# CSCI-1200 Data Structures — Spring 2014
Homework 3 — Resizable Tables

In this assignment you will build a custom data structure named `Table`. Building this data structure will give you practice with pointers, dynamic array allocation and deallocation, and writing templated classes. The implementation of this data structure will involve writing one new class. You are not allowed to use any of the STL container classes in your implementation or use any additional classes or structs. Please read the entire handout before beginning your implementation. Also, be sure to review the Lecture 7 notes and our implementation of the `Vec` class mimicking STL's `vector`.

The Data Structure

The resizable `Table` class is a simple two-dimensional collection of data values of template type `T`. Like ordinary C/C++ arrays, you can `get` and `set` individual entries in the 2D grid of data. But additionally, like the STL vector class, you can dynamically grow or shrink the number of values stored in the table using the `push_back_row`, `push_back_column`, `pop_back_row`, and `pop_back_column` functions. Below is a diagram of the data structure you will implement. In this example `T` is type `char`.
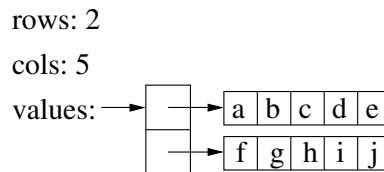
The `Table` class has 3 member variables: `rows` and `cols`, unsigned integers representing the current size of the Table, and `values`, an array that stores pointers to each row of the table. Each row of the table is an array with exactly as many entries as there are columns in the Table. In the above example, a call to `get(1,2)` returns 'h'. The `set` function takes in 3 arguments: the row and column indices and the new value for that entry in the table. Attempting to `get` or `set` a value outside the bounds of the table is an error. Your program should check for bad indices, print a warning message to `std::cerr` and call `exit(1)`. The user of the `Table` class can access the table's current size by calling `numRows` and `numColumns`.

Modifying the Data Structure

The overall dimensions of the `Table` data structure can be modified with functions similar in syntax to the STL vector `push_back` and `pop_back` functions. The `rows` and `cols` variables are updated as needed, all affected arrays are reallocated to be *exactly* as big as needed, the data is copied into the new arrays, and all old replaced arrays are deleted. The `push_back_row` and `push_back_column` functions take in a single argument, an STL vector, that contains the data values for the new row or column in the table. For example, if `new_row` is a vector of 5 `char`s: A, B, C, D, and E, then a call to `push_back_row(new_row)` will result in the new data structure diagram below left. Note that the top level array of pointers is reallocated, copied, & deleted, but the existing row data arrays can be reused.
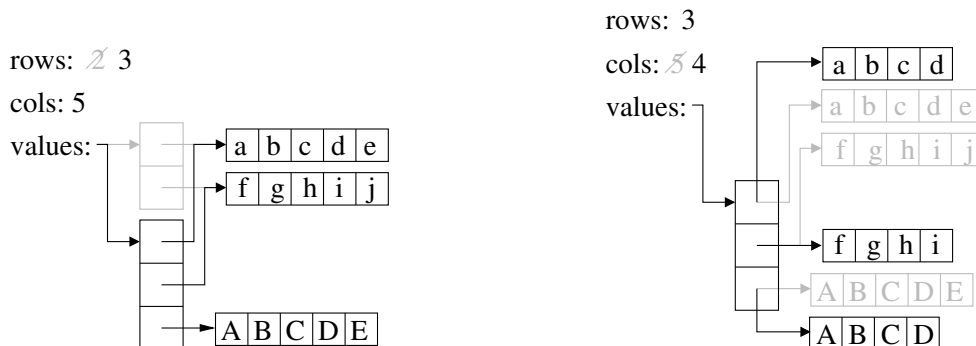
If the user next calls `pop_back_column()` on this example each row is shortened by one entry by allocating new smaller arrays, copying data, and cleaning up the unused memory. This is shown in the figure above right. The `push_back_column` and `pop_back_row` member functions work similarly. It is an error to attempt to `push_back` a row or column using a vector of data that is the wrong size (too long or too short). It is also an error to attempt to `pop_back` a row if `rows==0` or to `pop_back` a column if `cols==0`. Your program should check for these cases and print a message to `std::cerr` and call `exit(1)`.

