV`, as illustrated in the two examples below. You should copy this code and modify it to suit your needs. **The `OpenCV` parameters that you use must be the same as those in the examples.** The `OpenCV` version that is installed on `moss` is 3.4.6. The documentation can be accessed at `https://docs.opencv.org/3.4.6/`. Within this website, the search bar can be used to look up any functions that you would like to learn more about. You can find some test images in the following directory on `moss`: `/local/courses/csse2310/resources/a4/testimages/` . You are welcome to also use your own test images.

To use the `OpenCV` library functions, you will need to add the `#include` directives in the examples below to your code. You will also need to link with the libraries containing these functions. To do this, use the following compiler arguments when linking:
`-L/usr/lib64 -lopencv_core -lopencv_imgcodecs -lopencv_objdetect -lopencv_imgproc`

Face and eye recognition is achieved using Haar cascade classifiers. If you would like to learn more about this method, refer to the tutorial at `https://docs.opencv.org/4.11.0/db/d28/tutorial_cascade_classifier.html`.

Example 2: `OpenCV` usage example to detect faces and eyes within a provided image and draw ellipses around faces and circles around eyes.

```
#include <stdio.h>
#include <opencv2/imgcodecs/imgcodecs_c.h>
#include <opencv2/imgproc/imgproc_c.h>
#include <opencv2/objdetect/objdetect_c.h>

// OpenCV parameters
const float haarScaleFactor = 1.1;
const int haarMinNeighbours = 4;
const int haarFlags = 0;
const int haarMinSize = 0;
const int haarMaxSize = 1000;
const int ellipseStartAngle = 0;
const int ellipseEndAngle = 360;
const int lineThickness = 4;
const int lineType = 8;
const int shift = 0;

// File locations
const char* const faceCascadeFilename = "/local/courses/csse2310/resources/a4/"
                                        "haarcascade_frontalface_alt2.xml";
const char* const eyesCascadeFilename = "/local/courses/csse2310/resources/a4/"
                                        "haarcascade_eye_tree_eyeglasses.xml";

int main(int argc, char** argv)
{
    argc--;
    argv++;
    if (argc != 2) {
        fprintf(stderr, "Usage: ./detectfaces inputfile outputfile\n");
        return 1;
    }
    char* inputFilename = argv[0];
    char* outputFilename = argv[1];
    // Load the cascades
    CvHaarClassifierCascade* faceCascade = (CvHaarClassifierCascade*)cvLoad(
            faceCascadeFilename, NULL, NULL, NULL);
    CvHaarClassifierCascade* eyesCascade = (CvHaarClassifierCascade*)cvLoad(
            eyesCascadeFilename, NULL, NULL, NULL);
    if (!faceCascade || !eyesCascade) {
        return 2;
    };
    // Load the image to detect faces as a 3 channel BGR
    IplImage* frame = cvLoadImage(inputFilename, CV_LOAD_IMAGE_COLOR);
    if (!frame) {
        // Free memory
        cvReleaseHaarClassifierCascade(&faceCascade);
        cvReleaseHaarClassifierCascade(&eyesCascade);
        return 3;
    }
    // Grayscale and equalise the input image
    IplImage* frameGray = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
    cvCvtColor(frame, frameGray, CV_BGR2GRAY);
    cvEqualizeHist(frameGray, frameGray);
    // Create memory for calculations, allocate and clear it
    CvMemStorage* storage = 0;
    storage = cvCreateMemStorage(0);
    cvClearMemStorage(storage);
    // Detect faces
    // Set flags to 0 because don't need to do optimisation to speed up
    // processing
    CvSeq* faces = cvHaarDetectObjects(frameGray, faceCascade, storage,
            haarScaleFactor, haarMinNeighbours, haarFlags,
            cvSize(haarMinSize, haarMinSize), cvSize(haarMaxSize, haarMaxSize));
    // Iterate through each detected face and draw an ellipse around it
    for (int i = 0; i < faces->total; i++) {
        CvRect* face = (CvRect*)cvGetSeqElem(faces, i);
        CvPoint center
                = {face->x + face->width / 2, face->y + face->height / 2};
        const CvScalar magenta = cvScalar(255, 0, 255, 0);
        const CvScalar blue = cvScalar(255, 0, 0, 0);
        cvEllipse(frame, center, cvSize(face->width / 2, face->height / 2), 0,
```

```
                ellipseStartAngle, ellipseEndAngle, magenta, lineThickness,
                lineType, shift);
        IplImage* faceROI
                = cvCreateImage(cvGetSize(frameGray), IPL_DEPTH_8U, 1);
        cvCopy(frameGray, faceROI, NULL);
        cvSetImageROI(faceROI, *face);
        // Create memory for calculations, allocate and clear it
        CvMemStorage* eyeStorage = 0;
        eyeStorage = cvCreateMemStorage(0);
        cvClearMemStorage(eyeStorage);
        // Detect eyes within each face
        CvSeq* eyes = cvHaarDetectObjects(faceROI, eyesCascade, eyeStorage,
                haarScaleFactor, haarMinNeighbours, haarFlags,
                cvSize(haarMinSize, haarMinSize),
                cvSize(haarMaxSize, haarMaxSize));
        if (eyes->total != == 2) { // should only have 2 eyes, skip if not
            continue;
        }
            // Draw a circle around each eye
            for (int j = 0; j < eyes->total; j++) {
                CvRect* eye = (CvRect*)cvGetSeqElem(eyes, j);
                CvPoint eyeCenter = {face->x + eye->x + eye->width / 2,
                        face->y + eye->y + eye->height / 2};
                int radius = cvRound((eye->width / 2 + eye->height / 2) / 2);
                cvCircle(frame, eyeCenter, radius, blue, lineThickness, lineType,
                        shift);
            }
        }
        // Free memory
        cvReleaseImage(&faceROI);
        cvReleaseMemStorage(&eyeStorage);
    }
    // Save the processed image
    cvSaveImage(outputFilename, frame, 0);
    // Free memory
    cvReleaseImage(&frame);
    cvReleaseImage(&frameGray);
    cvReleaseHaarClassifierCascade(&faceCascade);
    cvReleaseHaarClassifierCascade(&eyesCascade);
    cvReleaseMemStorage(&storage);
    return 0;
}
```

