The University of Queensland
School of Electrical Engineering and Computer Science

# CSSE2310 – Semester 2, 2024
# Assignment 1 (Version 1.2)

Marks: 75
Weighting: 15%
**Due: 3:00pm Friday 23 August, 2024**

This specification was created for the use of Zhengxuan ZHANG (s4905773) only.
Do not share this document. Sharing this document may result in a misconduct penalty.
Specification changes since version 1.0 are shown in red and are summarised at the end of the document.
Changes from v1.1 to v1.2 are shown in blue.

## Introduction

The goal of this assignment is to give you practice at C programming. You will be building on this ability in the remainder of the course (and subsequent programming assignments will be more difficult than this one). You are to create a program (called `uqentropy`) that will determine the strength of possible passwords entered on `stdin` based on either (or both) their presence in a set of known passwords (from one or more files) and/or the length and characteristics of the possible passwords. The assignment will also test your ability to code to a particular programming style guide, and to use a revision control system appropriately.

## Student Conduct

**This 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**<br>Code provided to you in writing **this semester** by CSSE2310 teaching staff (e.g., code hosted on Blackboard, found in `/local/courses/csse2310/resources` on `moss`, posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class). | **Permitted**<br>May be used freely without reference. (You must be able to point to the source if queried about it – so you may find it easier to reference the code.) |
| **Code you wrote this semester for this course**<br>Code you have personally written this semester for CSSE2310 (e.g. code written for A1 reused in A3) – provided you have not shared or published it. | **Permitted**<br>May be used freely without reference. (This assumes that no reference was required for the original use.) |

| **Code Origin** | **Usage/Referencing** |
|---|---|
| **Unpublished code you wrote earlier**<br>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**<br>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`**<br>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**<br>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**<br>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**<br>Code, in any programming language:<br>• copied from any website or forum (including StackOverflow and CSDN);<br>• copied from any public or private repositories;<br>• copied from textbooks, publications, videos, apps;<br>• copied from code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution);<br>• written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member);<br>• written by an AI tool that you did not personally and solely interact with;<br>• written by you and available to other students; or<br>• from any other source besides those mentioned in earlier table rows above. | **Prohibited**<br>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.<br>This prohibition includes code written in other programming languages that has been converted to C. |
| **Code that you have learned from**<br>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**<br>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

The `uqentropy` program will read candidate passwords from standard input (`stdin`) and evaluate their strength by determining their *entropy* – which is a measure of randomness or uncertainty. The more "random" a password, the harder it is to guess, so the stronger it is.

Two different approaches will be used by `uqentropy` to calculate entropy.

## Entropy

The entropy ($E$) of a password can be defined as $\log_2(N)$ where $N$ is the number of possible passwords – i.e. the number of possible combinations of the set of symbols used to create the password. $N$ can be defined as $S$ to the power of $L$, (i.e. $S^L$) where $S$ is the size of the set of symbols and $L$ is the length of the password. Entropy ($E$) is measured in bits[1]. A password with 10 bits of entropy would be equivalent in strength to a set of 10 bits chosen perfectly randomly – it would require $2^{10}$ guesses to guarantee guessing the password. Each extra bit of entropy makes a password twice as hard to guess.

For example, a password known to be constructed from 8 lower case letters (a to z only) has an entropy of $E = \log_2(26^8) \approx 37.6$ bits. In this case, $S = 26$ (the number of symbols in the set – lower case letters in this case), and $L = 8$ (the length of the password).

As another example, if a password is known to be constructed from a set of symbols including lower case letters, upper case letters and digits (62 possible symbols) and has length 10, then the entropy of the password is $E = \log_2(62^{10}) \approx 59.5$ bits.

Another way of thinking about entropy is based on how many guesses it would take to guess the password using a brute force attack of all possible passwords. The expected number of guesses ($n$) is half the number of possible passwords ($N$). Sometimes a brute force attack will get it right quite quickly, other times you'll need to check nearly all possible passwords. On average, you'll need to check 50% of the possible passwords. Using the formula above ($E = log_2(N)$), if you consider $N = 2n$ then you can define entropy as $E = log_2(2n)$ (where $n$ is the expected number of guesses to guess the password).

