Contents

1 Objective

In Assignment 1, you developed a text file tokenizer that organized tokens based on their starting letter and recorded the line numbers where each token was found. A key aspect of that assignment was managing memory directly – you had to explicitly tell the program when to reserve memory and when to release it. While this gave you a fundamental understanding of memory management in C++, it also introduced complexities and potential for errors like memory leaks.

Assignment 2 builds directly upon the functionality of Assignment 1. Your task remains the same: to read a text file, break it down into tokens, index these tokens by their first character, and keep track of the line numbers where they appear.

However, the core difference in this assignment is how you will implement this. Instead of using custom-built data structures and manually managing memory, you will now leverage the power and safety of the C++ Standard Library. This means replacing your custom solutions with pre-built efficient components like containers (such as `std::vector`, `std::list`, `std::string`) and using *iterators* to navigate and manipulate data within these containers.

The primary objectives of Assignment 2 are twofold:

- **Achieving Functional Equivalence with Modern C++:** Your program should produce the same output as your Assignment 1 solution for the same input files. This ensures that the core logic of tokenizing and indexing remains consistent.
- **Embracing RAII for Safety:** You will learn to design your code following the RAII principle.[1] This modern C++ technique ensures that resources, especially memory, are automatically managed by tying their lifecycle to the lifespan of objects. By using Standard Library containers, you will inherently benefit from RAII, leading to more robust and safer code that is less prone to memory leaks. Furthermore, this assignment will provide you with practical experience in performing tasks like searching for data and inserting it in a sorted manner manually using iterators within these Standard Library containers. This will deepen your understanding of how these fundamental tools work.

[1] RAII, which stands for Resource Acquisition Is Initialization, is a core principle in modern C++ designed to make resource management, especially memory management, safer and more automatic. Think of it this way: when your program needs a resource – like a block of memory to store your tokens or a file to read – you acquire that resource when you create an object. The key idea of RAII is to tie the *lifetime* of this resource to the *lifetime* of that object.

When the object is created (during its initialization), it takes control of the resource. Crucially, when the object is automatically destroyed (for example, when it goes out of scope at the end of a function or a block of code), it automatically releases the resource it was managing.

This automatic management of resources through RAII greatly reduces the risk of common programming errors like memory leaks. You no longer have to remember to manually deallocate every piece of memory you allocate because the objects themselves take care of it. By using Standard Library containers, you are essentially leveraging the RAII principle without having to write complex memory management code yourself. This makes your code cleaner, more robust, and easier to reason about, allowing you to focus on the core logic of your tokenizer and indexer rather than the intricacies of manual memory management.

By completing Assignment 2, you will not only create a more robust and maintainable version of your tokenizer but also gain crucial experience in transitioning from manual memory management to the best practices of modern C++. This is a fundamental step in becoming a proficient C++ developer.

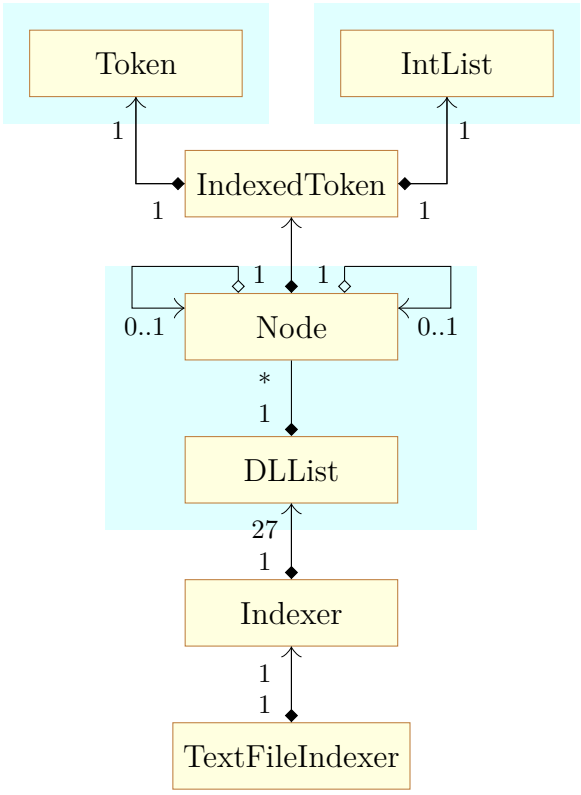
2 Contrasting Assignment 1 and Assignment 2: Embracing Modern C++

