University of Nottingham
UK | CHINA | MALAYSIA

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

Group Name	Bubble Tea				
Group Members					
Name 1	Meenakshi Murugappan	**ID 1**	20713959		
Name 2	Ge, Siyu	**ID 2**	20609637		
Name 3	Tiew Xen Xuen	**ID 3**	20715271		
Marks **Total (100)**	Program Functionalities (65)	Bonus (5)	Code Quality (10)	Presentation / Demo (10)	Report / Documentation (10)
Submission Date	5 December 2025				

Table of Contents

1.0 Introduction

Cyberbullying has become a serious concern in today's online environment, where harmful messages and toxic language can spread quickly across social media and digital communication platforms. Detecting such behavior early is important to protect users and create safer online spaces. This program involves creating a Cyberbullying / Toxic Text Analyzer in C, a simple tool designed to help identify harmful or offensive language in digital text. As online communication becomes more common, understanding and detecting toxic words is increasingly important for promoting safer and more respectful interactions. This program aims to support that goal by analysing text files and highlighting patterns that may indicate cyberbullying.

Scope of the Project:
- Reads and processes text or CSV files for analysis.
- Cleans the text by removing punctuation, converting words to lowercase, and filtering out common stopwords.
- Identifies toxic or abusive words based on a predefined list.
- Counts total and unique words, sentences, and characters.
- Calculates key statistics such as lexical diversity, average word length, and average sentence length.
- Highlights the most frequently used words and the most severe toxic terms.
- Allows users to interact through a simple menu and generate a full analysis report.

What the Implementation Added Aims to Achieve:
- Detect and measure the presence of toxic language in a given text.
- Provide users with a clearer understanding of text patterns linked to cyberbullying.
- Offer meaningful statistics that help interpret the tone and structure of the content.

2.0 Key Design Choices

This program required several important design decisions to ensure the text analyzer works smoothly, remains easy to understand, and fits within the limitations of C programming. The choices below explain how the program was structured and why the approaches were selected.

2.1 Data structures used

Data structures in the program involve:
- The program uses arrays to store words, stopwords, and toxic words because they are simple to manage and provide predictable memory usage in C.
- A custom struct (WordInfo) stores each word together with its frequency count and toxicity level, keeping the data organized and easy to work with.
- Integer variables are used to track important totals, such as the number of sentences, characters, and unique words.
- Counters like word_count, sentence_count, and character_count ensure that all statistics remain updated as the text is processed.

- Global variables are used so that different functions can share and update the same data easily, without needing to pass large structures around.
- Memory limits such as MAX_WORDS, MAX_STOPWORDS, and MAX_TOXIC are defined using #define, helping the program stay structured and preventing potential overflow issues.

2.2 File handling strategy

The file handling strategy in this program focuses on ensuring reliable reading of different file types while preventing errors and supporting text processing.

- The program uses standard C file functions such as fopen, fgets, and fclose to reliably read text files, stopwords, and toxic word lists.
- Before processing any file, the program checks whether the file exists or is empty. This helps prevent errors and ensures that users receive clear feedback if the file cannot be used.
- A dedicated helper function is included to safely extract text from CSV files, especially when dealing with quoted fields. This allows the program to correctly handle different CSV formats and avoid misreading content.
- The program uses fscanf to read toxic words and their severity levels from toxicwords.txt, ensuring that each toxic term is paired accurately with its corresponding severity value.
- File pointer validation is performed consistently to ensure the program does not continue if a file cannot be opened, to improve stability and preventing potential crashes.

2.3 Tokenization approach

The tokenization approach focuses on cleaning and breaking text into meaningful words while ensuring consistency for accurate analysis.

- Any unwanted characters such as punctuation, numbers, or symbols are ignored, and only alphabetic characters are retained when forming a word.
- All letters are converted to lowercase so that variations like "Hate" and "hate" are treated as the same word.
- The program marks the end of a word whenever it encounters a non-alphabetic character, ensuring clean separation between tokens.
- Sentence-ending symbols such as ., ?, and ! are counted to help estimate sentence boundaries and calculate average sentence length.

2.4 Stopword handling

Stopword handling ensures that common and low-value words do not interfere the meaningful text analysis.

- Stopwords are loaded from the external file stopwords.txt, allowing the list to be easily updated without modifying the code.
- Before a word is added to the main dataset, the program checks whether it appears in the stopword list.
- Words identified as stopwords are skipped entirely, preventing them from influencing frequency counts or skewing analysis results.

- This filtering step helps the analyser focus on more informative and meaningful words in the text.