This approach gives a better estimate of entropy when a password isn't truly random. For example, if an attacker uses a list of common passwords in a brute force attack then they will be able to guess the password quite quickly if a common password is in use. Suppose that a user uses the password "`password`" and that this is entry number 7 on a list of common passwords used by an attacker[2]. The expected number of guesses to brute force this password would be 7. The entropy of this password calculated using this approach would therefore be $E = log_2(2 \times 7) \approx 3.8$ bits – significantly lower than the value of 37.6 determined above for a combination of 8 lower case letters.

## Password strength

We will use the following ratings of passwords based on their entropy:

- $< 35$ – very weak
- 35 to $< 60$ – weak
- 60 to $< 120$ – strong
- 120 or higher – very strong

These are somewhat arbitrary but they provide reasonable guidance to users.

[1] Strictly speaking, information entropy is measured in shannon (Sh) – see `https://en.wikipedia.org/wiki/Shannon_(unit)` but bits is more commonly seen.

[2] `https://nordpass.com/most-common-passwords-list/`, accessed 25 July 2024

## Command Line Arguments

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

`./uqentropy [--leet] [--double] [--digit-append numdigits] [--case] [listfilename ...]`

The square brackets (`[]`) indicate optional arguments or groups of arguments. The *italics* indicate placeholders for user-supplied value arguments. An ellipsis (...) indicates the previous argument can be repeated.

`uqentropy` expects zero or more option arguments to follow the command name (with an associated value argument in the case of `--digit-append`). Option arguments can be in any order. Zero or more filenames (password lists) will follow this – though if an option argument is given then at least one *`listfilename`* argument must be provided. It is acceptable to run `uqentropy` with no command line arguments. It is also possible to provide one or more *`listfilename`* arguments without any option arguments.

Some examples of how the program might be run include the following[3]:

```
./uqentropy

./uqentropy /usr/dict/share/words

./uqentropy --case ./passwordfile.txt

./uqentropy --double --leet --case ./passwordfile.txt

./uqentropy --leet list1 list2 list3 /usr/share/dict/words

./uqentropy --digit-append 5 words.txt
```

The meaning of the arguments is as follows. More details on the expected behaviour of the program are provided later in this document.

- `--case` – this argument specifies that we will compare each candidate password (from `stdin`) not just against the passwords in the given file(s) but also against any variants of those passwords obtainable by changing the case of any subset of the letters in those passwords (i.e. each letter can either be upper or lower case). If this argument is present then at least one filename must be given on the command line.
- `--leet` – this argument specifies that when we compare the candidate password against those in the password list(s), then we will also check various letter substitutions as commonly used in 'leetspeak'. If this argument is present then at least one filename must be given on the command line.
- `--digit-append` – this argument specifies that when we compare the candidate password against those in the password list(s), then we will also check those passwords with any combination of 1 to *`numdigits`* digits appended. If the `--digit-append` argument is present (with its associated value argument) then at least one filename must be given on the command line.
- `--double` – this argument specifies that we should also compare the candidate password against any combination of two passwords from the password list(s). If the `--double` argument is present then at least one filename must be given on the command line.
- *`listfilename`* ... – filenames specified on the end of the command line are assumed to be the names of files containing password lists against which the candidate password(s) are to be compared. There may be zero of more of these file names specified. It can be assumed that the first (or only) file name argument does not start with the characters "`--`". (If the user wants to specify a file whose name does start with "`--`" then they should prefix the name with "`./`". Any arguments after the first file name are also assumed to be file names, even if they begin with "`--`".)

