Cyberbullying / Toxic Text Analyzer in C

Reflective Report

COMP1028: Programming and Algorithms

Group name: xyz

Xie, Yifei: 20796507

Zhang, Chaoyuan:20791965

Tong Hao Ming: 20711051

Zhong, Xinyi: 20797574

GitHub Link:
https://github.com/bmtpow20060206/Group_xyz_COMP1028_coursework.git

1.Abstract

Our team collaboratively designed a standalone C program for cyberbullying text analysis, supporting single-file and batch processing of TXT files and complex CSV files (including files exceeding 100 MB). It is capable of providing word-frequency statistics, lexical diversity metrics, readability scores, and executing rule-based detection of toxic phrases with severity classification. By employing modular design, a separate-chaining hash table, multiple sorting algorithms, chunked large-file reading, and comprehensive error handling, the program's functionality fully meets all course requirements.

During the design process, we clearly identified several limitations of the current program: it does not support direct analysis of CSV files, requiring prior conversion to a TXT file for subsequent analysis; the toxic dictionary and stop-word dictionary have limited content coverage; the hash table uses a fixed size and lacks dynamic resizing support; console encoding relies on a Windows-specific workaround; and the recursive QuickSort poses a risk of stack overflow on extremely large inputs. These limitations highlighted the gap we observed between the acquisition of purely theoretical knowledge in computer science and its application in practical projects.

Through critical reflection on these shortcomings, we proposed concrete future improvement suggestions, including implementing dynamic hashing, adopting iterative sorting, integrating lightweight machine-learning techniques, and utilizing an established CSV parser. Beyond successfully demonstrating data structures and algorithms, all members gained deeper insight into the program's real-world reliability and the application of C language concepts in a functional system.

2. Key Design Choices

The architecture of the Cyberbullying Text Analyzer was fundamentally designed based on principles of modularity, performance scaling, and data robustness. The core system is logically partitioned into three key components: Analysis Logic (content.c), File Input/Output (I/O) (file.c), and Reactive Error Handling (error.c). Our design decisions were ultimately

guided by the goal of balancing rapid processing speed with the efficient and reliable presentation of data.

2.1 Data Structures and Performance Balance

Our data structure design aims to balance $O(1)$ insertion speed during the processing phase with $O(n \log n)$ sorting efficiency during the reporting phase.

Hash Table with Separate Chaining: A Hash Table utilizing a prime number size (10007) and separate chaining was implemented, achieving an average time complexity of $O(1)$ for word insertion and frequency updates.

Design Trade-off: The Hash Table employs a fixed size (HASH_TABLE_SIZE 10007) and lacks a dynamic expansion mechanism. While this choice simplifies implementation and guarantees stable average O(1) performance, it presents the limitation of potential data overflow and performance degradation if the number of unique words exceeds the maximum capacity.

Fixed-Capacity Array for Sorting: To facilitate high-performance sorting algorithms, we use the [hash_table_to_array] function to flatten the Hash Table's linked structure into a fixed-capacity (MAX_WORDS 10000) WordNode array.

Design Trade-off: Although this fixed-capacity array structure is suitable for high-performance, random-access dependent sorting algorithms (such as Quick Sort), it sets a strict upper limit on the maximum number of unique words the system can process.

2.2 Functional Modularity and Robust I/O

Functional decomposition focuses on fault tolerance and the processing of diverse inputs, especially in large file and complex CSV scenarios.

Text Normalization Pipeline: The [process_word] function calls [normalize_word] to implement preprocessing, uniformly stripping punctuation and forcing lowercase conversion, which ensures statistical accuracy.

Chunked Large File Reading: For files exceeding 5MB, we implemented the [read_large_file] function, which adopts an 8KB memory block chunked reading strategy,

effectively mitigating the risk of memory overflow when processing large inputs.

Custom CSV Parsing: A custom [csv_all_columns_to_text] parser was implemented to correctly handle complex formats, such as embedded commas within quotation marks, enhancing CSV file compatibility.

Limitation: Although the custom parser solves a specific problem, implementing a CSV parser from scratch in a learning project is often more prone to errors and less robust than using an established library, representing a critical area for future engineering improvement.

Reactive Error Handling: The (error.c) module is dedicated to fault tolerance, providing detailed error codes and user-friendly recovery options (e.g. handle_file_not_found_error).

2.3 Sorting Strategy and Benchmarking

The sorting strategy is designed to achieve maximal performance while providing transparent performance comparison.