2.5 Toxic word detection strategy

The toxic word detection strategy is designed to identify harmful or abusive language in a clear and consistent way throughout the analysis.

- Toxic words and their severity levels are loaded from toxicwords.txt, making it easy to update or expand the list whenever needed.
- Each word goes through the same cleaning process as regular words, ensuring that comparisons are fair and unaffected by uppercase letters or punctuation.
- The program uses fscanf to read toxic words along with their severity levels, guaranteeing that every term is matched with the correct intensity rating.
- As each word is processed, the program checks it using the get_toxicity() function, which returns the severity level if the word is considered toxic.
- Toxic words are tracked separately so the program can measure not only how often they appear, but also how severe the language is overall.
- Severity levels help distinguish between mild offensive words and more harmful expressions.

2.6 Sorting algorithm used and justification

The program includes a simple sorting approach to organize words by their frequency, making the analysis easier to analyse.

- A basic bubble sort algorithm is used to arrange words from most frequent to least frequent.
- Bubble sort was chosen because it is straightforward to implement and easy to understand, which fits well with the goals and scope of this project.
- Although bubble sort is not the fastest algorithm, its simplicity makes it suitable for the expected dataset size and avoids unnecessary complexity.
- Using this algorithm, ensures that the program can reliably identify the most common and most toxic words without adding extra functions.

2.7 Menu-driven user interface design

The program uses a simple menu-driven interface to guide users through its features, making the system easy to navigate for users.

- The interface presents a list of options, allowing users to load a file, view general statistics, check toxic word analysis, display the top N words, save a report, or exit the program.
- Each option is selected using a number, which keeps the interaction straightforward and avoids confusion.
- The menu runs inside a loop, enabling users to perform multiple actions without restarting the program.
- Input validation is included to handle invalid choices, ensuring that the program remains user-friendly.
- Clear prompts and messages help users understand what the program does at each step.

3.0 Challenges Faced and Lessons Learned

Developing the Cyberbullying / Toxic Text Analyzer came with several challenges, especially when text processing in C requires careful handling of memory and strings. There are few main difficulties encountered, and the key lessons learned throughout the project.

3.1 Challenges Faced

- **String handling was very complex**
  Working with strings in C was one of the biggest challenges in the project. C does not provide built-in functions for handling punctuation, token boundaries, lowercase conversion, or word cleaning. As a result, we had to manage these tasks manually, which required extra attention and careful coding to avoid errors.

- **Managing fixed-size arrays**
  Storing words, stopwords, and toxic words in fixed-size arrays meant working within strict memory limits. We had to ensure the program never exceeded these boundaries, as C does not prevent out-of-range access. This required careful planning and validation to avoid storing more items than the predefined limits, and to ensure the program remained stable and safe during processing.

- **Process of memory safety and avoiding overflow**
  As C does not provide automatic memory protection, we had to be extremely careful when using functions like strcpy, handling array indexes, and working near the boundaries of fixed-size arrays. Even a small mistake could lead to memory overflow, crashes, or unpredictable behaviour, making memory safety a constant concern. We were also worried that any small error in memory handling could affect the final results.

- **Handling large text files**
  Processing thousands of words from CSV and text files made it clear that the program needed efficient loops and careful checks. Without this, the analyzer could slow down or even fail to process the entire file correctly. This shows managing large inputs required extra attention to ensure smooth the program.

- **Implementing CSV file extraction correctly and properly**
  Passing quoted text fields in CSV files was challenging and required creating a custom function to handle them properly. We had to make sure the function could deal with edge cases such as commas inside quotes or unexpected formatting without breaking the program or misreading the content. This became one of the most challenging parts of the entire development process.

- **Ensuring consistent text cleaning**
  Making sure that stopwords, toxic words, and normal words all went through the same cleaning process was important to avoid mismatches. Without consistent cleaning such as converting everything to lowercase and removing unwanted characters the program

might treat similar words as different, leading to inaccurate results. Keeping this process the same way requires careful attention during the implementation.

- **Balancing simplicity and the function**
Choosing simple algorithms like bubble sort and avoiding overly complex features was necessary to stay within the project scope while keeping the code readable and manageable. We had to balance by adding useful functionality with maintaining a design that was easy to understand, debug, and extend. This required thoughtful decision-making throughout the development process.

3.2 Lessons Learned

- **Deeper understanding of C programming**
This project strengthened our understanding of core C concepts such as pointers, arrays, file I/O, and string manipulation. We gained firsthand experience with how these elements affect program behavior, efficiency, and stability, giving us a much clearer concept of how C operates behind the scenes. Through this process, we also developed a better understanding of how systems work underneath the user interface and how C controls these low-level operations.

