Your submission archive (only .zip or .tar.gz archives accepted) must include:

1. A **makefile** which behaves as the one provided or code that compiles with a makefile like the one provided.
   There is a 100% penalty for any deviation from this format.
2. Correct C++ source file–**mmap_fstream.cc**–for a library implementing the described class object.
3. Correct C++ header file–**mmap_fstream.cc**–for a library declaring the described class object.
4. A README.md file describing your project. It should list and describe:
   (a) Where you memory map the file.
   (b) Where you set the size of the file in the memory map (truncate/ftruncate), where applicable.
   (c) Where you read from the file contents using a memory map.
   (d) Where you write to the file using a memory map.
   (e) Where you save the file to disk using a memory map.

Note that all files above must be your original files. Submissions will be checked for similarity with all other submissions from both sections, but with consideration of what is provided.

Overview

This project explores file I/O using a memory map.

Your task is to create a class library which implements a replacement for a subset of the `std::fstream` class from the C++ standard template library.

Interface/Declarations

You must provide the following public interface for the library in `mmap_fstream.h`. You may extend the private/protected potions of your code in any way you see fit.

`mmap::fstream`

- + `fstream(void)`
- + `fstream(const std::string&)`
- + `fstream(const std::string&, std::ios_base::openmode)`
- + `void open(const std::string&)`
- + `void open(const std::string&, std::ios_base::openmode)`
- + `void close(void)`
- + `bool is_open(void) const`
- + `std::size_t size()(void) const`
- + `char get()`
- + `put(char)`

As mentioned in the comments, you may use overloading to reduce the number of methods.

Implementation

In the private interface, you must provide implementations for the methods described in the interface file `mmap_fstream.cc`.

In addition, you must document the parts of your code where the following take place. Look for the key phrase for each section. If your code is missing one or more, you will not earn points for any tests of that section.

1. Where you memory map the file. On the line above, include the comment:
   `// MEMORY MAP OPEN FILE`
   Without this line, you will receive zero test points.
2. Where you set the size of the file in the memory map (truncate/ftruncate), where applicable. On the line above, include the comment:
   `// ALLOCATE FILE MEMORY`
   Without this line, you will receive zero points for the last two tests.
3. Where you read from the file contents using a memory map. On the line above, include the comment:
   `// READ FROM FILE`
   Without this line, you will receive zero points for the `get` test.
4. Where you write to the file using a memory map. On the line above, include the comment:
   `// WRITE TO FILE`
   Without this line, you will receive zero points for the two `put` tests.
5. Where you save the file to disk using a memory map. On the line above, include the comment:
   `// SAVE TO DISK`
   Without this line, you will receive zero points for the two `put` tests

Any attempt to use interfaces other than mmap, munmap, f/truncate, msync, or character pointer assignment to implement this project will result in an Academic Integrity violation and a -100% score for your project.

Cont.

Notes

References

- I have included a file providing an example of file-backed memory mapping. It is not all you require, but it does provide much.
- It can be built with

  ```
  g++ -std=c++17 -Wall -pedantic -o grow-mmap-file grow_mmap_file.cc
  ```

- It can be invoked by

  ```
  ./grow-mmap-file test.txt
  ```

Grading

Grading is based on the performance of your library. There are no points for "coding effort" when code does not compile and run.

You are provided with some basic tests. You may want to extend these if you find yourself programming toward passing only these tests.

The tests examine your code in the following ways:

0. Attempt to open files and determining their sizes
1. Attempt to read from existing files
2. Attempt to write to new files
3. Attempt to append to existing files

Note that previous tests must be passed for points from subsequent tests, i.e. your library must open files before files can be read and read files before they can be read.

The portions of the project are weighted as follows:

1. **style/README.md**: 2.0
2. **compilation against public interface**: 1.0
3. **Test 0**: 1.0
4. **Test 1**: 1.0
5. **Test 2**: 2.0
6. **Test 3**: 3.0

The End.