Assignment 3 Journal

Assignment 3 Journal: File I/O and Array Problems

Name: Boris Bojanov

ID: 3608550

Date Started: October 30, 2024

Date Completed: November 27, 2024

Overview of the Assignment Journal

In this journal, I chronicle my step-by-step progress for Assignment 3, which focuses on C++ file I/O, array manipulation, and template programming. This includes my planning, design decisions, testing approach, and reflections on each of the six (6) problems. Documenting my thoughts and challenges throughout the process proved essential in efficiently managing my time and deepening my understanding of the course material.

Daily/Weekly Progress and Activities

Week 1 (Oct 30–Nov 2): Reviewing Requirements and Setting Up

1. **Initial Reading**
    - Thoroughly read the Assignment 3 description, taking note of the six (6) distinct problems:
        1. Counting words in a file
        2. Displaying file contents one line at a time
        3. Implementing a `TextFileReader` class

4. Performing floating-point array operations
5. Extending the `Book` class and creating a `Bookshelf`
6. Implementing a custom `Set` template

- Determined that a large part of this assignment would involve file I/O in C++ streams, plus revisiting OOP concepts from previous assignments.

2. **Gathering Resources**
   - Consulted DevDocs C++ for references on file handling (`ifstream`, `ofstream`, `getline`, etc.) and template programming.
   - Looked up various forum discussions about user-friendly file I/O prompts, to ensure best practices in error handling.
3. **Environment Setup**
   - Confirmed my compiler (MinGW on Windows 10) was working properly and could handle file operations smoothly.
   - Created a folder structure with separate subfolders for each problem to keep my code organized.

Reflection: Taking time to plan each problem's implementation details helped me avoid confusion later. Understanding the requirements thoroughly is crucial before diving into coding.

Week 2 (Nov 3–Nov 7): Problem 1 – Counting Words in a File

1. **Design Decisions**
   - Chose to prompt the user for the file name rather than use command-line arguments, as it felt more user-friendly.
   - Decided to store the file content line-by-line, then split by whitespace for counting words, ensuring robust handling of different text inputs.
2. **Process and Challenges**
   - **Implementation**:
     - Used a simple loop with `getline()` to read each line.

    - Employed a string stream or manual parsing to split each line into words.
    - Incremented a counter for each word found.
  - **Challenges**:
    - Needed to handle edge cases, such as empty lines or lines with multiple spaces. It was somewhat tricky for me because I did consider every edge case during implementation.
    - Ensured the program handled a missing or invalid filename by displaying an error message and exiting.
3. **Testing**
  - Tested using the given `excerpt.txt` file and confirmed that the total word count made sense compared to manual checks.
  - Tried smaller custom text files to see if blank lines or punctuation affected the count.
4. **Reflections**
  - This problem refined my knowledge of file input, error handling, and string parsing in C++.
  - Also reminded me to always check for edge cases in textual data.

Connection to Learning Outcomes: Improved my confidence working with file streams and string operations—key components of C++ programming.

Week 3 (Nov 8–Nov 10): Problem 2 – Display File Contents One Line at a Time

1. **Design Decisions**
  - Elected to prompt user input for the file name.
  - Decided to use `cin.get()` to pause after printing each line.
2. **Implementation and Challenges**
  - **Implementation**:

    - Opened the file in read mode (`ifstream`).
    - Read each line with `getline()`, displayed it, then waited for `Enter` before printing the next line.
- **Challenges**:
    - Ensuring cross-platform consistency. Windows and Unix-based systems may handle newline characters differently.
    - Added a small prompt (e.g., "Press <Enter> to see the next line...") to give the user clearer instructions.

3. **Reflections**
    - Learned the importance of user experience in console applications—small details like prompts and clear instructions enhance usability.
    - Reinforced how to handle file I/O line by line with incremental user interaction.

Connection to Learning Outcomes: Practiced controlling program flow and integrating user input for improved interactivity.

Week 4 (Nov 11–Nov 14): Problem 3 – TextFileReader Class

1. **Design Decisions**
    - Created a `TextFileReader` class with:
        - An internal `std::string` array of size 100 for storing lines.
        - Two constructors (default and one that takes a filename).
        - Member functions: `contents()` (returns a combined string), `display()` (prints lines with line numbers).
    - Created a `TextFileReaderDemo` class with a `main()` that accepts command-line arguments (for filename).
2. **Process and Challenges**
    - **Implementation**:

    - The second constructor automatically opens the file and reads contents into the array.
    - `contents()` merges the stored lines into a single `std::string` or `std::stringstream`.
    - `display()` prints each line with format “line i: <text>”.
  - **Challenges**:
    - Handling lines beyond the 100th entry. For now, I simply stopped reading after 100 lines, as per requirements.
    - Ensured the line numbering started at 1, not 0.
3. **Testing**
  - Used the `excerpt.txt` file, tested the program with command-line arguments (e.g., `TextFileReaderDemo excerpt.txt`).
  - Confirmed lines were read correctly and displayed with correct line numbers.
4. **Reflections**
  - This problem showed the power of bundling related file operations within a dedicated class.
  - Reinforced constructor usage in practical contexts (e.g., automatically reading a file upon object creation).