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: ./uqentropy [--leet] [--double] [--digit-append 1..8] [--case] [listfilename ...]
```

to standard error (with a following newline), and exit with an exit status of 2.

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

- The `--digit-append` option argument is given but it is not followed by a single digit value argument which is an integer between 1 and 8 inclusive.
- One (or more) of `--case`, `--leet`, `--digit-append` (with a value), and/or `--double` is specified but no file name is given on the command line.
- Any of the option arguments is listed more than once.

[3] This is not an exhaustive list and does not show all possible combinations of arguments. The examples assume the existence of the listed files.

- An unexpected argument is present.
- Any argument is the empty string.

Checking whether files exist or can be opened is not part of the usage checking (other than checking that filename values are not empty). This is checked after command line validity as described below.

## File Checking

Your program must check that any password list files mentioned on the command line can be opened for reading and contain at least one valid password. If a file can not be opened for reading, then your program must print the message:

```
uqentropy: unable to open 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.

If a file can be opened for reading, then the contents of the file must be read into memory. Each file is assumed to contain a list of possible passwords where each entry is separated by any number of whitespace characters[4]. Whitespace characters at the start or end of the file are to be ignored. Valid passwords must contain at least one character and will never contain whitespace characters (since these separate passwords) nor any non-printable characters[5].

If an openable file is found to contain any non-printable characters (besides whitespace characters that separate passwords), then your program must print the message:

```
uqentropy: invalid character found in file "filename"
```

to `stderr` (with a following newline) where *`filename`* is replaced by the name of the file from the command line. The double quotes must be present. This message must be printed at most once per file, even if the file contains multiple non-printable characters. (This means `uqentropy` can stop reading a file as soon as such an error is detected.)

If an openable file ~~with no non-printable characters~~ that passes the check above is found to contain no valid passwords, then your program must print the message:

```
uqentropy: "filename" does not contain any passwords
```

to `stderr` (with a following newline) where *`filename`* is replaced by the name of the file from the command line. The double quotes must be present.

Every file name listed must be checked and read into memory in turn – in the order they appear on the command line. Multiple error messages may be printed – but at most one per file.

If any error occurs, then, after checking all files, your program must exit with an exit status of 20.

## Program Behaviour

### Program Startup

Assuming that the checks above are successful, then your program must print the following to standard output (`stdout`), with a newline at the end of each line:

```
Welcome to UQEntropy!
Written by s4905773.
Enter candidate passwords to check their strength.
```

Your program must then repeatedly read lines from `stdin` (each terminated by a newline character or pending EOF). If the line (excluding any terminating newline) is a valid candidate password then it will be evaluated for its strength. Valid candidate passwords will (a) have at least one character, and (b) only contain printable characters and (c) not contain any whitespace characters. If the line entered is not a valid candidate password then your program must print the following message (terminated by a newline) to `stderr` and attempt to read another candidate password from `stdin`:

```
Password is invalid
```

If a valid candidate password is entered, then your program must then calculate the entropy of the password using the first method below, and possibly also the second method, depending on the command line arguments to the program.

[4] A whitespace character is any character for which the C function `isspace()` returns true, e.g., space, newline, tab, etc.
[5] A non-printable character is any character for which the C function `isprint()` returns false

### Entropy Calculation 1 – Based on Symbols Used

If a valid candidate password has been entered then its entropy must be calculated based on the symbols used in the password and the size of the sets they belong to. Your program must examine the candidate password to determine the type(s) of symbols used. The symbol set size must be determined based on the following table – based on the row that matches the set of symbols used in the candidate password.

| **Symbols Used** | **Set Size** |
|---|---|
| Digits in the range 0 to 9 only | 10 |
| Lower case letters (a to z) only | 26 |
| Upper case letters (A to Z) only | 26 |
| Non alphanumeric printable ASCII characters only, e.g., any of<br>`! " # $ % & ’ ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ‘ { \| } ∼` | 32 |
| Any combination of the above sets | Sum of the set sizes |

The entropy ($E_1$) of the candidate password should then be calculated as described on page 3, i.e. $E_1 = log_2(S^L) = L \times log_2(S)$ where $S$ is the size of the set of symbols used and $L$ is the length of the candidate password. For example, if the candidate password is “`abcd3f6h!j`” then $S = 68$ (26 for lowercase letters plus 10 for digits plus 32 for other ASCII characters) and $L = 10$, so the entropy is $E_1 = log_2(68^{10}) = 10 \times log_2(68) \approx 60.9$. If the candidate password is “`a1B2c3%`”, then the entropy is $E_1 = log_2((26 + 26 + 10 + 32)^7) = log_2(94^7) = 7 \times log_2(94) \approx 45.9$. It is best to use the second variant of the formula when computing this, i.e. $E_1 = L \times log_2(S)$, to avoid overflow when calculating $S^L$.

