COMP1028 Programming and Algorithm
Session: Autumn 2025
Group Coursework (25%)

Group Name	**We C your C**				
Group Members					
Name 1	**Hue Wei Feng**		**ID 1**	**20711677**	
Name 2	**Ian Yap Yu Sheng**		**ID 2**	**20723833**	
Name 3	**Valiant Tai Zi Pin**		**ID 3**	**20705238**	
Name 4			**ID 4**		
Name 5			**ID 5**		
Marks **Total (100)**	Program Functionalities (65)	Bonus (5)	Code Quality (10)	Presentation / Demo (10)	Report / Documentation (10)
Submission Date	**5/12/2025**				

1. Introduction

This project aims to implement a Toxic Text Analyzer in C programming language to perform toxicity analysis and general statistics analysis on text files.

The core aim is to process a given text file or a CSV file, compute essential word and character statistics, identify and quantify the frequency of toxic words, and provide various reporting and comparison functionalities.

The functionalities the current project achieved are:

- **File Loading and Preprocessing**: Allows loading of a .txt or .csv file and preparing its content for analysis.
- **Text Analysis**: Calculates general statistics such as total words, unique words, character counts (letters, digits, punctuation, whitespace), and average sentence length.
- **Toxic Word Detection**: Identifies and tracks the frequency of toxic words based on a toxic word dictionary text file.
- **Reporting**: Displays toxicity analysis, word frequencies, and an ASCII-based bar chart for the top 10 toxic words.
- **Persistent Storage**: Saves the detailed analysis results to a CSV file.
- **Comparison**: Compares the toxicity percentage of two previously saved analysis CSV files and finds out which file is more toxic.
- **Adding new words to dictionaries**: Allows users to add new words to the toxic words dictionary dynamically.
- **Menu-driven Interface**: Provides an interactive menu for easy program navigation.
- **Large file handling:** The current program is capable of running analysis on large files, though execution time increases proportionally.

2. Key Design Choices

2.1 Data structures used (arrays and structs)

- **Structs used:** Structs such as WordEntry and ToxicWordEntry are used for storing and tracking word frequencies. WordEntry holds the word and its frequency. ToxicWordEntry is similar, tracking toxic words specifically. Furthermore, the Analysis struct is used to store all the generated statistics from the current user-input file.

- **Dynamic Arrays of Struct are implemented:** Dynamically allocated arrays of structs for unique words and unique toxic words allow the program to process text files without being constrained by fixed buffer sizes.

- **Usage of Realloc:** Realloc is used when the initial capacity that is originally set is reached, then the uniqueCapacity or uniqueToxicCapacity is reallocated to double the size of the current memory to store more data. Doubling helps reduce the number of times needed to call the function realloc. For example, if we did realloc every time we reached the capacity and we still have 100 words left not yet processed, that means we need to use realloc 100 times.

2.2 File handling strategy

- **Temporary File to hold contents:** Contents of the user's chosen file are written to a temporary file named read.txt. This allows us to read from the read.txt file line-by-line using fgets. So that whether it's a CSV file or a text file, we will only have a single function to read the file contents without having two different functions (one for reading .csv file and one for reading .txt file).

- **Normalization during Load:** The words from both the toxic words and stop words dictionaries are converted to **lowercase** using tolower() during the loading process. This ensures that comparisons during the main analysis are case-insensitive.

- **Dynamic Output Naming:** Users can save the analysis results to a CSV file. The function will then generate the output filename based on the input file name. For example, document.txt becomes document_analysis.csv, and document1.txt becomes document1_analysis.csv. This allows multiple files to each have a different analysis output file.

- **Toxicity Analysis File Comparison:** The compareFileToxicity function implements a file reading strategy to compare the toxicity of two previously generated analysis files. Instead of analysing two files and comparing their data, the function reads previously generated _analysis files to compare the data of the two files. This helps negate the process of repeated analysis of files if the user just wants to compare toxicity between two files.

2.3 File Modularisation

- **Use of header files and .c files:** The Toxiclib.h serves as the public interface of the library. It contains all essential definitions required by the main program, including: constants, data structs and global variables. While the .c file(function.c) contains the actual logic, function bodies, and global variable definitions. Through the use of .c and .h files, we are able to improve the readability of the code without needing to add all the functions and functions logic in one large .c file.

2.4 Tokenization approach (punctuation removal, case folding, splitting etc.)

- **Case Folding:** All characters in the text are converted to lowercase using tolower() during the file processing loop in generalStats. This ensures that "Word," "word," and "WORD" are treated as the same token. Furthermore, this is also to allow the comparison of words with toxic words and stopwords, which were also converted to lowercase.