A fundamental shift in approach distinguishes Assignment 2 from Assignment 1. While Assignment 1 provided a valuable exercise in understanding low-level memory management through the implementation of custom data structures, Assignment 2 focuses on leveraging the power and safety of the modern C++ Standard Library to achieve the same functional requirements.

In Assignment 1, the program's architecture heavily relied on custom-built classes that necessitated manual dynamic memory management.

For example, the `Token` class directly manipulated raw C-strings using manual `new[]` and `delete[]` operations to store token data. This approach, while illustrating pointer usage, carried the inherent risks of memory leaks if `delete[]` was forgotten and potential buffer overflows if string lengths were not carefully managed.

Similarly, the `IntList` class managed a dynamic array of integers, requiring manual resizing (allocating new memory, copying elements, and deallocating the old memory) and error-prone manual index tracking. This introduced complexity and the possibility of out-of-bounds access if index checks were not implemented correctly.



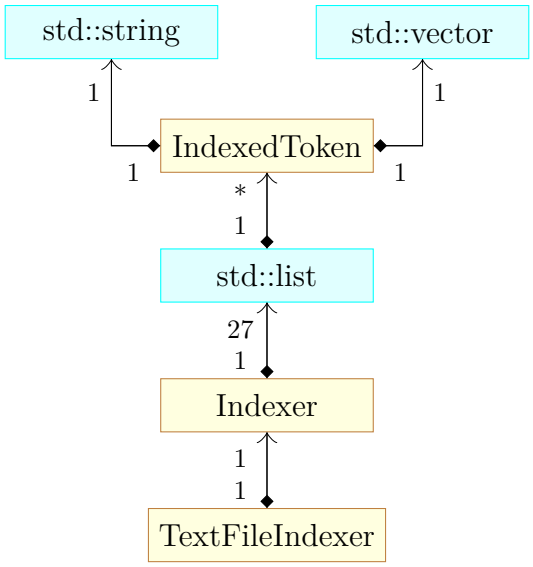
The `DLList` class implemented a doubly linked list from scratch, involving intricate pointer manipulation for node insertion and deletion. While this exercise deepened understanding of linked list structures, it was also complex and prone to errors such as broken links or memory leaks if nodes were not correctly managed.

While this hands-on approach in Assignment 1 reinforced foundational concepts such as pointers, dynamic arrays, and custom linked lists, it also inherently introduced risks like memory leaks (forgetting to free allocated memory), dangling pointers (accessing memory that has already been freed), and added unnecessary complexity in managing data structures.

In Assignment 2, these custom classes are entirely replaced by their robust and memory-safe counterparts from the modern C++ Standard Library.

The `Token` class is eliminated in favor of `std::string`. This offers automatic memory management, ensuring that memory is allocated and deallocated as needed without explicit programmer intervention.

Additionally, `std::string` provides a rich set of built-in methods for string operations like comparison, concatenation, and searching, simplifying the code.



The `IntList` class is replaced by `std::vector<int>`. This dynamic array handles memory resizing automatically, growing or shrinking as needed. It also provides built-in bounds checking (especially when using methods like `.at()`), significantly reducing the risk of out-of-bounds errors and simplifying data storage.

The `DLList` class, along with its internal `Node` implementation, is replaced by `std::list`, a standardized doubly linked list that ensures efficient insertions and deletions at any point in the list. By leveraging iterators for these operations, it eliminates the need for manual pointer manipulation, making the code cleaner and less error-prone. Here, `IndexedToken` represents a structure or class that holds a token (as a `std::string`) and its associated line numbers (as a `std::vector<int>`).

