CS 5393 C++ FOR CS: LAB 4

C++ LIBRARY DEVELOPMENT

Overview: In this lab, you will combine the Memory Manager and Profiler systems into a unified, professional-grade library. You'll design and implement a cohesive API that allows developers to leverage both systems effectively, package the solution as a shared library, and demonstrate its effectiveness through comprehensive testing and analysis. While the base requirements specify implementation for a single platform with static linking, additional credit opportunities are available for cross-platform support and dynamic linking capabilities.

Repo Link: https://classroom.github.com/a/KGpPa2XM

Due: December 9th, 2024 @ 11:59pm.

CODE REQUIREMENTS (40%)

1. LIBRARY ARCHITECTURE AND INTERFACE DESIGN (15%)

- Design a unified API that integrates Memory Manager and Profiler functionality:
  - For instance you could implement and document the following design patterns:
    - Factory Pattern (for system instantiation)
    - Singleton Pattern (for global access)
    - Observer Pattern (for system events)
    - Strategy Pattern (for varying memory allocation strategies)
  - Create comprehensive UML diagrams showing:
    - Class hierarchies
    - Design pattern implementations
    - System interactions
    - Component relationships
- Provide configuration options for:
  - Memory pool size and alignment settings
  - Profiling granularity and output formats
  - Debug/Release mode behaviors

2. STATIC LIBRARY IMPLEMENTATION (15%)

- Create a static library that encapsulates all functionality:
  - Implement proper symbol visibility and export controls
  - Handle platform-specific compilation requirements
- Required Features:
  - Memory allocation and tracking
  - Performance profiling and reporting

- Error handling and reporting
- Configuration management

3. LIBRARY DISTRIBUTION AND INTEGRATION (10%)

- Provide build system support:
  - make configuration for your target platform
  - Clear build instructions
  - Dependency management
- Create integration documentation:
  - Header files and library distribution
  - Integration examples

DOCUMENTATION AND ANALYSIS REQUIREMENTS (60%)

1. TECHNICAL DESIGN DOCUMENT (15%)

- Detailed architecture overview:
  - System component descriptions
  - Design pattern implementations and rationale
    - Explicit explanation of each pattern used
    - Justification for pattern selection
  - Interface design decisions
- System Architecture Diagrams:
  - System diagram showing system structure
  - Class diagrams showing pattern implementations
- Implementation details:
  - Memory management strategy
  - Profiling methodology
  - Performance optimizations

2. USER GUIDE AND API DOCUMENTATION (15%)

- Installation and setup instructions:
  - Building from source
  - Integration into existing projects
  - Platform requirements
- API documentation:
  - Interface descriptions
  - Usage examples

3. PERFORMANCE ANALYSIS AND DEMONSTRATION (10%)

- Comprehensive test suite demonstrating:

    - Memory allocation patterns and performance
    - Profiling accuracy and overhead
    - Real-world usage scenarios
- Performance analysis:
    - Memory usage patterns
    - Timing and profiling results
    - Optimization opportunities

4. CASE STUDY AND OPTIMIZATION REPORT (20%)

The case study component of this lab serves as a critical bridge between theoretical understanding and practical application of C++ performance principles. Students will create a demonstration application that not only validates their library's functionality but also showcases their deep understanding of C++ memory mechanics and performance optimization. Through implementing and then optimizing a complex algorithm or data structure, students will explore how different memory access patterns and data structure designs fundamentally impact program performance. This exploration should be backed by concrete evidence from their profiler and memory manager, using the collected metrics to validate their optimization decisions. The analysis should demonstrate a thorough understanding of how memory layout, access patterns, and data structure design choices directly influence CPU cache utilization and overall program efficiency. By connecting observable performance metrics to specific implementation decisions, students will demonstrate their grasp of both the practical and theoretical aspects of C++ performance optimization.

- Detailed analysis of a specific optimization case:
    - Initial performance characteristics
    - Identified bottlenecks
    - Implemented optimizations
    - Measured improvements
    - Analysis backed by profiler data
    - Memory usage impact

EXTRA CREDIT OPPORTUNITIES (UP TO 10% ADDITIONAL)

1. CROSS-PLATFORM SUPPORT (5%)

- Additional platform support beyond primary platform:
    - Windows (MSVC, MinGW)
    - Linux (gcc, clang)
    - macOS (clang)
- Platform-specific optimizations
- Build system support for all platforms

2. DYNAMIC LINKING SUPPORT (5%)

- Implementation of dynamically linked library version:

- Proper dynamic library creation
- Dynamic loading example application
- Documentation for both static and dynamic linking

SUBMISSION REQUIREMENTS

1. All source code must be submitted via the provided GitHub repository
2. Complete technical report in PDF format
3. Test application source code and results
4. Any additional tools or scripts used for visualization
5. You can update the compile command in the make file to ensure it will properly compile your code. You can add as many additional make commands as desired, be sure any additional make commands are documented in the report above.

If your project fails the compile action upon final push to git repo, it will receive a 0. Be sure to resolve any issues with the compile action prior to deadline.