- **Punctuation Removal:** All punctuation marks are identified using ispunct(). Once counted, each punctuation character is immediately replaced with a space (' ') in the buffer. This guarantees that words attached to punctuation (e.g., "word!" or (word)) are correctly separated from the punctuation mark before the splitting phase begins.

- **Word Splitting:** Once the text is normalised, the function strtok() is used to split the modified buffer into word tokens. The delimiter string " \t\n\r" ensures that words are correctly separated by spaces, tabs, newline characters, and carriage returns.

- **Post-Tokenisation Cleanup:** After strtok returns a token, additional logic is used to remove any leading or trailing non-alphabetic characters (which might include numbers or remaining special characters), ensuring only the core alphabetical word is processed.

2.5 Stopword handling

- **Normalisation Stopwords:** The words are normalised to lowercase using the tolower() function during the loading process before storage. This pre-processing step guarantees case-insensitive matching when analysing the input text.

- **Linear Search Exclusion:** The isStopword function uses a direct linear search (strcmp) to check if any tokenised word is the same as the words in the pre-loaded dictionary. If a match is found, the word is immediately discarded, meaning it is excluded from both the totalWords count and the uniqueWords tracking list. This ensures that common words like 'the' or 'and' do not skew the unique words to total words ratio (lexical diversity).

2.6 Sorting Algorithm used and justification

- **Sorting Algorithm Used Is Bubble Sort**: While Bubble Sort is known to be an inefficient sorting algorithm for comparison, and is not the most efficient choice for large-scale data analysis (where Merge Sort would be preferred). However, it was selected for this project due to ease of implementation as its straightforward logic minimises the complexity of managing nested loops and struct swapping.

2.7 Toxic Word Detection Strategy

- **Normalisation Toxic Word:** Just like stopwords, these dictionary words are normalised to lowercase using tolower() during loading to ensure accurate, case-insensitive matching against the text tokens. The add_words feature allows users to permanently append new words to this dictionary file, enhancing the system's ongoing relevance.

- **Linear Search Comparison:** Also, same as Stopwords, it uses a direct linear search (strcmp) to check if any tokenised word is the same as the words in the pre-loaded dictionary. If a match is found, the global variable toxicword count is incremented. The word frequency is tracked using trackToxicWord, which manages the dynamic array uniqueToxicWords.

2.8 Menu-driven user interface design

- **ASCII Design:** The program uses an ASCII character interface, constructed using standard ASCII characters such as hyphens - to create visual separation, headers (like "TOXIC TEXT ANALYSER"), and structure. Furthermore, the bar chart created to visualise the top 10 toxic words also uses ASCII characters.

3. Challenges Faced and Lessons Learned

3.1 Challenges Faced

3.1.1 Memory Management and Freeing Pointers

A significant early challenge was the correct management of dynamic memory, particularly preventing memory leaks and ensuring capacity for large inputs. Initially, we faced limitations because we had not allocated sufficient memory space for the dynamically sized word lists (unique words and unique toxic words) to handle large text files.

Thus, we decided to use dynamic allocation strategy. We used malloc() for the initial allocation and implemented realloc() with a capacity doubling technique to dynamically expand storage for the unique word lists as needed. Furthermore, we decided to always look out for when to use free(), especially after any allocated buffer was no longer needed, such as the buffer used in GeneralStats, ensuring that every call to malloc or realloc was matched by a corresponding call to free to prevent memory leaks.

3.1.2 File handling

Reading the user's input file presented a hurdle, as the program needed to handle potentially different file types (e.g., .txt, .csv) without knowing the format in advance.

Therefore, in order to ensure the analysis function (generalStats) could rely on a single, normalised stream of raw text, we implemented the strategy of writing the input file's contents to a temporary working file(read.txt). This decision unified the input format, allowing the tokenisation logic to remain simple and consistent regardless of the original file extension without needing to create separate functions to analyse different file formats.

3.1.3 Time management

Due to the large amount of coursework during development, we had to carefully manage the project scope. This meant prioritising core functionality and essential robustness features over implementing highly advanced, high-complexity features.

Thus, we decided to focus our efforts on core functions first rather than pursuing secondary features like using a faster sorting algorithm (Merge Sort). This strategy ensured the deliverable was stable and functional.