The `Indexer` class, which originally managed a raw array of 27 custom `DLList` objects, will now use a `std::array<std::list<IndexedToken>, 27>`. Using `std::array` ensures compile-time size checks (preventing accidental out-of-bounds access at compile time) and stack allocation for this fixed-size collection (potentially improving performance).

The use of `std::list<IndexedToken>` within each element simplifies the code by leveraging iterators for sorted insertions of `IndexedToken` objects based on their first character, and again, eliminates the need for manual memory management for the lists themselves. The fixed size of 27 corresponds to the 26 letters of the alphabet plus one additional slot for tokens starting with non-alphabetic characters.

Again, by undertaking this transition in Assignment 2, you will gain invaluable experience in leveraging the power and safety of the C++ Standard Library. You will learn how to replace custom, error-prone implementations with well-tested, efficient, and memory-managed components, which is a fundamental skill in modern C++ development.

3 Beyond Our Specific Choices: Exploring Alternatives in the C++ Standard Library

It's important to recognize that the specific C++ Standard Library components we've highlighted as replacements for your Assignment 1 custom classes are just some of the many powerful tools available. Our choices were guided by selecting standard containers that closely mirrored the functionality you implemented manually, making the transition more intuitive. However, the Standard Library offers a rich landscape of options, and different choices might be suitable depending on specific performance requirements or design considerations.

For instance, when considering how to store the line numbers for each token (the functionality of your `IntList`), `std::vector<int>` is a natural replacement due to its dynamic array nature and efficient storage of sequential data. However, you could also have considered:

- `std::list<int>` : A doubly linked list of integers. While offering efficient insertion and deletion at any point (like your custom `DLList`), it might have slightly higher memory overhead per element and potentially slower random access compared to `std::vector`.
- `std::deque<int>` (double-ended queue): Provides efficient insertion and deletion at both the front and back, as well as relatively fast random access. It could be a viable alternative if the order of line number insertion or retrieval had specific front/back requirements.
- `std::set<int>` : A container that stores unique integers in a sorted order. If the requirement was only to track the "unique" line numbers where a token appeared, a `std::set<int>` could be an efficient choice, automatically handling duplicates and keeping the line numbers sorted.
- `std::forward_list<int>` : A singly linked list of integers. If the primary operations involved were insertion and deletion at the beginning of the list (or after a known position) and forward traversal, `std::forward_list` could be a more memory-efficient alternative to `std::list`. It has less overhead per element since it only needs to store a pointer to the next element, not the previous one. However, it does not provide efficient backward traversal or direct access to the end of the list.

Similarly, for the `Indexer` class, which organizes tokens into 27 sections, we suggested `std::array<std::list<IndexedToken>, 27>`. While `std::array` is a good fit for a fixed-size collection known at compile time, other options exist:

`std::vector<std::list<IndexedToken>>` : A dynamic array of lists. If the number of sections needed was not fixed at 27 or could potentially change, `std::vector` would offer more flexibility in resizing.

`std::map<char, std::list<IndexedToken>>` : A map where the key is the first character of the token and the value is a list of `IndexedToken` objects starting with that character. This could provide more direct and potentially faster access to the list of tokens based on their starting character, although it might have a slightly different

underlying structure than a simple array.

Finally, for storing the sequence of `IndexedToken` objects within each of the 27 sections (replacing your custom `DLList`), we chose `std::list<IndexedToken>` due to its efficient insertion and deletion, which is likely important for maintaining a sorted list of tokens. Alternatives could include:

`std::vector<IndexedToken>` : If insertions and deletions were primarily at the end, or if random access to tokens was a frequent operation, `std::vector` might be more performant overall due to contiguous storage. However, inserting in the middle while maintaining sorted order would involve more overhead (shifting elements).

`std::deque<IndexedToken` : Similar to `std::vector` but with efficient insertion/deletion at both ends.