### Entropy Calculation 2 – Based on Matching Passwords From List(s)

If any password lists are provided on the command line then `uqentropy` will check whether the candidate password can be found in the list(s) and will determine the *match number*, i.e., determine the number of guesses an attacker would need to make before matching the candidate password, if the attacker was using the given password list(s) and any secondary matching approach specified on the command line. If a match is found, then no further checking is performed and the entropy can then be calculated based on the match number. Attempted matching approaches are checked in the order listed here. See the Hints section for some suggestions on how to do this matching.

- Each valid entry in the list(s) is checked in turn, i.e., each valid entry in the first password list file is checked in the order that they appear in the file, followed by each valid entry in the second password list file (if specified), etc. If an exact match is found, then no further checking is performed. See below for how entropy is then calculated when a match is found.

- If no exact match has been found after checking each valid entry and `--case` has been specified on the command line, then every valid entry in the password list(s) that contains one or more letters is checked again (in the same order as before) using all combinations of upper and lower case letters in place of the letters used in the entry. For example, if an entry in a password list file is `abc123`, then your program must determine whether the candidate password matches any of the following: `Abc123`, `aBc123`, `abC123`, `ABc123`, `AbC123`, `aBC123`, or `ABC123`. If a match is found then no further checking is performed. The match number is incremented as if all combinations of letters (other than the original) were checked, i.e. $2^n - 1$ is added to the match number, where $n$ is the number of letters in the password entry. (In this example, 7 would be added to the match number when this entry is checked.)

- If no match has yet been found and `--digit-append` has been specified on the command line (with value $n$), then every entry in the password list(s) that does not end in a digit is checked again (in the same order as before) using all combinations of 1 to $n$ digits, appended to the entry. For example, if an entry in the password list file is `password` and `--digit-append 2` is specified on the command line, then your program must determine whether the candidate password matches `password0`, `password1`, . . . , `password9`, `password00`, `password01`, . . . , `password99`. If a match is found then no further checking is performed. The match number is incremented based on the number of passwords checked. For example in this case, if the candidate password is `password13`, then the match number would be incremented by 24 (for 0,1,. . . ,9,00,01,. . . ,12,13). The match number would be incremented by $10^1 + 10^2 + \ldots + 10^n$ if no match is possible against that password entry.

- If no match has yet been found and `--double` has been specified on the command line, then it must be checked whether the candidate password will match the concatenation of every pair of valid entries in the password list(s). For example, if the password list contains the passwords `password`, `123456`, and `abc123`, then your program must determine whether the candidate password will match the values `passwordpassword`, `password123456`, `passwordabc123`, `123456password`, `123456123456`, `123456abc123`, `abc123password`, `abc123123456`, or `abc123abc123`. The match number is incremented based on the number of passwords that would need to be checked to find a match, or $n^2$ if no matches are found (where n = number of valid entries in the password list(s)). Passwords must be assumed to be checked in the order shown here, i.e., all combinations that begin with the first entry would be checked before all combinations that begin with the second entry, etc.

- If no match has yet been found and `--leet` has been specified on the command line, then every entry in the password list(s) that contains any of the following letters (lower or upper case) is checked again (in the same order as before) using all combinations of 'LEETspeak' substitutions shown in the following table:

| **Letter** | **Substitute Symbol** |
|---|---|
| a/A | @ or 4 |
| b/B | 6 or 8 |
| e/E | 3 |
| g/G | 6 or 9 |
| i/I | 1 or ! |
| l/L | 1 |
| o/O | 0 |
| s/S | 5 or $ |
| t/T | 7 or + |
| x/X | % |
| z/Z | 2 |