3.1.4 Header files and variables standardisation

Working collaboratively (three people contributing to the code) became challenging when our development process initially lacked standardisation. Different members used different variable names for the same purpose, leading to integration issues and redundant code. Some members used global variables while some used a struct. When we decided to combine all of our codes, it showed errors for variables having multiple initialisations. It took a few days to standardized and remove the redundant variables.

Therefore, we decided to try to implement separate .c and .h files to make the code more modular and readable. The header file (Toxiclib.h) provided a single place to define and extern all global variables, constants, structs and function prototypes. This standardisation helps to make sure the variables we are using are standardised, and we can easily view what variables have been used without needing to check each file separately.

3.2 Lesson Learned

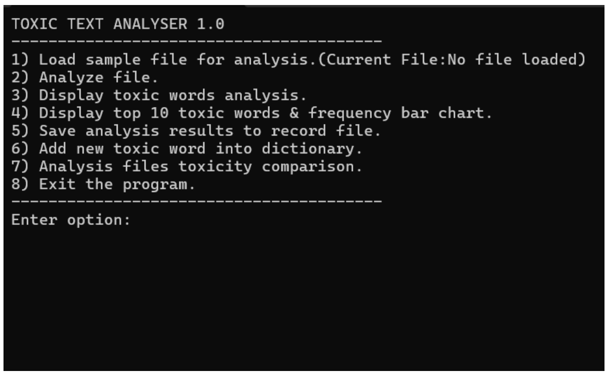
3.2.1 Time Management and standardisation of code

First and foremost, the early struggles with conflicting variable names across teammates highlighted the importance of standardising variables and communication during development. Furthermore, the time constraints of this semester taught us the importance of better project and time management, as we did not allocate enough time for this project, causing us to leave some advanced features out. Last but not least, in future we will try to implement a more efficient way to sort the words by frequency rather than using bubble sort.

3.2.2 Algorithms and C programming

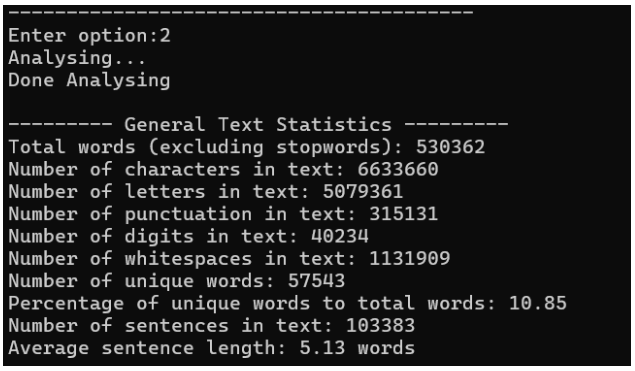
We learned to properly malloc, realloc, and free memory blocks after the buffer used was no longer needed for writing stable code and to prevent leaks. Moreover, we learned the benefit of using structs over scattered global variables, which helped improve code readability and allowed us to standardise the variables. In addition, we now understand the differences between reading and writing file types as different files types require different reading methods. Last but not least, we understand now that different sorting algorithms plays a huge role in sorting through large files and in future we will use more efficient sorting algorithms.

Appendices



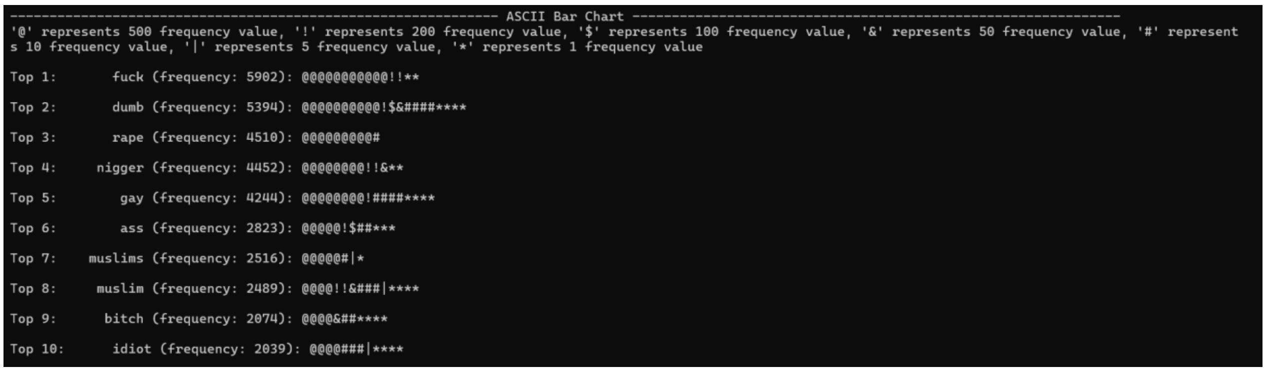
```
TOXIC TEXT ANALYSER 1.0
---------------------------------------
1) Load sample file for analysis.(Current File:No file loaded)
2) Analyze file.
3) Display toxic words analysis.
4) Display top 10 toxic words & frequency bar chart.
5) Save analysis results to record file.
6) Add new toxic word into dictionary.
7) Analysis files toxicity comparison.
8) Exit the program.
---------------------------------------
Enter option:
```