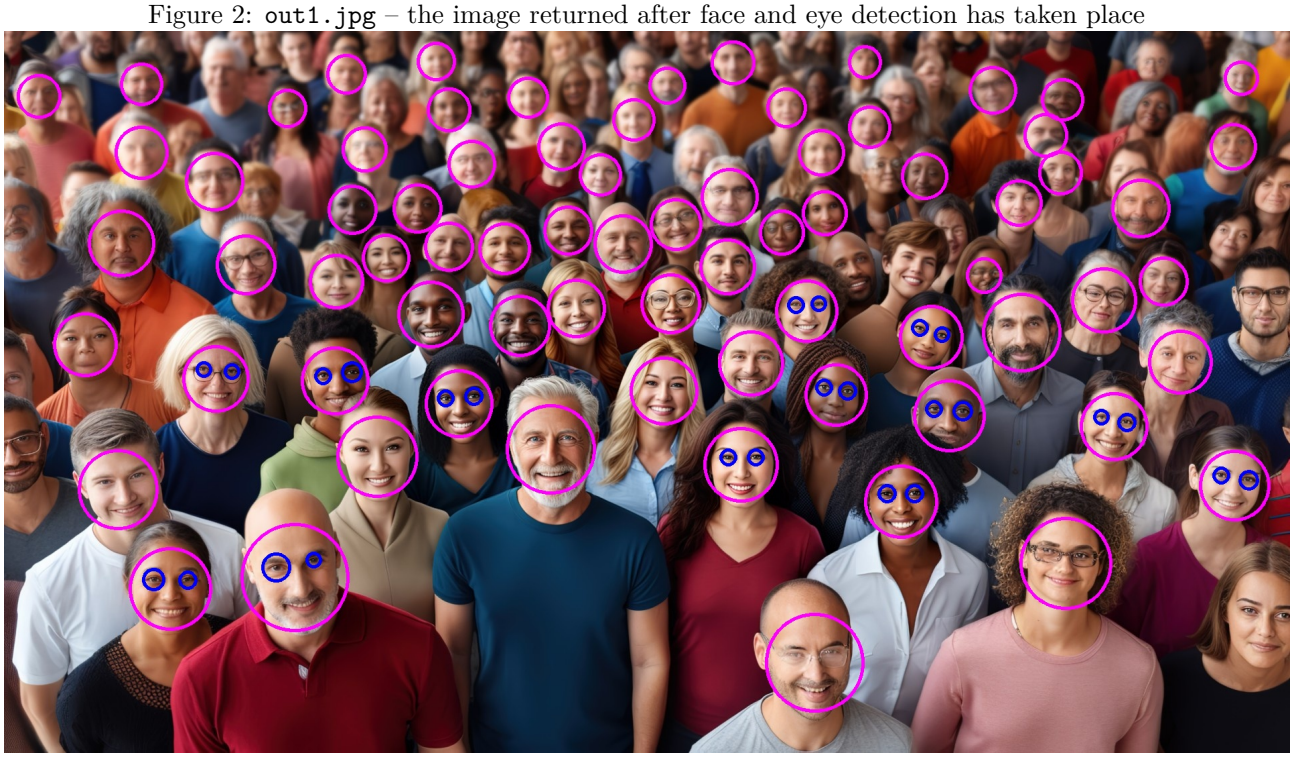
When compiled to a program called `detectfaces`, it can be run using the command:

```
./detectfaces AdobeStock_691406423.jpeg out1.jpg
```

