**Flash Memory Logger and Mock Framework task description**

To avoid “it worked on my computer”, use this online environment to run the main.cpp code:

<https://www.programiz.com/cpp-programming/online-compiler/>

## **How to use flash memory logger**

MockFlashMemory **flashMemory**;

FlashMemoryLogger **logger**(flashMemory);

std::vector<uint8\_t> logData = {(0, 0, 1, 2, 3, 4, 5, 6}; //2 first bytes are ID, 6 last ones are data

logger.**writeLog**(logData); //write vector to flash

std::vector<uint8\_t> logData = logger.**readLog**(i); //access specific log entry

std::cout << "Log ID " << i << " data: ";

for (const auto& byte : logData) {

std::cout << std::hex << static\_cast<int>(byte) << " ";

}

std::cout << std::endl;

flashMemory.**erasePage**(0); //deleting the first page of memory

logger.showMemoryUtilization(); // prints nr of logs used and % space used

### **Overview**

This code demonstrates a simple implementation of a mock flash memory framework and a flash memory logger module. The MockFlashMemory class provides functions to write, read, and erase pages in the mock flash memory. The FlashMemoryLogger class uses the MockFlashMemory class to write logs into flash memory and read logs from flash memory.

### **MockFlashMemory Class**

The **MockFlashMemory** class represents a mock flash memory framework. It has the following member functions:

1. Constructor **MockFlashMemory():** Initializes the mock flash memory by resizing the memory vector to accommodate the specified number of pages. Each page is initially filled with 0xFF.
2. writePage **void writePage(int page, const std::vector<uint8\_t>& data):** Writes data to a specific page in the flash memory. It performs the following steps:
   * Checks if the provided page number is valid.
   * Checks if the data size is equal to the page size.
   * Copies each byte from the data vector to the corresponding position in the flash memory.
3. readPage **std::vector<uint8\_t> readPage(int page):** Reads the content of a specific page in the flash memory. It performs the following steps:
   * Checks if the provided page number is valid.
   * Retrieves the data from the flash memory based on the specified page.
   * Returns a vector containing the data.
4. erasePage **void erasePage(int page):** Erases the content of a specific page in the flash memory. It performs the following steps:
   * Checks if the provided page number is valid.
   * Fills the specified page with 0xFF.
5. printFirstPage **void printFirstPage():** Prints the content of the first page in the flash memory. It iterates over the bytes in the first page and prints their hexadecimal values.

### **FlashMemoryLogger Class**

The **FlashMemoryLogger** class represents a flash memory logger module. It interacts with the **MockFlashMemory** class to write and read logs from the flash memory. It has the following member functions:

1. Constructor **FlashMemoryLogger(MockFlashMemory& flashMemory):** Initializes the flash memory logger with a reference to a MockFlashMemory object. It also initializes the log ID and memory rewrite flag.
2. writeLog **void writeLog(const std::vector<uint8\_t>& logData):** Writes a log to the flash memory. It performs the following steps:
   * Checks if the log data size is equal to the log size defined in the class.
   * Calculates the page and log offset based on the current log ID.
   * Reads the current page data from the flash memory.
   * Copies the log data to the appropriate position in the page data.
   * Updates the log ID in the page data.
   * Writes the updated page data back to the flash memory.
   * Updates the log ID and memory rewrite flag.
3. readLog **std::vector<uint8\_t> readLog(uint16\_t logId)**: Reads a log from the flash memory based on its log ID. It performs the following steps:
   * Calculates the page and log offset based on the provided log ID.
   * Reads the corresponding page data from the flash memory.
   * Retrieves the log data from the page data and returns it as a vector.
4. showMemoryUtilization **void showMemoryUtilization()**: Prints the current memory utilization in terms of logs stored. It calculates the utilization percentage based on the log ID and total available logs.

## **Nice to have, but outside of the scope of the project**

There are many constraints of flash memory and the embedded system environment:

* **Durability** - After many writing cycles to one spot, the data gets corrupted. To avoid that, data needs to be written evenly throughout the memory. Also avoiding overwriting the same value to the same spot extends the life of flash memory, but this is outside of the scope.
* **Speed** - For STM32 and Atmega microcontrollers have delays for reading and writing. Often underlying Flash drivers use blocking delay that can cause problems in time critical application such as emergency braking. This falls outside of the task of the scope.
* **Power failure** during the write cycle can cause data corruption. Control sums and other detection methods are outside of the scope of this task.