Figure 1: *Menu Interface for users*



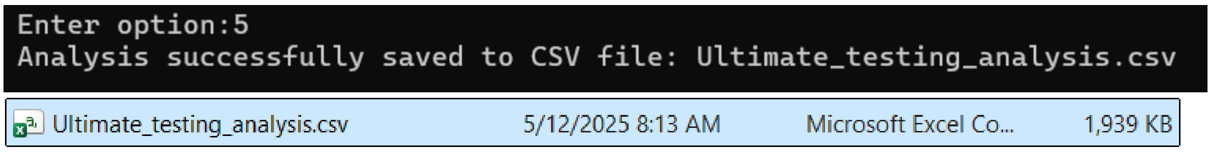
```
---------------------------------------
Enter option:2
Analysing...
Done Analysing

--------- General Text Statistics ---------
Total words (excluding stopwords): 530362
Number of characters in text: 6633660
Number of letters in text: 5079361
Number of punctuation in text: 315131
Number of digits in text: 40234
Number of whitespaces in text: 1131909
Number of unique words: 57543
Percentage of unique words to total words: 10.85
Number of sentences in text: 103383
Average sentence length: 5.13 words
```

Figure 2: *Large File Handling*



```
-------------------------------------------------------------- ASCII Bar Chart --------------------------------------------------------------
'@' represents 500 frequency value, '!' represents 200 frequency value, '$' represents 100 frequency value, '&' represents 50 frequency value, '#' represent
s 10 frequency value, '|' represents 5 frequency value, '*' represents 1 frequency value

Top 1:       fuck (frequency: 5902): @@@@@@@@@@@!!**

Top 2:       dumb (frequency: 5394): @@@@@@@@@@!$&####****

Top 3:       rape (frequency: 4510): @@@@@@@@@#

Top 4:     nigger (frequency: 4452): @@@@@@@@!!&**

Top 5:        gay (frequency: 4244): @@@@@@@@!####****

Top 6:        ass (frequency: 2823): @@@@@!$##***

Top 7:    muslims (frequency: 2516): @@@@@#|*

Top 8:     muslim (frequency: 2489): @@@@!!&###|****

Top 9:      bitch (frequency: 2074): @@@@&##****

Top 10:      idiot (frequency: 2039): @@@@###|****
```

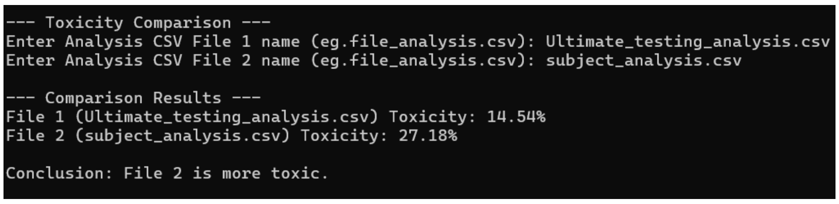
Figure 3: *Ascii Bar Chart*



```
Enter option:5
Analysis successfully saved to CSV file: Ultimate_testing_analysis.csv
```

Ultimate_testing_analysis.csv   5/12/2025 8:13 AM   Microsoft Excel Co...   1,939 KB

Figure 4: *File Analysis Output*



```
--- Toxicity Comparison ---
Enter Analysis CSV File 1 name (eg.file_analysis.csv): Ultimate_testing_analysis.csv
Enter Analysis CSV File 2 name (eg.file_analysis.csv): subject_analysis.csv

--- Comparison Results ---
File 1 (Ultimate_testing_analysis.csv) Toxicity: 14.54%
File 2 (subject_analysis.csv) Toxicity: 27.18%

Conclusion: File 2 is more toxic.
```

Figure 5: File Toxicity Comparison

Appendix A: ReadMe

Instructions to run our program:

Visual Studio Code:

1. Change your task JSON file
2. Change the args from "${file}", to "${fileDirname}\main.c", "${fileDirname}\functions.c",
3. Press run and debug

Visual Studio Community:

