CMSC 341 - Project 0: Using GL, Memory Management, Testing, and Efficiency - Spring 2021

Due: Friday Feb 12, before 9:00 pm

There are no late submission folders for Project 0. You must turn it in on-time to receive credit.

Addenda

- 01/30/21 - In the constructor Puzzle::Puzzle(int size), if the user passed a value less than 0 to the constructor, the m_size variable will be set to zero.

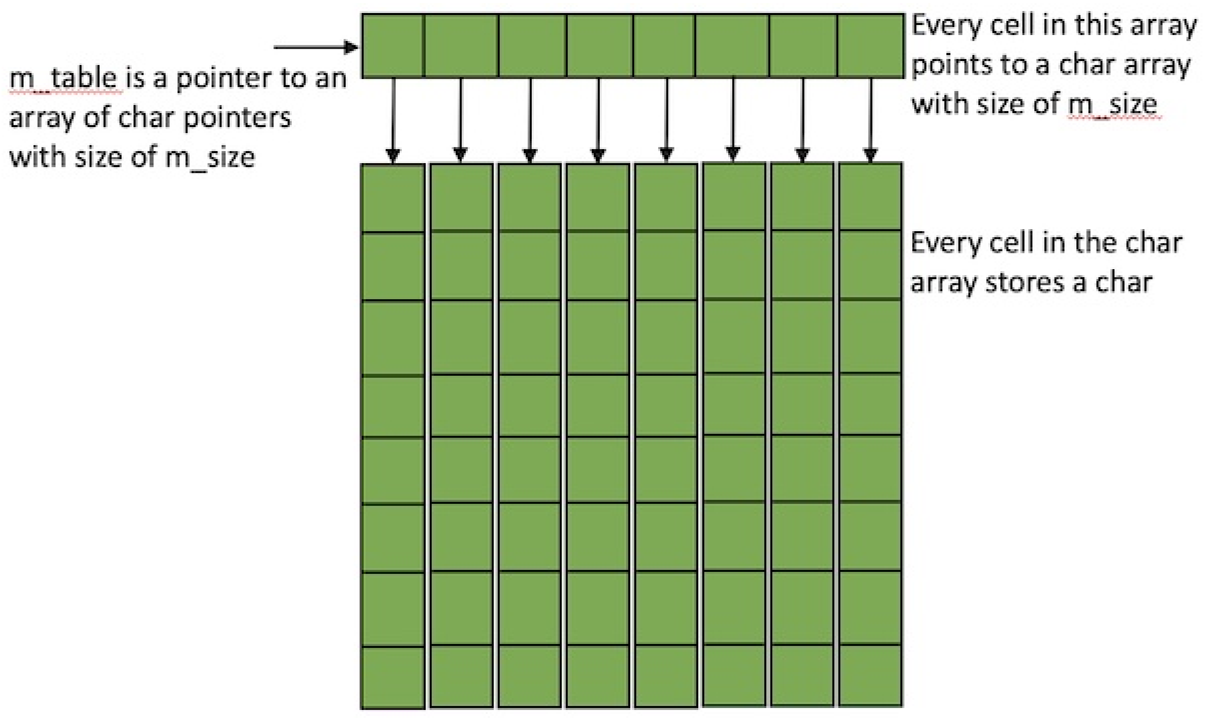
Objectives

- Review the procedures to access the GL servers and to compile programs on GL.
- Review C++ memory management (allocating and deallocating memory dynamically), including copy constructors, destructors, and assignment operators.
- Use Valgrind to check for memory leaks.
- Learn how to test a project.
- Learn how to measure the efficiency of algorithms.
- To ensure that you are able to submit project files on GL.

Introduction

In this project, you will complete a C++ class by writing a copy constructor, destructor, and assginment operator. Furthermore, you will write a tester class and a test program and use Valgrind to check that your program is free of memory leaks. Finally, you will submit your project files on GL. If you have submitted programs on GL using shared directories (instead of the `submit` command), then the submission steps should be familiar.

The `Puzzle` class in this project contains a 2-dimensional array which stores the alphabetic characters like crossword puzzles. For the simplicity the array is populated with random characters. There are no meaningful words in the puzzles this class generates. The sample output in the file driver.txt shows an example of a table. The following figure visualizes the 2-d structure.



m_table is a pointer to an array of char pointers with size of m_size
Every cell in this array points to a char array with size of m_size
Every cell in the char array stores a char

Assignment

Step 1: Create your working directory

Create a directory in your GL account to contain your project files. For example, `cs341/proj0`.

Step 2: Copy the project files

You can right-click on the file linkes on this page and save-as in your computer. Then, transfer the files to your GL account using SFTP.

For this project, you are provided with the skeleton .h and .cpp files, a sample driver, and the output of driver program:

- puzzle.h - The interface for the Puzzle class.
- puzzle.cpp - The skeleton for the Puzzle class implementation.
- driver.cpp - The sample driver program.
- driver.txt - The output of sample driver program.

