University of Nottingham
UK | CHINA | MALAYSIA

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

Group Name	**B&G**				
Group Members					
Name 1	**Zhao, Hengyi**		**ID 1**	**20792131**	
Name 2	**Wang, Jinbo**		**ID 2**	**20795326**	
Name 3	**Li, Lingyu**		**ID 3**	**20791542**	
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	**2025/12/5**				

1. Introduction

This project is a command-line text analysis tool developed in C. It is designed to process text files, provide statistical insights, identify potentially toxic language, and generate reports based on the analysis.

The primary aims of the implementation are:

File Processing: To reliably open and read user-specified text files as the source for analysis.

Statistical Analysis: To calculate and display key metrics, including total word count, line count, character count, and individual word frequencies.

Text Tokenization: To parse raw text into individual words (tokens), handling punctuation and case-folding to ensure accurate analysis.

Stopword Filtering: To identify and exclude common, low-value words (like "the", "a", "is") from the statistical results using a predefined stopword dictionary.

Toxic Word Detection: To scan the text for words present in a separate, predefined toxic word dictionary and report their occurrences.

Data Reporting: To present the analysis results to the user in a clear, structured format, with options for sorting data alphabetically or by frequency.

User Interface: To provide a simple, interactive menu-driven interface for ease of use.

2. Key Design Choices

Describe in concise way the major design decisions in your implementation (can be in point form). Suggested subsections include (delete sections not applicable to you):

2.1 Data structures used (arrays, structs, hash tables, etc.)

Structs: A central AnalysisResult struct is used to aggregate all data from a file analysis session. This includes file statistics, lists of word frequencies, and detected toxic words, providing a clean way to pass complex data between functions.

Dynamic Arrays (char **): Dictionaries for stopwords and toxic words are loaded into dynamic arrays of strings (char **). This approach is flexible, as the size of the dictionaries is not hardcoded and is determined at runtime based on the content of the dictionary files. The lines from the input file are also stored in a similar structure.

Array of Structs: An array of WordFrequency structs (e.g., struct WordFrequency { char *word; int count; }) is used to store unique words and their corresponding counts, which is essential for statistical analysis and sorting.

2.2 File handling strategy

The implementation uses standard C library functions (fopen, fgets, fclose) for all file I/O operations.

Files are read line-by-line using fgets into a dynamically allocated buffer. This strategy is memory-efficient, preventing the program from crashing when processing very large files that may not fit entirely into RAM.

File handling logic is modularized, with separate functions responsible for reading dictionary files versus the main text file to be analyzed.

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

Splitting: The strtok function is used to split each line of text into words based on delimiters like spaces, tabs, and newlines.

Case Folding: All tokens are converted to lowercase using the tolower function. This ensures that words like "Hello" and "hello" are treated as the same token, leading to more accurate word frequency counts.

Punctuation Removal: Punctuation is handled by including common punctuation marks in the strtok delimiter set or by iterating through tokens to strip them manually, ensuring that "word." and "word" are treated identically.

2.4 Stopword handling

A list of stopwords is loaded from an external file (stopwords.txt) at program startup.

During the word analysis phase, after a word is tokenized, it is compared against the loaded stopword list.

If a token is found in the stopword list, it is ignored and not included in the final word frequency statistics.

2.5 Toxic word detection strategy

Similar to stopwords, a dictionary of toxic words is loaded from toxicwords.txt.

Each token from the input text is compared (case-insensitively) against the toxic word dictionary.

When a match is found, the word and the line number of its occurrence are recorded in the AnalysisResult struct for later display.

2.6 Sorting algorithm(s) used and justification

The standard library's qsort function is used for all sorting tasks.

Justification: qsort is highly optimized, efficient (average-case O(n log n) time complexity), and part of the C standard library, making it a reliable and portable choice. It avoids the need to implement a sorting algorithm from scratch.

Custom comparison functions are supplied to qsort to enable sorting the word frequency list alphabetically (using strcmp) and numerically by frequency count.

2.7 Menu-driven user interface design

The user interface is built around a central loop in main() that continuously displays a list of options to the user.

User input is captured using scanf.

A switch statement maps the user's integer choice to the corresponding function call (e.g., '1' for loading a file, '2' for displaying statistics). This creates a simple, intuitive, and guided user experience.

3. Challenges Faced and Lessons Learned

Challenges Faced:

Memory Management: The most significant challenge was ensuring correct and disciplined memory management. Dynamically allocating memory for strings, arrays of strings, and structs required careful tracking with malloc, realloc, and free. Early versions of the code suffered from memory leaks due to forgetting to free allocated memory after use.