  For example, if an entry in a password list file is `qwertyui`, then your program must determine whether the candidate password matches any of the following: `qw3rtyui`, `qwer7yui`, `qwer+yui`, `qwertyu1`, `qwertyu!`, `qw3r7yui`, `qw3r+yui`, `qw3rtyu1`, `qw3rtyu!`, `qwer7yu1`, `qwer7yu!`, `qwer+yu1`, `qwer+yu!`, `qw3r7yu1`, `qw3r7yu!`, `qw3r+yu1`, `qw3r+yu!`. If a match is found then no further checking is performed. The match number is incremented as if all combinations of subsitutions (other than the original) were checked, i.e. $2^a 3^b - 1$ is added to the match number, where $a$ is the number of letters in the password entry that can be substituted by a single character and $b$ is the number of letters in the password entry that can be substituted by two characters. (In this example, 17 would be added to the match number when this entry is checked because $a$ is 1 (the 'e' can be substituted by one character ('3')) and $b$ is 2 (both the 't' and the 'i' can be substituted by two characters) and $2^1 \times 3^2 - 1 = 17$.)

If a match is found then the following message must be printed to `stdout`:

```
Candidate password would be matched on guess number N
```

If a match is not found then the following message must be printed to `stdout`:

```
No match would be found after checking N passwords
```

In both cases, the message is followed by a newline and $N$ is replaced by the match number (the number of passwords that would have to be checked using the given combination of `uqentropy` option arguments and password lists). You can assume that the match number will never overflow a 64 bit unsigned integer, i.e. an `unsigned long` type on `moss` (i.e. we will not test situations where this overflow happens). Neither of these messages is printed if there are no password list filenames given on the `uqentropy` command line.

If a match would be found during the checks above then the entropy ($E_2$) of the password is calculated as described on page 3, i.e., $E_2 = log_2(2n)$ where $n$ is the match number, i.e. the number of passwords that would need to be checked to find a match using the given combination of option arguments and password lists. If no match is found then the entropy $E_2$ can not be calculated.

### Calculating Overall Entropy and Password Strength

If a password match was found, i.e. $E_2$ was able to be calculated, then the "overall entropy" ($E$) of the password will be the minimum of $E_1$ and $E_2$, i.e. $E = min(E_1, E_2)$.

If no password match would be found (or no password lists or option arguments were provided on the command line), then the overall entropy ($E$) will just be $E_1$ as described above.

Your program must output the overall entropy for the candidate password by printing the following message to `stdout` (with a trailing newline):

```
Password entropy calculated to be E
```

where $E$ is replaced by the overall entropy, rounded down to the nearest 0.1. (One digit must always be shown after the decimal point.)

Your program must then print one of the following messages to `stdout` (with a trailing newline):

- `Password strength rating: very weak` – if the overall entropy is $< 35$
- `Password strength rating: weak` – if the overall entropy is in the range 35 to $< 60$
- `Password strength rating: strong` – if the overall entropy is in the range 60 to $< 120$
- `Password strength rating: very strong` – if the overall entropy is $\geq 120$

Your program must then attempt to read another candidate password from `stdin` and repeat the process above.

### Exiting the Program

If end-of-file (EOF[6]) is detected on `stdin` when your program goes to read a line then your program must exit. If at least one candidate password rated "strong" or "very strong" has been entered, then your program must exit with status 0. If no "strong" or "very strong" password has been entered (including the case when no valid passwords are entered), then your program must print the following message to `stdout` (with a terminating newline):

```
No strong password(s) have been identified
```

and exit with exit status 14.

## Other Requirements

Your program must open and read each password list file only once[7] and store its contents in dynamically allocated memory. Your program must free all allocated memory before exiting, including if it exits due to a usage or file error. (Your program does not have to free memory if it exits due to a signal, e.g. the interrupt signal generated by pressing Ctrl-C.)

# Example Sessions

Some example interactions with `uqentropy` are shown below. Lines typed by the user (i.e. the command and text entered on standard input) are shown in **`bold green`** for clarity. Other lines are output to standard output. Note that the $ character is the shell prompt – it is not entered by the user nor output by `uqentropy`.