- **Collaborative coding and problem-solving**
Working together allowed us to divide tasks, test each other's functions, and combine ideas to solve problems more effectively. As we explained our code to one another, we deepened our understanding of how the program worked and strengthened our ability to reason through C-related challenges. Collaboration made debugging easier and improved the overall quality of the final implementation.

- **Learning to manage strings manually**
This project required us to clean words, remove punctuation, convert text to lowercase, and check characters one by one. Through this process, we learned how important it is to be precise when working with strings in C, since the language does not provide high-level text-processing tools. By handling everything manually, it helps us to strengthen our understanding of how string manipulation really works in C.

- **Improving understanding of file handling**
Using fopen, fgets, fscanf, and pointer validation improved our understanding of how to safely read and process text file and CSV files in C. We also learned how to detect empty files using fseek and ftell, which helped prevent crashes and undefined behaviour. This experience showed us how important it is to handle file operations carefully, as even small mistakes can affect the entire process.

- **Extracting data from CSV files**

Writing a custom function to extract text from CSV lines taught us how to deal with real-world challenges using only basic C tools. Handling edge cases such as quoted fields, commas inside quotes, and irregular formatting helped us develop a stronger problem-solving mindset. This experience showed us how important it is to think carefully about input structure when working with file formats in C.

- **Understanding the trade-off between simplicity and performance**
  Using bubble sort in this project taught us that simple algorithms can still be effective. We learned that writing code that is easy to understand and maintain can sometimes be more important than achieving maximum efficiency. This experience helped us balance readability, correctness, and performance based on the actual needs and constraints of the project.

- **Building a functional menu-driven program**
  Implementing a loop-based menu taught us how to structure a program that accepts repeated user input, handles invalid selections, and guides the user through different features smoothly. This experience helps to strengthen our understanding of program flow control, input validation, and create a simple but effective user experience in C. It also showed us how important clear prompts and logical ordering when designing programs.

4.0 Debugging and Testing the Program

Throughout the development of the Cyberbullying / Toxic Text Analyzer, we used a combination of manual testing, print-based debugging, and incremental validation to ensure that every component of the program worked correctly. To evaluate the program with real-world data, we also tested it using a publicly available cyberbullying dataset from Kaggle: (https://www.kaggle.com/datasets/andrewmvd/cyberbullying-classification) **"Cyberbullying Classification"**. Below are the main approaches we used while debugging and testing the implementation.

- **Using print statements for step-by-step debugging**
  Since C does not have built-in debugging tools, we relied heavily on printf statements to monitor variable values at different stages of the program. This approach helped us trace issues in tokenization, word cleaning, stopword checks, and toxicity detection, allowing us to quickly pinpoint where the logic needed adjustment.

- **Testing file loading with different scenarios**
  We tested the program with a variety of input files to ensure that file handling was reliable and error-free. This included:
  - ➢ empty files (to confirm that fseek and ftell correctly detected them)
  - ➢ files with very long lines

➢ text files containing punctuation, numbers, or mixed cases

These tests helped confirm that the program could safely handle different file structures and edge cases without crashing or misreading the content.

- **Validate string cleaning and tokenization**
  We tested the clean_word() function by trying different types of words, including those with uppercase letters, punctuation, and special characters. This helped us verify that:
    ➢ unwanted characters were correctly removed
    ➢ words were consistently converted to lowercase
    ➢ token boundaries were recognized and split properly
    ➢ sentence-ending characters were counted accurately during processing

These tests ensured that the program handled text cleaning and tokenization properly before moving on to more complex analysis.

- **Testing the correctness of sorting**
  To ensure that the bubble sort algorithm was implemented correctly, we tested it using a variety of data sets. This included small lists to verify basic ordering, larger lists to evaluate performance and accuracy, and words with identical frequencies to see how ties were handled. Throughout these tests, we confirmed that the sorting function consistently produced the correct order and allowed the program to display the top words and top toxic terms accurately.

- **Verifying menu functionality**
  We also spent time verifying that the menu system worked smoothly by repeatedly testing each option. This helped us confirm that valid inputs triggered the correct functions, invalid inputs were handled gracefully, multiple actions could be performed without restarting the program, and the loop exited correctly when selecting "6. Exit." These tests showed that the menu-driven interface was user-friendly.