Hybrid Sorting Implementation: Three algorithms were implemented: Quick Sort (default, average O(n log n)), Merge Sort (for stability), and Bubble Sort (for performance benchmarking).

Known Risk: While Quick Sort is highly efficient, its recursive nature presents a known risk of stack overflow when processing data of extreme magnitude.

Performance Comparison Feature: The unique [compare_sorting_algorithms] function actively demonstrates the performance disparity between $O(n^2)$ and O(n log n) algorithms on real-world text data, directly validating the selection of Quick Sort for the main report.

3. Challenges Faced and Solutions

We encountered three major difficulties and resolved them as follows:

3.1 Cross-platform Character Encoding and Mojibake Issues

Problem: During testing, we found that the default Windows command line used local encoding and could not correctly display special characters in UTF-8 text, resulting in content like Chinese being displayed as garbled characters (mojibake). Concurrently, text files generated by different editors might contain Byte Order Mark (BOM) headers, affecting the

correct parsing of file content.

Solution: We automatically execute the chcp 65001 command upon program startup to switch the Windows console to UTF-8 mode (as seen in init_console_encoding in tool.c). For file reading, we implemented encoding detection logic capable of identifying and skipping the UTF-8 BOM header, and providing explicit error messages for unsupported encodings like UTF-16. Although this solution requires adjustment for Linux and macOS (presenting a cross-platform portability limitation), it works effectively in our primary testing environment, Windows.

3.2 Unified Handling of TXT and CSV Files

Problem: CSV file parsing was more complex than anticipated, especially when field content itself contained commas (e.g. "hello, world"), where simple string splitting led to data corruption.

Solution: We implemented a custom CSV parsing function, [csv_all_columns_to_text], which correctly handles content enclosed in quotes. This parser tracks the quotation state, ensuring that splitting only occurs at commas outside of fields. Although we found that a few boundary cases require further refinement in actual usage, this implementation fundamentally achieved reliable conversion from CSV to plain text.

3.3 Dynamic Memory Management and Resource Release

Problem: When consecutively processing multiple files, the program failed to promptly clean up old analysis data (like the hash table and text buffers), leading to segmentation faults or continuous memory growth.

Solution:

1.We managed string concatenation by implementing the [safe_strcat] function, which automatically handles memory reallocation (realloc) and necessary boundary checks, preventing buffer overflow issues.

2.A core centralized cleanup function, [cleanup_analyzer], was introduced. Before each new analysis starts, this function systematically frees all nodes in the hash table and all dynamically allocated text buffers, effectively solving the resource unreleased problem and

ensuring the stability of program execution.

4.How We Debugged and Tested Our Program

To ensure both the accuracy and robustness of the analyzer, we adopted a multi-layered strategy combining defensive programming with systematic unit and integration testing.

4.1 Debugging and Defensive Coding

We focused on preventing errors at the source:

Memory Safety: By checking for NULL pointers immediately after every memory allocation call (e.g. in safe_strcat), we ensured the program reports the error promptly and halts execution upon memory exhaustion, thereby preventing a crash.

Resource Management: We introduced [cleanup_analyzer] to centralize resource deallocation, which systematically resolves resource leaks by freeing the hash table and all dynamic buffers.

Structured Errors: Using the standardized ErrorCode and handle_error function provides clear diagnostic messages (e.g., ERROR_FILE_CORRUPTED), significantly accelerating the troubleshooting of file I/O issues.

4.2 Core Algorithm and Performance Validation

We isolated and validated the critical computational components:

Sorting Performance: The dedicated [compare_sorting_algorithms] feature empirically proved the significant efficiency advantage of the default Quick Sort (O(n log n) average) over Bubble Sort (O(n2)).

Analysis Logic: We validated [normalize_word] to ensure data normalization consistency; we tested [detect_toxic_phrases] to confirm accurate identification of multi-word toxic phrases.

4.3 Integration and System Testing

We conducted end-to-end testing to verify the reliable coordination between modules:

Heterogeneous Files: We verified that the [csv_all_columns_to_text] custom parser correctly

preserved sentence structure in complex CSV files.

Large File Stress: We confirmed that [read_large_file's] 8KB chunking mechanism effectively prevented heap memory overflow during loading.

Encoding Compatibility: We verified that [init_console_encoding] successfully set the console output to UTF-8 encoding in the Windows environment, thereby improving the display issues with special characters.