`AdobeStock_691406423.jpeg` is shown in Figure 1. The created `out1.jpg` file is shown in Figure 2.

Figure 1: `AdobeStock691406423orig.jpeg` – the original image used for face detection

Figure 2: `out1.jpg` – the image returned after face and eye detection has taken place



Example 3: `OpenCV` usage example to detect faces within a provided image and replace the faces with a provided replacement image.

```
#include <stdio.h>
#include <opencv2/imgcodecs/imgcodecs_c.h>
#include <opencv2/imgproc/imgproc_c.h>
#include <opencv2/objdetect/objdetect_c.h>

// OpenCV parameters
const float haarScaleFactor = 1.1;
const int haarMinNeighbours = 4;
const int haarFlags = 0;
const int haarMinSize = 0;
const int haarMaxSize = 1000;
const int bgraChannels = 4;
const int alphaIndex = 3;

// File location
const char* const faceCascadeFilename = "/local/courses/csse2310/resources/a4/"
                                        "haarcascade_frontalface_alt2.xml";

int main(int argc, char** argv)
{
    argc--;
    argv++;
    if (argc != 3) {
        fprintf(stderr, "Usage: ./replacefaces inputfile outputfile\n");
        return 1;
    }
    char* inputFilenameDetect = argv[0];
    char* inputFilenameReplace = argv[1];
    char* outputFilename = argv[2];
    // Load the cascade
    CvHaarClassifierCascade* faceCascade = (CvHaarClassifierCascade*)cvLoad(
            faceCascadeFilename, NULL, NULL, NULL);
    if (!faceCascade) {
        return 2;
    };
    // Load the image to detect faces as a 3 channel BGR
    IplImage* frame = cvLoadImage(inputFilenameDetect, CV_LOAD_IMAGE_COLOR);
    if (!frame) {
        // Free memory
        cvReleaseHaarClassifierCascade(&faceCascade);
        return 3;
    }
    // Load the image to replace faces as a 3 channel BGR or 4 channel BGRA.
    // For PNG format, IMREAD_UNCHANGED comes with alpha so that it's
    // BGRA. For JPEG, there's only 3 channels (BGR).
    IplImage* replace
            = cvLoadImage(inputFilenameReplace, CV_LOAD_IMAGE_UNCHANGED);
    if (!replace) {
        // Free memory
        cvReleaseImage(&frame);
        cvReleaseHaarClassifierCascade(&faceCascade);
        return 4;
    }
    // Grayscale and equalise the input image
    IplImage* frameGray = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
    cvCvtColor(frame, frameGray, CV_BGR2GRAY);
    cvEqualizeHist(frameGray, frameGray);
    // Create memory for calculations, allocate and clear it
    CvMemStorage* storage = 0;
    storage = cvCreateMemStorage(0);
    cvClearMemStorage(storage);
    // Detect faces
    // Set flags to 0 because don't need to do optimisation to speed up
    // processing
    CvSeq* faces = cvHaarDetectObjects(frameGray, faceCascade, storage,
            haarScaleFactor, haarMinNeighbours, haarFlags,
            cvSize(haarMinSize, haarMinSize), cvSize(haarMaxSize, haarMaxSize));
    // Iterate through each detected face and replace it with an image
    for (int i = 0; i < faces->total; i++) {
        CvRect* face = (CvRect*)cvGetSeqElem(faces, i);
        IplImage* resized = cvCreateImage(cvSize(face->width, face->height),
```

```
                IPL_DEPTH_8U, replace->nChannels);
        // Resize the replacement image to be the size of the face
        cvResize(replace, resized, CV_INTER_AREA);
        char* frameData = frame->imageData;
        char* faceData = resized->imageData;
        // Iterate through each pixel in the image to replace
        // and draw over the original image
        for (int y = 0; y < face->height; y++) {
            for (int x = 0; x < face->width; x++) {
                int faceIndex
                        = (resized->widthStep * y) + (x * resized->nChannels);
                // if BGRA, then look at the alpha channel
                if ((resized->nChannels == bgraChannels)
                        && (faceData[faceIndex + alphaIndex] == 0)) {
                    // If alpha is 0 then skip this pixel
                    continue;
                }
                // Frame is BGR which is 3 channels
                int frameIndex = (frame->widthStep * (face->y + y))
                        + ((face->x + x) * frame->nChannels);
                frameData[frameIndex + 0] = faceData[faceIndex + 0];
                frameData[frameIndex + 1] = faceData[faceIndex + 1];
                frameData[frameIndex + 2] = faceData[faceIndex + 2];
            }
        }
        // Free memory
        cvReleaseImage(&resized);
    }
    // Save the processed image
    cvSaveImage(outputFilename, frame, 0);
    // Free memory
    cvReleaseImage(&frame);
    cvReleaseImage(&replace);
    cvReleaseImage(&frameGray);
    cvReleaseHaarClassifierCascade(&faceCascade);
    cvReleaseMemStorage(&storage);
    return 0;
}
```