- **Cross checking report output**
  After saving the analysis report, we manually reviewed the generated analysis_report.txt file to ensure that:
    ➢ all statistics were printed accurately
    ➢ the toxic word analysis matched the results shown in the program
    ➢ the formatting was clear, structured, and easy to read
    ➢ the top word frequencies listed in the report were correct

These checks helped confirm that the saved output was reliable and consistent with what the program displayed on screen.

5.0 Conclusion

This project allowed us to apply C programming skills that we learned to build a working Cyberbullying / Toxic Text Analyzer. Through manual string handling, tokenization, file processing, sorting, and toxicity detection, we gained hands-on experience. The program successfully reads files, cleans and analyses words, detects toxic terms, and generates a clear report of the results.

Despite challenges such as managing fixed-size arrays, ensuring memory safety, and handling CSV formatting, we strengthened our problem-solving abilities and learned to design careful, step-by-step algorithms in C. Overall, the project deepened our understanding of how c programming supports real-world applications and improved our confidence in developing structured and reliable C programs. We also successfully completed a fully functioning program capable of identifying and analysing toxic words within a text.

Appendix A: ReadMe

ReadMe provides instructions on how to compile, run, and use the Cyberbullying / Toxic Text Analyzer program using Visual Studio Code.

1. Install the C/C++ extension (Microsoft).
2. Install MinGW (Windows) to provide the GCC compiler.
3. Open the folder containing main.c (or your file name, e.g., testt.c).
4. Open the Terminal in VS Code.
5. Compile the program using GCC:
   gcc main.c -o main.exe
6. After compiling, an EXE file (main.exe) will be created in the same folder.
7. Run the program by typing:
   ./main.exe
8. The program will start, and the menu-driven interface will appear.

Required Files:

Make sure the program is in the same folder as the following files:

- stopwords.txt – list of stopwords to ignore during processing
- toxicwords.txt – list of toxic words with severity values
- cyber.txt – a text file containing data extracted from the Kaggle dataset
  Dataset used:
  https://www.kaggle.com/datasets/andrewmvd/cyberbullying-classification
  (Any other .txt or .csv file may also be used)

If any of these files are missing, the program may show warnings or fail to process the analysis correctly.

GitHub Link & Video Link

GitHub Repository: https://github.com/Meena-2503/Cyberbullying-Toxic-Analyzer-Bubble-Tea-.git
Video Presentation Link:
https://drive.google.com/file/d/1_YlhUj_ozcxJhw3UCtiW9gNf2vyEUzPx/view?usp=drivesdk

Appendix B: Marking Scheme Summary

Criteria	Description / Notes
Text Input & File Processing	Successfully loads .txt and .csv files using fopen, fgets, and fscanf. Detects empty files using fseek and ftell. Handles file errors safely and provides clear messages when files cannot be opened.
Tokenization & Word Analysis	Words are processed manually only alphabetic characters are kept, and everything else is converted to lowercase to avoid mismatches. The program accurately counts words, characters, and sentences while ignoring unnecessary punctuation and symbols, to ensure clean and consistent analysis.
Toxic Word Detection	Toxic words and their severity levels are loaded from toxicwords.txt. Each processed word is checked against this list, allowing the program to track total toxic occurrences, severity levels, and the number of unique toxic words detected.
Sorting & Reporting	A simple bubble sort is used to organize words by frequency. The program then generates a detailed analysis_report.txt that includes statistics, toxic analysis, and the most frequent words, all formatted neatly for easy reading.
Persistent Storage	All analysis results word frequencies, toxicity levels, and general text statistics are stored in analysis_report.txt. This allows users to revisit the results anytime without rerunning the program.
User Interface	The menu-driven design allows users to navigate the program easily, perform multiple actions, and recover gracefully from invalid inputs. Clear instructions help guide the user step-by-step through the analysis process.
Error Handling & Robustness	The program checks missing files, prevents overflow by monitoring array bounds, and handles invalid menu choices. This helps the program run smoothly and avoid crashes.
Code Quality & Modularity	The code is organized into clear, well-named functions such as clean_word(), process_file(), and sort_by_frequency() making it easier to understand, maintain, and extend if needed.
Bonus Features	Extra touches include a custom CSV parser for handling quoted fields, sentence detection using punctuation, lexical diversity measurement, and calculations for average word and sentence length. Toxic words are also ranked based on severity for deeper insight.
Presentation / Demo	The final program demonstrates smooth workflow, logical processing, and meaningful output that reflects the text accurately. The results are easy to interpret and show clear evidence of correct implementation.
Report / Documentation	The program is supported by a detailed and well-organized report that explains the design choices, challenges, lesson learned, and testing approach clearly.