5.Lessons Learned about Problem-Solving in C

Throughout this project, we developed a clearer and more practical understanding of problem-solving in C. Many of the challenges we encountered did not arise from algorithmic logic itself, but from memory management, pointer handling, and boundary conditions. Operations such as string concatenation, freeing hash-table nodes, and managing file buffers required meticulous attention. A single missing deallocation or null-pointer check was sufficient to cause system instability, reinforcing the notion that in C, ensuring safety can be more demanding—and more crucial—than implementing the intended functionality.

We also observed that the stability of the program is heavily influenced by the characteristics of the input data. Quoted fields in CSV files, inconsistent file encodings, and large file sizes made features that initially seemed straightforward significantly more complex. Addressing these issues required proactive validation and explicit error-handling mechanisms rather than assuming ideal or well-formed input. This experience broadened our understanding of what reliable software must account for in real use cases.

Furthermore, many commonly used functions—such as text normalization, CSV parsing, and structured error reporting—had to be manually implemented. This compelled us to think more deliberately about module separation and interface design to prevent the system from becoming difficult to maintain. Although this increased the implementation effort, it contributed to a more disciplined and organized development process.

Finally, we gained a deeper appreciation for the trade-offs between performance and implementation choices. The use of fixed-size hash tables, recursive sorting algorithms, and chunk-based file reading demonstrated that theoretical optimality does not always translate into practical effectiveness. Many decisions required balancing computational efficiency, memory constraints, and operational stability.

Overall, this project highlighted that effective problem-solving in C extends beyond algorithmic correctness. It demands careful handling of resources, rigorous validation of edge cases, and an ongoing awareness of system limitations. Many of the solutions we arrived at were the result of examining failures and understanding their root causes, rather than simply achieving functional outputs.

6. README Instructions

1. System Requirements：

C compiler - GCC and A terminal or command prompt

2. The source code files located in the same directory:

main.c，content.c/h，file.c/h，tool.c/h，error.c/h，stopwords.txt，toxicwords.txt

3. Compilation

gcc -o analyzer.exe main.c file.c conent.c tool.c error.c

4. Running the Program

Windows users run analyzer.exe directly

When launched, the program displays a main menu that allows users to load files, analyze text, generate reports, compare files, and configure settings.

5. Supported Input Files

.txt — Plain text

.csv — Structured text (comma-separated values)

5.1 Automatic File Validation

The system automatically checks whether the file: Exists，Is empty，Is damaged or unreadable，Uses a supported encoding

If issues are detected, the program displays error messages

6. CSV Automatic Conversion

When the user inputs a CSV file, the program performs an automatic extraction and conversion process:

1. Reads all textual content from the CSV file
2. Converts it into a TXT file with the same base name
3. Saves the generated TXT file in the current directory
4. Displays a message informing the user that conversion is complete

Important Note： After the CSV-to-TXT conversion, the user must re-run the program and load the newly generated TXT file to proceed with analysis.

7. Basic Usage Procedure

Step 1 — Load Data

1. Choose **Option 1** to load a single file (e.g., data.txt).

- The system automatically checks whether the file exists.

2. Or choose **Option 2** to load multiple files by entering space-separated filenames for batch processing.

Step 2 — Visualize and Analyze Data

After data is loaded, users can view analytical results:

1. Option 3: Display core statistics

- Word count, sentence count, reading level, toxicity density

2. Option 4: Show Top-N high-frequency words

- Uses sorting algorithms to rank words

3. Option 7: View a structured ASCII table summarizing stats, lexical diversity, and toxicity

4. Option 10: Generate ASCII charts

- Word-frequency bar chart
- Toxicity-severity pie chart

Step 3 — Save Results (Optional)

- Users may export the analysis:

1. Option 5: Save both analysis_report.txt and toxicity_report.txt

2. Option 6: Save toxicity results only

3.Option 8: Produce a comprehensive report with all metrics included

Step 4 — Access Advanced Tools

1.**Option 9**: Compare two files side-by-side

- Results saved to comparison_report.txt

2.**Option 11**: Adjust system settings

- Enable/disable ASCII charts
- Toggle auto-saving
- Other configuration options

7. Video recording link

https://www.youtube.com/watch?v=VuxmmrJ5BFs

8. Data Sources for Toxicity and Stopword Lexicons

Stopword source: https://www.cs.cmu.edu/~biglou/resources/bad-words.txt

Origin of toxic words: https://www.kaggle.com/datasets/rowhitswami/stopwords

Analyze the source of the text:

https://www.kaggle.com/datasets/andrewmvd/cyberbullying-classification