SMU | LYLE SCHOOL OF ENGINEERING

CS 5393 C++ FOR CS: FINAL

C++ INTEGRATED DEVELOPMENT TOOL SUITE

Overview: In this lab, you will create an integrated development tools library that combines your memory manager and profiler implementations with runtime debugging capabilities. You will implement a singleton debugger interface that enables runtime visualization and analysis of memory and performance data through LLDB's expression interface. The system will be packaged as a professional-grade library with comprehensive documentation and visualization tools. You may use any existing code, samples, demos and documentation you have already created from previous labs in the course.

Repo Link: https://classroom.github.com/a/v5NxGpDp

Due: December 15th, 2024 @ 5:00pm.

A node based LLDB REPL client/websocket server is available to enable loading and deubbing of c++ in realtime. The repo which contains documentation, and samples can be found at: https://github.com/HuMIn-Game-Lab/LLDBDebugger You are allowed to utilize/modify this code in any way needed. You do not need to connect your c++ applications directly to LLDB/websocket server. Deubgging via LLDB is done post development, after compile time. It loads the executable and provides an interface in realtime to engage with the running application. You must only provide an interface within your code that will enable the ability to interact with the code during a runtime debug session.

CODE REQUIREMENTS (50%)

1. LIBRARY IMPLAMENATION (15%)

- Create a shared library (.so/.dll) containing your Memory Manager, Profiler, and Debugger systems.
- Include proper header files for public interfaces.
- Ensure proper design of library interface to minimize dependencies.

2. DEBUGGER INTERFACE (15%)

- Ensure the interface can be accessed through LLDB's expression evaluator during debug sessions
- Implement proper memory and resource management for singleton pattern
- Implement a singleton Debugger class with the following minimum interface:

```
class Debugger {
public:
    static Debugger* GetInstance();
    virtual std::string MemorySnapshot() = 0;  // Returns memory state visualization data
    virtual std::string PerformanceSnapshot() = 0;  // Returns performance metrics
    // Additional helper methods as needed
private:
    static Debugger* s_instance;
    // Implementation details
};
```

3. MEMORY VISUALIZATION (10%)

- Implement memory state capture and formatting
- Include allocation tracking with source information
- Show memory block states, and sizes
- Provide memory usage statistics
- Return data in a format suitable for visualization (JSON recommended)

4. PERFORMANCE VISUALIZATION (10%)

- Implement performance data capture and formatting
- Include function timing hierarchies
- Show call counts and execution statistics
- Return data in a format suitable for visualization (JSON recommended)

DOCUMENTATION AND ANALYSIS REQUIREMENTS (40%)

1. TECHNICAL DESIGN DOCUMENT (10%)

- Complete API reference for public interfaces
- Library integration guide for both static (and/or dynamic linking if provided)
- Build and installation instructions for supported platforms

2. USER GUIDE (15%)

- Step-by-step debugging session examples
- Command reference for LLDB integration
- Memory analysis interpretation guide
- Performance data analysis methodology
- Visualization tool setup and usage

3. DEMONSTRATION (15%)

- Sample Application
    - Create a demonstration program that exercises all library features
    - Include examples of memory-intensive operations

    - Demonstrate performance profiling scenarios
    - Show debugging session workflows

EXTRA CREDIT OPPORTUNITIES (UP TO 10% ADDITIONAL)

1. ENHANCED VISUALIZATION (UP TO 10%)

- List of possible examples:
    - Interactive memory map visualization
    - Real-time performance data graphing
    - Custom visualization themes
    - Additional visualization formats

2. ADDITIONAL FEATURES (UP TO 10%)

- List of possible examples:
    - Custom LLDB command integration
    - Additional analysis tools
    - Performance comparison tools
    - Memory leak detection utilities

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.