Testing, Debugging, and Printing

We provide a main function that will exercise your data structure; however, the tests are not thorough. Some of these tests are initially commented out. We recommend you get your class working on the basic tests, and then uncomment the additional tests as you implement and debug the key functionality of the `Table` class. Study the provided test cases to understand what code triggers calls to your `Table` copy constructor, assignment operator, and destructor and verify that these functions are working correctly.

It is your responsibility to add additional test cases, including examples where the template class type `T` is something other than `char`. You must also implement a simple `print` function to help as you debug your class. Include examples of the use of this function in your new test cases. Your function should work for `Table`s containing `char`, `int`, and *reasonably short* `string`s. The `print` function does not need to work for more complex types. Please use the example output as a guide (we will grade this output by hand).

Discuss your testing and debugging strategy in your README.txt file. What tools did you use (gdb/lldb/Visual Studio debugger, Valgrind/Dr. Memory, std::cout & print, etc.)? How did you thoroughly test the "corner cases" of the Table class design and implementation?

Performance Analysis

The data structure for this assignment (intentionally) involves a lot of memory allocation & deallocation. Certainly it is possible to revise this design for improved performance and efficiency or adapt the data structure to specific applications. For this assignment, please implement the data structure *exactly* as described.

In your `README.txt` file include the order notation for each of the `Table` member functions described above: `get`, `set`, `numRows`, `numColumns`, `push_back_row`, `push_back_column`, `pop_back_row`, `pop_back_column`, and `print`. You should assume that calling `new []` or `delete []` on an array will take time proportional to the number of elements in the array. In your answers use n = the number of rows and m = the number of columns.

Looking for Memory Leaks

To help verify that your data structure does not contain any memory leaks and that your destructor is correctly deleting everything, we include a batch test function that repeatedly allocates a `Table`, performs many operations, and then deallocates the data structure. To run the batch test case, specify 2 command line arguments, a file name (`small.txt`, `medium.txt`, or `large.txt`) and the number of times to process that file. If you don't have any bugs or memory leaks, this code can be repeated indefinitely with no problems.

On Unix/Linux/OSX, open another terminal/shell and run the `top` command. While your program is running, watch the value of "RES" or "RPRVT" (resident memory) for your program. If your program is leaking memory, that number will continuously increase and your program will eventually crash. Alternately, on Windows, open the Task Manager (Ctrl-Shift-Esc). Select "View" → "Select Columns" and check the box next to "Memory Usage". View the "Processes" tab. Now when your program is running, watch the value of "Mem Usage" for your program (it may help to sort the programs alphabetically by clicking on the "Image Name" column). If your program is leaking memory, that number will continuously increase.

Memory Debuggers

We also recommend using a memory debugging tool to find errors. Information on installation and use of the memory debuggers "Valgrind" (available for Unix/Linux/OSX) and "Dr. Memory" (available for Linux/Windows) is presented on the course webpage:

http://www.cs.rpi.edu/academics/courses/spring14/ds/memory_debugging.php

The homework submission server is configured to run your code with Valgrind to search for memory problems. Your program must be memory error free to receive full credit.

Extra Credit

For extra credit, implement versions of the `push_back` functions that accept a `Table` as the argument, allowing multiple rows (or columns) to be added. Also, for extra credit, you may rewrite your `Table` implementation to use pointers and pointer arithmetic for access to the internal data arrays *instead of* the subscripting operator, `[]`. *Note:* We recommend that you fully debug your program using the more intuitive `[]` accessor for arrays before switching to pointer arithmetic!

Submission

Do all of your work in a new folder named `hw3` inside of your Data Structures homeworks directory. Use good coding style when you design and implement your program. Be sure to make up new test cases and don't forget to comment your code! Please use the provided template `README.txt` file for any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.** When you are finished please zip up your files exactly as instructed for the previous assignments and submit it through the course webpage.