When compiled to a program called `replacefaces`, it can be run using the command:

```
./replacefaces AdobeStock_691406423.jpeg smiley1.png out2.jpg
```

`smiley1.png` is shown in Figure 3. The created `out2.jpg` file is shown in Figure 4.

Figure 3: `smiley1.png` – the image that will be used to replace detected faces

Testing

As well as testing your client with the demo server (`demo-uqfacedetect`) and your server with the demo client (`demo-uqfaceclient`) on `moss`, remember that you can use `netcat` to simulate and capture requests/responses.

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

Figure 4: `out2.jpg` – the output after face replacement

deadline. Gradescope will run the test suite immediately after you submit. When this is complete[4] 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.

In addition to this, you can use a web browser on your local machine to interact with your `uqfacedetect` server on `moss`. You will need to use `ssh` to forward a port on your local machine to the port your server is listening on on `moss`. You will also need to install a program called `websockify` that will listen on a different local port and will allow your web browser to communicate using the communication protocol defined for `uqfacedetect`. If your server correctly responds to an invalid prefix, then the returned response file contents containing HTTP headers and HTML with embedded Javascript will allow you to specify the port that `websockify` is listening on, upload an image from your local computer (and optional replacement image), specify an image operation, submit it for processing, and display the result.

To implement port forwarding, you should:

- start a `uqfacedetect` server on `moss` – assumed to be listening on port N
- from your local machine (e.g. laptop) start another `ssh` session that also forwards a local port. For example:

  `ssh -L 8080:localhost:`*`N`* *`sNNNNNNN`*`@moss.labs.eait.uq.edu.au`

  will forward any connections to port 8080 on your local machine to port N on `moss` (via `moss`' `localhost` interface). You can choose a port number other than 8080 to listen on.
- Use the URL `http://localhost:8080/` in your local web browser to access your server. This will send an HTTP GET request to `uqfacedetect` which should result in the response file being returned (because the HTTP request does not start with the expected prefix) and the HTML within will be displayed by your browser.
- Start `websockify` listening on another local port (e.g. 8081) that will forward WebSockets connections on that port to your forwarded port, e.g. run

  ```
  websockify 8081 localhost:8080
  ```

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

- Fill in the HTML form – including specifying the WebSockets port as 8081 – and press "Submit".

