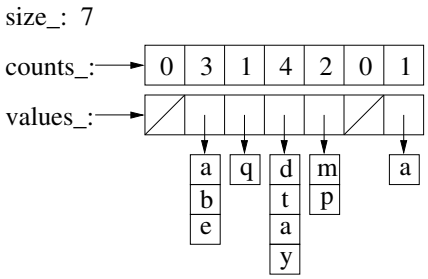
CSCI-1200 Data Structures — Fall 2025
Homework 3 — Undo Array

In this assignment you will build a custom data structure named `UndoArray`. 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 notes for Lecture 7 & 8 and the Lab 4 material with our implementation of the `Vec` class mimicing STL's `vector`.

The Data Structure

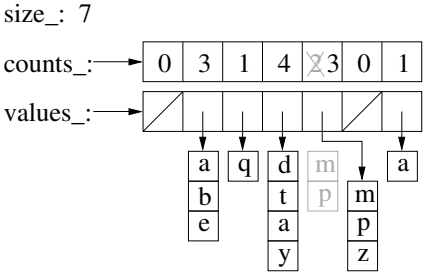
The `UndoArray` works like an ordinary fixed-size array that stores n values of template type `T`. Like ordinary C/C++ arrays, you can `get` and `set` individual entries in the array. What is unusual is that you can sequentially `undo` the calls to `set`. To make this work, we'll need to store the history of all values that were assigned to each index. Below is a diagram of the data structure you will implement. In this example `T` is type `char`:

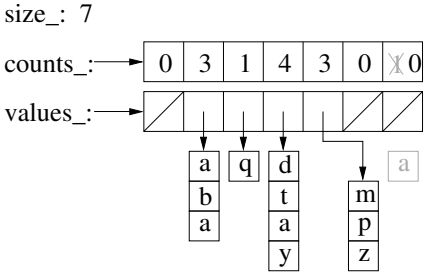


The `UndoArray` class has 3 member variables: `size_`, an unsigned integer representing the size of the `UndoArray`; `counts_`, an array that stores the number of times each position in the `UndoArray` has been set; and `values_`, an array of arrays that store the history of values for each position in the `UndoArray`. The current value of a position in the `UndoArray` is the last entry in the history. In the above example, a call to `get(1)` returns 'e'. The length of each history array is equal to the number of times that value has been *set* (minus the number of times it has been *undone*). If a position in the `UndoArray` is uninitialized, the corresponding position in the values array stores a NULL pointer (indicated with a slash). Attempting to read an uninitialized value in the `UndoArray` is an error. The boolean `initialized` function can be used to verify that the value is initialized. See the provided testing code for examples of the usage.

Modifying the Data Structure

When a position is *set*, the counts array is appropriately incremented, a new history array is allocated that is one longer than the previous history array, all values in the history are copied with the new value written at the end, the old history is deleted, and the corresponding values pointer is changed. For example, a call to `set(4,'z')` will result in the new data structure diagram below left.





When a position is *undone*, the counts array is decremented, a new history array is allocated that is one shorter than the previous history array, the history values are copied (except for the most recent value), the old history is deleted, and the corresponding values pointer is changed. It is an error to undo an uninitialized value. If the history contains only one value, that position becomes uninitialized; for example a call to `undo(6)` results in the data structure diagram above right.

Testing, Debugging, and Printing

We provide a `main.cpp` file with a small set of tests of your data structure. 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 `UndoArray` class. Study the provided test cases to understand what code triggers calls to your `UndoArray` 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 `UndoArray`s containing `char`, `int`, and *reasonably short* `string`s. The `print` function does not need to work for more complex types. Please closely follow the overall layout and contents of the sample print output. However, this output will not be autograded (the TAs will grade this output manually).

When your data structure encounters an invalid request, for example, getting the value of a slot in the array that is uninitialized, setting a slot in the array that does not exist, etc., your program should print a descriptive message to `std::cerr` and exit. We will be compiling the submitted `UndoArray` header file multiple times with different test cases to fully test your implementation.

Memory Diagramming

You should sketch more diagrams of specific examples of this data structure as you develop, test, and debug your implementation. When you show your work on this homework to a TA or mentor to ask for help in lab or in office hours you must be prepared to show a detailed diagram that corresponds to the specific code you are asking about. You will also submit an image or PDF (named diagram.png or diagram.jpg or diagram.pdf) of a neatly-drawn "before" and "after" diagram from one of the interesting steps in your `StudentTests` function. The diagram can be digitally drawn or it can be drawn on paper and photographed or scanned. Be sure to document what tools you used to create the diagram in your README.txt file.

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. This homework was designed to give you lots of practice with pointers and memory management – *not* to optimize performance. For this assignment, please implement the data structure *exactly* as described in the handout and illustrated in the diagrams.

In your `README.txt` file include the Big 'O' Notation for each of the `UndoArray` member functions described above: `size`, `set`, `initialized`, `get`, `undo`, and `print` *and* don't forget the constructors, destructor, and assignment operator. 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 the variables n = the size of the array and h = the length of the longest history for one space of the array.

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 an `UndoArray`, 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.

```
./undo_array_test.exe small.txt 100
```

On Unix/Linux/OSX, open another 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 if you leave it running for hours or days. 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. Go ahead, comment out some of your `delete` statements to confirm you can see this behavior.

Memory Debuggers

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

http://www.cs.rpi.edu/academics/courses/fall25/csci1200/memory_debugging.php

See also the notes from Lecture 6. Submitty will run your code with Dr. Memory to search for memory problems. Your program must be memory error free *and* memory leak free to receive full credit.

Extra Credit

For extra credit, implement `push_back` and `pop_back` functionality for your `UndoArray` implementation. These functions will change the size of the values and counts arrays. Assume that these functions are rarely used and the array must be increased or decreased by exactly one slot. Write extra test cases to ensure this functionality is correct and memory leak free. Include Big 'O' Notation analysis of these additional functions in your `README.txt` file.

Submission

Be sure to write your own new test cases and don't forget to comment your code! Use the provided template `README.txt` file for notes you want the grader to read. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" document. List the names of anyone you talked to about the problem or debugging and all references you consulted in preparing your solution.