Step 3: Complete the `Puzzle` class

Complete the class by writing the alternative constructor, copy constructor, assignment operator, and the private function members makeMem() and clear() in puzzle.cpp. The class interface is provided in puzzle.h.

void Puzzle::makeMem()	This function allocates memory to the 2d array m_table. If the size of array is 0, no memory will be allocated. The size of array is specified by the variable m_size. The m_size variable specifies both the number of rows and the number of columns for m_table.
void Puzzle::clear()	This function deallocates the memory in the object and sets the size to zero. if the size of m_table is already zero, there is no need to deallocate since there is no memory allocated.
Puzzle::Puzzle(int size)	This is the alternative constructor. The input variable size specifies the number of rows and number of columns for m_table. This constructor calls the private functions makeMem() and fill() to allocate memory in the object and to initialize all cells of m_table. If the size argument is less than 0, the m_size will be set to zero.
Puzzle::Puzzle(const Puzzle& rhs)	This is the copy constructor. It creates a deep copy of rhs in the current object. The copy constructor can call the private function makeMem() to create memory.
const Puzzle& Puzzle::operator=(const Puzzle& rhs)	This is the assignment operator. It creates a deep copy of rhs. **Reminder:** an assignment operator needs protection against self-assignment.

Step 4: Test your code

You must write and submit a test program along with the implementation of the Puzzle class. To test your Puzzle class, you implement your test functions in the `Tester` class. The `Tester` class resides in your test file. You name your test file `mytest.cpp`. It is strongly recommended that you read the testing guidelines before writing test cases. A sample test program including `Tester` class, and sample test functions are provided in driver.cpp. You add your Tester class, test functions and test cases to `mytest.cpp`. The following two test functions are provided to you in driver.cpp in the Tester class. We recommend that you study the implementation of these functions.

bool Tester::testCopyConstructor(const Puzzle& puzzle)	This function tests the correctness of copy constructor. If the copy is performed correctly, the function returns true, otherwise it returns false. There are multiple cases to check for the correctness of copy operations. For example, a copy constructor should make a deep copy, then the test function should check for that. Or the corresponding values of the two copies should be equal, then the test function needs to check for it.
void Tester::measureInsertionTime(int numTrials, int N)	An important matter in data structures is the efficiency or running time of algorithms. This test function is an example for analyzing the running time of the insertion operation in a Puzzle object (the call to the alternative constructor). The running time grows/scales with the amount of data. We expect that the running time grows linearly corresponding to the growth of puzzle size. The puzzle size is specified by (m_size x m_size). For example, if m_size is 1000, then the puzzle size is 1000000. If m_size is 2000, the puzzle size is 4000000. When we move from 1000000 data items to 4000000 data items, the data size grows by a factor of 4.

	Therefore, if we increase the puzzle size by a factor of 4, we expect that the running time increases by a factor of 4.

Your test program must test the correctness of assignment operator, and proves that the running time of your copy constructor grows linearly;

- Check that a copy is made. The new puzzle should contain exactly the same data as the source puzzle.
- Check that the copy is *deep*.
- Check *edge cases*. For example, do they work correctly if the source puzzle is empty?
- For the assignment operator, check that you have guarded against self-assignment.
- Write a test function that measures the running time of the copy constructor. Run the copy constructor with at least 5 trials, and compare the measured times.

Step 5: Check for memory leaks

Run your test programs using Valgrind. For example, assuming you have compiled `mytest.cpp`, producting the executable `mytest.out`, run the command

```
valgrind mytest.out
```

If there are no memory leaks, the end of the output should be similar to the following:

```
==8613==
    ==8613== HEAP SUMMARY:
    ==8613==     in use at exit: 0 bytes in 0 blocks
    ==8613==   total heap usage: 14 allocs, 14 frees, 73,888 bytes allocated
    ==8613==
    ==8613== All heap blocks were freed -- no leaks are possible
    ==8613==
    ==8613== For lists of detected and suppressed errors, rerun with: -s
    ==8613== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The important parts are “in use at exit: 0 bytes” and “no leaks are possible.” The last line is also important as memory errors can lead to leaks.

Step 6: Link your shared directory

Follow the instructions on the Project Submission page to make a symbolic link to the shared directory in your home directory.

Step 7: Submit your files

See the “What to Submit” section, below.

Implementation Notes

Each project has a section on implementation notes. These point out some issues that you might encounter while developing your code. You should look through the Implementation Notes before you start coding.

For Project 0, there are only a few notes:

- The class declaration (`Puzzle`) and provided function implementations in `puzzle.cpp` may not be modified in any way. No additional libraries may be used, but additional `using` statements are permitted.
- The locations for the function implementations are clearly marked in `puzzle.cpp`. They must be written at the specified locations; in particular, they must not be written “in-line.”
- Private helper functions may be added, but must be declared in the private section of the `Puzzle` class. There is comment indicating where private helper fuction declarations should be written. **Note:** in this project, the private functions which are already declared, provide you the required functionality.
- You should read through the coding standards for this class.

What to Submit

You must submit the following files to the `proj0` submit directory:

- `puzzle.h`
- `puzzle.cpp`

- `mytest.cpp` (**Note:** This file contains the declaration and implementation of your Tester class as well as all your test cases.)

If you followed the instructions in the Project Submission page to set up your directories, you can submit your code using the following command:

```
cp puzzle.h puzzle.cpp mytest.cpp ~/cs341proj/proj0/
```

Grading Rubric

The following presents a course rubric. It shows how a submitted project might lose points.

- Conforming to coding standards make about 10% of the grade.
- Correctness and completeness of your test cases (mytest.cpp) make about 15% of the grade.
- Passing tests make about 30% of the grade.

If the submitted project is in a state that receives the deduction for all above items, it will be graded for efforts. The grade will depend on the required efforts to complete such a work.