**Example 1**: Example `uqentropy` session – with no command line arguments. Note that the user presses Ctrl-D (to terminate input – detected by the program as EOF) just before the line "`No strong password(s) have been identified`" is printed.

```
$ ./uqentropy
Welcome to UQEntropy!
Written by s4905773.
Enter candidate passwords to check their strength.
abc
```

[6] EOF is "end of file" – the stream being read (standard input in this case) is detected as being closed. Where standard input is connected to a terminal (i.e. a user interacting with a program) then the user can close the program's standard input by pressing Ctrl-D (i.e. hold the Control key and press D). This causes the terminal driver which receives that keystroke to close the stream to the program's standard input. Note that a candidate password might be terminated by EOF rather than newline. Such an entry must be treated as if the word was followed by a newline. EOF will be detected on the subsequent attempt to read a word.

[7] `rewind()`ing the file or `fseek()`ing to the beginning are not permitted.

```
Password entropy calculated to be 14.1
Password strength rating: very weak
123456
Password entropy calculated to be 19.9
Password strength rating: very weak
pAs5w0rd
Password entropy calculated to be 47.6
Password strength rating: weak
No strong password(s) have been identified
$ echo $?
14
$
```

**Example 2**: Example `uqentropy` session – with `--case` argument and a small password entry file. Note that entries in the password entry file don't have to be on separate lines – they are just separated by whitespace characters.

```
$ cat passwordsfile
password
abc123 qwerty
$ ./uqentropy --case passwordsfile
Welcome to UQEntropy!
Written by s4905773.
Enter candidate passwords to check their strength.
csse2310
No match would be found after checking 328 passwords
Password entropy calculated to be 41.3
Password strength rating: weak
ABc123
Candidate password would be matched on guess number 265
Password entropy calculated to be 9.0
Password strength rating: very weak
qwerty
Candidate password would be matched on guess number 3
Password entropy calculated to be 2.5
Password strength rating: very weak
AlongPasswordWithLettersOfBothCase
No match would be found after checking 328 passwords
Password entropy calculated to be 193.8
Password strength rating: very strong
$ echo $?
0
$
```

In this case, the 328 passwords that would have to be checked by an attacker are the 3 in the file `passwordsfile`, 255 additional passwords from case variations of the password "password", 7 additional passwords from case variations of the password "abc123" and 63 additional passwords from case variations of the password "qwerty".

**Example 3**: Example `uqentropy` session – with `--digit-append` argument and a large password entry file – the system dictionary.

```
$ ./uqentropy --digit-append 1 /usr/share/dict/words
Welcome to UQEntropy!
Written by s4905773.
Enter candidate passwords to check their strength.
zoo9
Candidate password would be matched on guess number 5271058
Password entropy calculated to be 20.6
Password strength rating: very weak
alphaBET123
```

```
No match would be found after checking 5277558 passwords
Password entropy calculated to be 65.4
Password strength rating: strong
a1
Candidate password would be matched on guess number 46
Password entropy calculated to be 6.5
Password strength rating: very weak
$ echo $?
0
$
```