String Handling Complexities: C's null-terminated strings are powerful but require careful handling. Issues such as buffer overflows when using strcpy or gets (which was avoided in favor of fgets), and understanding the destructive nature of strtok, were key learning hurdles.

Robustness and Error Handling: Implementing checks for all possible failure points, such as a file not existing (fopen returning NULL) or memory allocation failing, was critical to making the program robust and preventing unexpected crashes.

Lessons Learned:

Importance of Modularity: Breaking the program into logical units (e.g., file_handler.c, text_processor.c, ui.c) made the code immensely easier to write, debug, and maintain.

Defensive Programming: We learned to anticipate errors, such as invalid user input or missing files, and to handle them gracefully rather than letting the program crash.

Pointers and Dynamic Memory: The project provided extensive, practical experience with pointers, dynamic memory allocation, and creating complex data structures like dynamic arrays of strings. This solidified theoretical concepts learned in class.

Collaborative Workflow: Working on a shared codebase highlighted the importance of clear communication, consistent coding style, and using a version control system like Git to manage changes effectively.

Reflect on what your group learned about C programming, algorithms, text processing, and working collaboratively.

Appendices (optional)

Include any additional materials, screenshots, tables, etc. (if any)

Appendix A: ReadMe

✧ Provide instructions for compiling and running your program. Example:
How to Compile

You will need the GCC compiler (MinGW-w64 on Windows). Navigate to the project's root directory in your terminal (like PowerShell) and run the following command:

- ***gcc.exe -g -Wall -I. -c *.c***
- ***gcc.exe *.o -o analyzer.exe***

This command compiles all .c files in the directory and links them together to create a single executable named ***analyzer.exe.***

✧ Required Dictionary Files

For the program to function correctly, the following files must be present in the same directory as the ***analyzer.exe*** executable:

- ***stopwords.txt:*** *A plain text file containing one stopword per line.*
- ***toxicwords.txt:*** *A plain text file containing one toxic word per line.*

✧ Example Usage:

Run ***analyzer.exe*** from the terminal:
The program will display a menu of options.
Enter the number corresponding to your desired action (e.g., '1' to analyze a new file) and press Enter.
Follow the on-screen prompts to provide filenames or choose further options.

✧ Dependencies

The project uses standard C libraries and does not have any external dependencies. A standard C compiler is all that is required.

GitHub Link & Video Link

GitHub Repository: ZHAOHENGYI-TECH/CyberbullyingTextAnalyzerCoursework: nottingham programming group project

Video Presentation Link: **http://youtube.com/watch?v=jE6mRDmvovo**

Appendix B: Marking Scheme Summary (Optional)

Criteria	**Description / Notes**
Text Input & File Processing	**We have become proficient in file input and processing. We can accurately read content from text and ultimately output a new text.**
Tokenization & Word Analysis	**Our design can accurately filter words from the word database and complete tasks such as sorting, using pointers and structures to accomplish this**.
Toxic Word Detection	**Our design can accurately identify toxic words in the target text and then apply sorting afterward.**
Sorting & Reporting	**We used three sorting methods, which are Sort by frenquency,Sort by alphabetically and Sort by toxicity level. They can sort the vocabulary and allow specifying a number N, displaying the top N words with the highest frequency.**
Persistent Storage	**Our design can generate a text file at the end and directly output the results.**
User Interface	**The user interface of our project is very clear and easy to understand, and we have established and outlined the steps for operation, using switch statements.**
Error Handling & Robustness	**We have checked the program's accuracy multiple times and found no syntax or logical errors, such as freeing memory promptly after using malloc to prevent memory damage.**
Code Quality & Modularity	**Our code has great advantages. We have used almost all the techniques, statements, and functions learned in the PROGRAMMING LAB class. The usage is simple and commented. We have tested the program multiple times, covering all branches, and the program is completely correct.**
Bonus Features	**We have 3-tier toxicity classification, severity(Mild, Moderate, Severe), and have calculate 1,2,3 to ever tier. We can detect toxic phrase, just like 'shut up', 'go to hell'.**
Presentation / Demo	**We recorded an explanatory video about this project, gave a presentation.**
Report / Documentation	**We wrote a very solid and well-structured report to introduce our designed project to the users. We explained many usages and their intricacies, with clear thinking throughout.**

Note: This may help to ease the marker on having an overview of your entire project in a glance.