Connection to Learning Outcomes: Applied OOP concepts with constructors, classes, and method organization to enhance code readability and maintainability.

Week 5 (Nov 15–Nov 18): Problem 4 – Floating Point Array Operations

1. **Design Decisions**
  - Planned to create three floating-point arrays of size 25.
  - First array stores loop counters (0 to 24), second array stores squares of the loop counter, third array stores sums of the corresponding elements.
2. **Implementation and Challenges**

- **Implementation**:
  - Used a `for` loop from 0 to 24 to fill the first two arrays.
  - Created another loop to add the corresponding elements and store results in the third array.
- **Challenges**:
  - Formatting the display as required ("counter; element + element = element").
  - Confirmed no off-by-one errors in indexing.

3. **Reflections**
   - A straightforward task, but valuable for reinforcing array fundamentals.
   - Showed how to structure loop logic cleanly.

Connection to Learning Outcomes: Strengthened my familiarity with array indexing, loops, and displaying results in a clear format.

Week 6 (Nov 19–Nov 23): Problem 5 – Book and Bookshelf Classes

1. **Design Decisions**
   - Extended the `Book` class from Assignment 2 to incorporate additional attributes or methods as needed.
   - Created a `Bookshelf` class that manages an `std::vector<Book>` (or `std::list<Book>`), storing 12 Book objects.
2. **Process and Challenges**
   - **Implementation**:
     - `Bookshelf` has a `main()` that populates 12 books with distinct attributes (title, ISBN, author, etc.).
     - Implemented a sorting routine using a custom comparator that sorts first by title, then by year of publication if titles match.
   - **Challenges**:

    - Ensuring the comparator function or lambda worked properly in C++.
    - Displaying unsorted and sorted lists in a readable format.
3. **Reflections**
    - Learned more about how to implement custom sorting in C++ using the STL's `std::sort` with a comparator.
    - Reinforced OOP design by reusing the `Book` class and incorporating it into a larger class (`Bookshelf`).

Connection to Learning Outcomes: Practiced advanced usage of the standard library (e.g., vectors and sorting), plus data encapsulation in a real-world scenario.

Week 7 (Nov 24–Nov 27): Problem 6 – Custom Set Template Class

1. **Design Decisions**
    - Chose to implement a `Set<T>` template backed by an `std::vector<T>` for internal storage.
    - Decided to store only unique elements (checked before insertion).
    - Created a nested iterator class that mimics standard library set behavior (with `begin()`, `end()`, `operator++`, etc.).
2. **Implementation and Challenges**
    - **Implementation**:
        - On insertion, verified that the element was not already in the vector before pushing it.
        - Added the nested `iterator` class with a pointer or index referencing the internal vector.
    - **Challenges**:
        - Understanding the differences between standard library iterators and custom iterators.
        - Testing correctness against `std::set` by comparing results of insertion and traversal.

3. **Testing**
    - Inserted various integer, string, and custom class elements into both my custom `Set` and `std::set`.
    - Ensured iteration order may differ (since `std::set` is typically ordered), but uniqueness matched across both containers.
4. **Reflections**
    - This was the most challenging part of the assignment, involving templates, iterators, and unique-element logic.
    - Gained a deeper appreciation for how the C++ STL is implemented and how to replicate some of its functionalities.

Connection to Learning Outcomes: Practiced advanced C++ template usage, demonstrating the ability to create generic, reusable data structures.

Final Reflections

Overall, Assignment 3 was an extensive exercise in **file handling, array manipulation, and template programming**. Each problem tackled a unique aspect of C++:

- **File I/O** in Problems 1–3 cemented my ability to handle text file operations, from word counting to reading entire files into arrays.
- **Array operations** in Problems 4–5 underscored the importance of data structure planning and loop logic.
- **Template and STL** usage in Problem 6 introduced me to creating my own data structures and iterators, giving me a behind-the-scenes view of the standard library's design.

Maintaining this journal was tremendously beneficial, as it helped me capture the progress, setbacks, and solutions for each problem. By the end, I had not only working C++ programs but also a clearer perspective on best practices for planning, testing, and implementing various programming tasks.

Key Takeaways:

- Thorough planning and iterative testing minimized confusion and rework.

- Proper documentation (journaling) made it easier to reflect on the growth of my skills.
- OOP and templates in C++ offer powerful abstractions but require attention to detail and consistent testing.

I believe the work in this assignment helped fulfill the course's learning outcomes regarding **file I/O, object-oriented programming, data structures, and template concepts**. I am more confident handling diverse tasks in C++ now, from reading files to building custom containers.

Sources and References

1. DevDocs C++ – for syntax and standard library references
2. Course forums and official C++ documentation for clarifications on file I/O, iterators, and templates

---

End of Assignment 3 Journal