## 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. Code from Ed Lessons C exercises may be useful to you.
2. The string representation of a single digit positive integer has a string length of 1.
3. You **may** wish to consider the use of the standard library functions `isspace()`, `isdigit()`, `islower()`, `isupper()`, `isprint()`, `toupper()` and/or `tolower()`. Note that these functions operate on individual characters, represented as integers (ASCII values).
4. Potentially useful functions from the maths library include `log2()`, `pow()`, and `floor()`.
5. Some other functions which **may** be useful include: `strcmp()`, `strncmp()`, `strcasecmp()`, `strlen()`, `strdup()`, `exit()`, `fopen()`, `fprintf()`, and `fgetc()`. You should consult the man pages for these functions.
6. You can print out a floating point number to one decimal place using the `%.1f` format specifier to `printf`.
7. Note that if an option argument is given on the command line, your program does not have to evaluate every possible password against the candidate password. For example, if `--case` is specified, you don't have to check whether every case combination of letters in each password entry matches the candidate password – you can just do a case insensitive comparison, e.g. with `strcasecmp()`. Similarly, if `--digit-append` is specified, you don't have to check every combination of digits appended to each password entry against the candidate password. Unlike an attacker, you know the candidate password and you can check whether the candidate password could be formed from a password entry with appended digits. Similar shortcuts apply for the other option arguments. Your program must determine how many guesses it would take an attacker to determine the password. This does not mean your program has to compute all those possible passwords and evaluate each one individually in order to count them. (You can if you wish, but some tests may time out if you do so.)
8. You may wish to consider using a function from the `getopt()` family to parse command line arguments. This is **<u>not</u>** a requirement and if you do so your code is unlikely to be any simpler or shorter than if you don't use such a function. You may wish to consider it because it gives you an introduction to how programs can process more complicated combinations of command line arguments. See the `getopt(3)` man page for details. Note that short forms of arguments are <u>not</u> to be supported, e.g. the argument "`-l`" (for specifying a 'leetspeak' check) should result in a usage error. To allow for the use of a `getopt()` family function, we will not test your program's behaviour with the argument "`--`" (which is interpreted by `getopt()` as a special argument that indicates the end of option arguments). We also won't test abbreviated arguments, e.g. `--le`, `--lee`, etc.
9. Note that `E1`, `e1`, `E2`, `e2`, `E`, `N`, etc. are not meaningful variable names and may result in style penalties if you use them.

# Possible 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 a program that prints the welcome message and exits.
2. Add code to repeatedly read lines from `stdin`, and then exit on EOF. Make sure that you ignore or strip off the newline each the line you read.
3. Add (and call) a function that determines whether a candidate password (the line you read) is valid or not. See line 165 of this specification for the requirements of a valid password.
4. Write (and call) a function that determines the entropy $E_1$ of a word as described on page 6. You will need to iterate over each character to determine the type(s) of characters used in your candidate password so that you can determine the set size. Add code to print the strength rating of the password based on this entropy.
5. Add code that parses the command line arguments and prints a usage error message and exits with status 2 if they are incorrect.
6. Add code that tries to open and read each password list file specified on the command line – and that checks for invalid passwords in the file(s).
7. Add code that checks whether a candidate password (from `stdin`) can be found in the password entries read from the file(s) – and if so, calculate the entropy $E_2$ and the overall entropy.
8. Add code to handle the `--case` option argument.
9. Add code to handle the other option arguments.

# 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()`
- `popen()`
- `mkfifo()` or `mkfifoat()`
- `fork()`
- `pipe()`
- `rewind()`
- `fseek()`
- `execl()`, `execvp()` or any other members of the exec family of functions
- 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.

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.

# Testing

You are responsible for ensuring that your program operates according to the specification. You are encouraged to test your program on a variety of scenarios. You should create your own password entry files and test your `uqentropy` program with a variety of command lines and input from `stdin`. A variety of programs will be provided to help you in testing:

- A demonstration program (called `demo-uqentropy`) that implements the correct behaviour is available on `moss`. You can test your program with a set of command line arguments and input values, and also run the same test with `demo-uqentropy` to check that you get the same output. You can also use `demo-uqentropy` to check the expected behaviour of the program if some part of this specification is unclear.
- A test script will be provided on `moss` that will test your program against a subset of the functionality requirements – approximately 50% of the available functionality marks. The script will be made available about 7 to 10 days before the assignment deadline and can be used to give you some confidence that you're on the right track. The "public tests" in this test script will not test all functionality and you should be sure to conduct your own tests based on this specification. The "public tests" will be used in marking, along with a set of "private tests" that you will not see.
- The Gradescope submission site will also be made available about 7 to 10 days prior to the assignment deadline. Gradescope will run the test suite immediately after you submit. When this is complete[8] 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.

# 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 password list files.

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

```
make
```

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 (i.e. the `uqentropy` 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/csse2310-2024-sem2/csse2310-sXXXXXXX/trunk/a1
```

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/a1` 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 a1
```

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

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)[9]. The zip file will be named

`sXXXXXXX_csse2310_a1_`*`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 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 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.