1. Add all the necessary files into one folder
2. Press run and debug

All files that must be included:

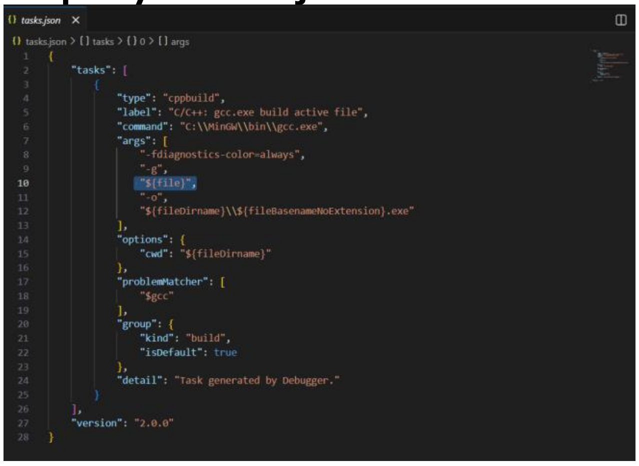
1. Stopwords.txt
2. ToxicWords.txt
3. main.c
4. Functions.c
5. Toxiclib.h

Sample Files for testing

1. Ultimate_testing.csv
2. subject.txt
3. Test1.txt
4. Test2.txt
5. Test3.txt

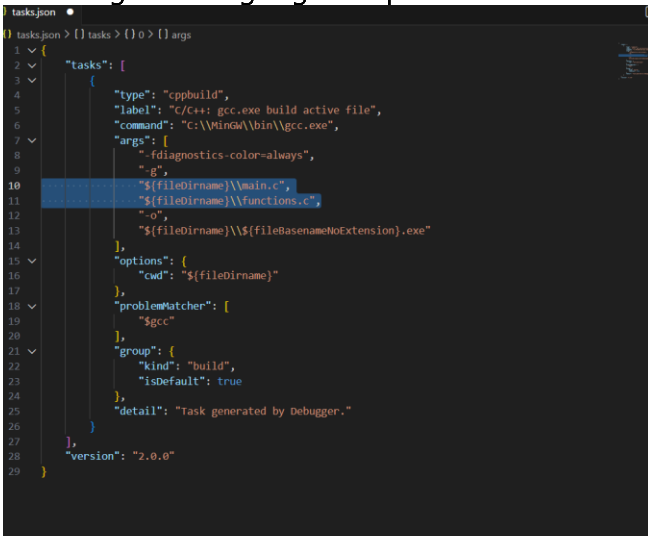
Vs Code changing Task Json file

1.Open your task json file



```
{
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: gcc.exe build active file",
            "command": "C:\\MinGW\\bin\\gcc.exe",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "${file}",
                "-o",
                "${fileDirname}\\${fileBasenameNoExtension}.exe"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "Task generated by Debugger."
        }
    ],
    "version": "2.0.0"
}
```

2.Change the highlighted part into



```
{
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: gcc.exe build active file",
            "command": "C:\\MinGW\\bin\\gcc.exe",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "${fileDirname}\\main.c",
                "${fileDirname}\\functions.c",
                "-o",
                "${fileDirname}\\${fileBasenameNoExtension}.exe"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "Task generated by Debugger."
        }
    ],
    "version": "2.0.0"
}
```

GitHub Link & Video Link

GitHub Repository:
https://github.com/Johnwicktheman/AssingmentProgrammingGroupWeCyourC

Video Presentation Link:
https://drive.google.com/drive/folders/1grZzSq-pST3jbg8dNTZUTdkBCwAWlWre?usp=sharing

Appendix B: Marking Scheme Summary

Criteria	Description / Notes
Text Input & File Processing	It can handle large files and support different input files
Tokenisation & Word Analysis	It can show average sentence length, lexical diversity and other statistics
Toxic Word Detection	Allows the expansion of toxic word dictionary
Sorting & Reporting	Used bubble sort and displays the Top 10 toxic words.

Persistent Storage	Allows the user to output the current results as an " _analysis.csv" file
User Interface	Allows the user to see the current Top 10 Toxic Word ASCII Bar chart and comparative reports between analysis files
Error Handling & Robustness	Handle wrong filenames and provide friendly user error recovery. Handle different sequences of options (e.g. Option 2 first before Option 1.
Code Quality & Modularity	Separated functions into .c and .h files to improve readability and standardise variables. Use of structs to store results
Bonus Features	Allows comparison between two _analysis csv files
Presentation / Demo	Explained the overall program
Report / Documentation	Explained the overall program