The key takeaway here is that the C++ Standard Library provides a rich set of tools, each with its own strengths and weaknesses in terms of performance characteristics (like insertion, deletion, access time) and memory management. In Assignment 2, we've chosen containers that closely align with the basic structure and primary operations of the custom classes you built in Assignment 1 to make the transition smoother. As you become more proficient with C++, you'll learn to analyze the specific requirements of a task and choose the most appropriate Standard Library components to achieve optimal results.

4 Implementation Guidelines and Constraints

- **STL Containers:** You must use `std::string` (for tokens), `std::vector<int>` (for line numbers), `std::list<IndexedToken>` (for sections), and `std::array<std::list<IndexedToken>, 27>` (to hold the sections) as specified.
- **RAII**: Rely entirely on RAII for resource management provided by the standard library containers. No manual `new`, `delete`, `new[]`, or `delete[]` calls are allowed or necessary.
- **Iterators:** Use standard library iterators (e.g., `list::begin()`, `list::end()`, `vector::cbegin()`, `vector::cend()`, `++it`, `*it`) or range-based for loops extensively for container traversal, searching, finding insertion points, and display.

  Use `const_iterators` where appropriate (e.g., in `const` member functions).
- **No STL Algorithms:** Functions from the `<algorithm>` header (like `std::sort`, `std::find`, `std::find_if`, `std::lower_bound`, `std::copy`, `std::unique`, etc.) are strictly prohibited.
- **No Lambdas:** Lambda expressions are strictly prohibited.

5 Deliverables and Submission Process

5.1 Submission Instructions:

Please refer to the course outline for the specific submission guidelines you must follow.

5.2 Required Files:

Ensure your submission includes all the following files:

- `IndexedToken.h`, `IndexedToken.cpp`
- `Indexer.h`, `Indexer.cpp`
- `TextFileIndexer.h`, `TextFileIndexer.cpp` (provided starter code)
- `README.txt`
- `main.cpp` (from Assignment 1)

6 Grading Criteria

Your assignment will be evaluated based on the following criteria:

1. **Correctness** (50%):
   - **Design & Requirements Adherence:** Implementation strictly follows specified design (class structure, function signatures, relationships) and fulfills all functional requirements.
   - **Accurate Functionality:** Program correctly reads and tokenizes input, processes data, and generates output as specified.
   - **Robust Edge Case Handling:** Graceful handling of empty files, punctuation, case (in)sensitivity (if required), invalid input formats, and boundary values without crashing.
2. Modern C++ Usage (30%): Correct use of
   - `std::string` for `Token`
   - `std::vector` for `IntList`
   - `std::list` for each section.
   - `std::array` for `Indexer` sections.
   - defined/`default`ed/`delete`d special member functions (Rule of Five).
   - Effective use of iterators for traversal/search/display as required.
3. **Code Quality** (20%):
   - **Clear and Consistent Code:** Ensure code is well-formatted for easy reading with consistent indentation and style. Use descriptive names for variables and functions. Add helpful comments for complex parts, but don't over-comment obvious code. Follow a defined style guide.
   - **Organized Structure:** Separate class declarations (.h) from implementations (.cpp). Make sure the project compiles correctly from these separate files. Use namespaces to keep code organized.
   - **Handles Errors Gracefully:** The program shouldn't crash on common errors like missing files or bad user input. Implement proper error handling and ensure

memory is managed correctly to prevent leaks.

- **Reasonable Efficiency:** While correctness is key, avoid significantly inefficient algorithms if simpler, faster options are available.
- **Modular and Simple Functions:** Design functions to do one thing well and avoid overly complex logic. Break down complex parts within classes using `private` helper functions.
- **Proper Use of const:** Apply `const` to functions that don't change the object and to references/pointers that shouldn't be modified.
- **Encapsulated Data:** Avoid global variables. Keep data within classes or pass it as needed to functions. Use access modifiers to control what can be accessed.