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[10] will incur a late penalty – see the CSSE2310/7231 course profile for details.

# Marks

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

## Functionality (60 marks)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has been generated correctly and has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Not all features are of equal difficulty.

Partial marks will be awarded for partially meeting the functionality requirements. A number of tests will be run for each marking category listed below, testing a variety of scenarios. Your mark in each 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 program can never open a password list file then we can not determine if your program can check for password matches correctly. 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 when `valgrind` is used to test for memory leaks. 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 output messages is used for functionality marking. Strict adherence to 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).

Marks will be assigned in the following categories.

1. Program correctly handles invalid command lines (usage errors) (8 marks)
2. Program correctly handles file errors: inability to read from file(s), invalid characters in file(s), and/or no passwords in file(s) (6 marks)
3. Program correctly prints the welcome message (with a variety of valid command lines) (3 marks)
4. Program correctly detects invalid candidate passwords (with a variety of valid command lines) (3 marks)
5. Program operates correctly when given no command line arguments (i.e. calculates entropy and strength rating correctly using method one) (5 marks)

[9] 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.

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

6. Program operates correctly when given one password entry file on command line (and no other arguments) – including a variety of such files (4 marks)
7. Program operates correctly when given multiple password entry files on the command line (and no other arguments) (4 marks)
8. Program operates correctly when given `--case` argument (and no other option arguments, but one or more password entry file arguments) (4 marks)
9. Program operates correctly when given `--digit-append` argument with a variety of valid digit counts and no other option arguments, but one or more password entry file arguments) (5 marks)
10. Program operates correctly when given `--double` argument (and no other option arguments, but one or more password entry file arguments) (4 marks)
11. Program operates correctly when given `--leet` argument (and no other option arguments, but one or more password entry file arguments) (4 marks)
12. Program operates correctly when given multiple option arguments (with a variety of password entry file arguments) (4 marks)
13. Program operates correctly, reads password entry files only once, and frees all memory upon exit (for a variety of scenarios tested above) (6 marks)

Some functionality may be assessed in multiple categories. For example, if your program can't correctly calculate (and print) entropy using the first method and print a password strength message then it will fail most tests.

## Style Marking

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[11].

## 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[12] then your automated and human style marks will both be zero. If you use any global variables then your automated and human style marks will both be zero.

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

- $W$ be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)

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

[12] 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.

- $A$ be the total number of style violations detected by `2310stylecheck.sh` when it is run over each of your `.c` and `.h` files individually[13].

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.

[13] 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<br>all commit messages are meaningless. |
| 1 | Some progressive development evident (three or more commits) AND<br>at least one commit message is meaningful. |
| 2 | Progressive development is evident (multiple commits) AND<br>at least half the commit messages are meaningful |
| 3 | Multiple commits that show progressive development of almost all or all functionality AND<br>at least two-thirds of the commit messages are meaningful. |
| 4 | Multiple commits that show progressive development of ALL functionality AND<br>meaningful messages for all but one or two of the commits. |
| 5 | Multiple commits that show progressive development of ALL functionality AND<br>meaningful messages for ALL commits. |

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

## Total Mark

Let

- $F$ be the functionality mark for your assignment (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).
- $V$ is the scaling factor (0 to 1) determined after interview(s) (if applicable – see the CSSE2310 Student Interviews section below) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

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

out of 75.

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

## CSSE2310 Student Interviews

**The teaching staff will conduct interviews with a subset of CSSE2310 students about their submissions**, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you legitimately use code from other sources (following the usage/referencing requirements outlined in this assignment, the style guide, and the AI tool use documentation requirements) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment unless there are documented exceptional circumstances that prevent you from attending.

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

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

## Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum.

### Version 1.1

- Provided an alternate formula for computing entropy ($E_1$) in order to avoid overflow. See page 6.
- Fixed mistake in `uqentropy` usage messages on page 4

### Version 1.2

- Clarified the range of match numbers to be handled – see page 7.
- Clarified that non-printable whitespace characters are permitted in password entry files – these will separate passwords – see page 5.