You can (re)start the server after starting the port forwarding, provided that you know your server can listen on the specified port. If you have to choose a different port on `moss` for your server, you'll need to start a port forwarding session to that port instead.

Style

Your program must follow version 3 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 `uqfacedetect`.
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.
3. You can test `uqfacedetect` and `uqfaceclient` independently using `netcat`. You can also use the provided demo programs `demo-uqfaceclient` and `demo-uqfacedetect` to (1) understand the expected functionality, and (2) test your client with the demo server and vice versa.
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. Since both `uqfacedetect` and `uqfaceclient` use the same communication protocol, functions related to the protocol should be implemented in separate `.c` and `.h` files and not duplicated in the client and server.

Possible Approach

1. Try implementing `uqfaceclient` first. (The programs are independent so this is not a requirement, but when you test it with `demo-uqfacedetect` it may give you a better understanding of how `uqfacedetect` works.)
2. For `uqfacedetect`, start with the multithreaded network server example from the lectures, gradually adding functionality for supported operations.

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()`
- Functions described in the man page as non standard, 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 `uqfaceclient` and `uqfacedetect`) 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 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-sem1/csse2310-s4828041/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)[5]. The zip file will be named

`s4828041_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

[5] 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.

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[6] will incur a late penalty – see the CSSE7231 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 CSSE7231 Student Interviews section below.

Functionality (70 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 can save an image correctly (since the image file contents come from 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.

Marks will be assigned in the following categories. There are 20 marks for `uqfaceclient` and 50 marks for `uqfacedetect`.

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

8. `uqfacedetect` correctly handles invalid command lines (2 marks)
9. `uqfacedetect` correctly listens for connections and reports the port (including inability to listen for connections) (2 marks)
10. `uqfacedetect` correctly responds to requests with an unexpected prefix (i.e. by sending the response file) (3 marks)
11. `uqfacedetect` correctly handles errors in the communication protocol and images sent by clients (6 marks)
12. `uqfacedetect` correctly handles face detection requests from a single client (4 marks)

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

13. `uqfacedetect` correctly handles face replacement requests from a single client (4 marks)
14. `uqfacedetect` correctly handles face detection and replacement requests from multiple clients (5 marks)
15. `uqfacedetect` correctly handles disconnecting clients and communication failure (including not exiting due to SIGPIPE) (3 marks)
16. `uqfacedetect` correctly implements client connection limiting (3 marks)
17. `uqfacedetect` correctly implements SIGHUP statistics reporting (including protecting data structures with mutexes or semaphores) (6 marks)
18. `uqfacedetect` does not leak memory and does not busy wait (2 marks)

19. (CSSE7231 only) `uqfacedetect` correctly listens on the port specified by `A4_STATS_PORT` (and handles inability to listen or environment variable not set) (3 marks)
20. (CSSE7231 only) `uqfacedetect` correctly responds to HTTP statistics requests (including invalid requests) from a single client connected at a time issuing one request per connection (4 marks)
21. (CSSE7231 only) `uqfacedetect` supports multiple simultaneous HTTP statistics clients and multiple sequential requests over each connection (3 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). CSSE7231 functionality in categories 18 and 19 will require correct statistics gathering.

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 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[7].

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[8] 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

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

[8] 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.

- 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[9].

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.

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.

[9] 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.

- 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 70 for CSSE7231 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 (see the CSSE7231 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 85 (for CSSE7231 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.

CSSE7231 Student Interviews

This section has been changed from assignments one and three – only a subset of students will be interviewed.

The teaching staff will conduct interviews with a subset of CSSE7231 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

- `uqfaceclient` – clarified sequencing of checking whether files can be opened (lines 92 to 106)
- Fixed memory leak in example code for face detection (example 2).
- Added line numbers to code listings (examples 2 and 3)
- Added `signal()` to the list of forbidden functions (line 469)
- Clarified that the communication protocol is not used when returning the response file (line 323)