CMPSC 122.1
February 15, 2018

Homework 2

Due: March 5, 2018

Overview

A vector is a container that can be treated like a normal array (created without using the *new* keyword) but also allows for dynamic resizing if the vector runs out of capacity to store new data. How is this possible? A vector is just an abstraction that performs dynamic memory allocation behind the scenes, but is presented in a way that is familiar to using an array, as well as providing additional functionality.

When instantiated, a vector will create a dynamically allocated array of the requested type. The size of the array is typically passed by the user, or handled with default values. Vectors allow for the use of array notation with braces *[]* as well as with an *at* function to access array elements. You can also use the *front* and *back* functions to retrieve the appropriate elements.

The *push_back* and *insert* functions should be used to put new information inside the vector, and incrementally increase the size of the vector. *at* and [] should only be used to reassign the value of an element that has been pushed into the function with *push_back*.

To allow for reassignment as well as access, the above functions return references to the requested element, they do not simply return the value at that index. There are other functions that vector provide that will be discussed in the sections below.

Program 1. (80 points) Integer Vector

Write a program (called *integer_vector.cpp*) that does the following:

1. Implement a class called *BasicVector* that:
    - Contains private member fields for:
        - *data* : integer pointer that is used to point to a dynamically allocated 1D integer array
        - *vector_size* : the current number of items in *data*
        - *vector_capacity* : the maximum number of items that can be contained in *data* before it needs to be resized
    - Contains the private member function:
        - *resize* : No return type (void), no parameters. Dynamically creates a new 1D array with twice the capacity of the existing array. Copy all of the existing elements over into the new array, update *vector_capacity*, use a temporary variable store the current address of *data* and delete [] to free the old array. Point *data* to the new array.
    - Contains public member functions:
        - *at* : returns an int reference to the array element specified. Has conditional logic that prevents using an index larger than the *vector_size* of the BasicVector object by calling *exit 1* to terminate the program. Accepts a single parameter, *int index.*
        - *operator[]* : returns an int reference to the array element specified. Has conditional logic that prevents using an index larger than the *vector_size* of

the BasicVector object by calling *exit 1* to terminate the program. Accepts a single parameter, *int index.* Do not use the *friend* keyword, implement as a class member function

- *front* : returns an int reference to the first array element. Calling *front* on an empty array is undefined behavior, so you do not need to account for it. Just assume it *shouldn't* be done.

- *back* : returns an int reference to the last array element. Calling *back* on an empty array is undefined behavior, so you do not need to account for it. Just assume it *shouldn't* be done.

- *push_back* : accepts an int and puts it in the first open index at the back of the array. No return type (returns void) If the *vector_size* of the array matches the *vector_capacity* before the new element is inserted, you will need to call the *resize* function, and then insert the new value. *vector_size* needs to be incremented after the element is inserted.

- *insert* : accepts two int, an index to insert at, and the value to be inserted. In order for a value to be inserted, all of the elements from the given index to the end of the array need to be shifted once to the right to make room for the inserted value. Be sure to check if this will cause the *vector_size* to overtake *vector_capacity,* and call *resize* before any elements are shifted. No return type (void). Do not allow insertion past the last element in the vector. Increment *vector_size* after successful element insertion.

- *pop_back :* no parameters, no return type. Removes the current last element from the array by setting element to 0 and decrementing vector*_size*. Should do nothing if array is empty.

- *size* : returns *vector_size* field

- *capacity* : returns *vector_capacity* field

- *print* : no return type, no parameters. Outputs the *vector_size* and *contents* of *data* on a single line, use the following format:

  elements(5): 2 -3 16 7 0

- Contains a non-default constructor (you do not need a default constructor):
  - Accepts one parameter for capacity. If the capacity is less than 16, use 16 as the capacity of *data*. Otherwise, set *vector_capacity* to the next largest power of 2, and create *data* with that new capacity.

- Contains a destructor:
  - That sets all the filled elements in *data* to 0, and discards with array properly with delete []

2. Ensure that the member functions of *BasicVector* adhere to certain limits on asymptotic complexity as specified below, with the assumption that our *n* (input) is the size of the array stored in our vector class:
    - *resize*, and any functions that potentially invoke *resize* have a complexity of *O(n)*
    - All other member functions have a complexity of *O(1)*

3. Create an interactive "command" driven prompt system that parses "commands" from the user that are used to invoke the appropriate functions:
    - Initially prompt the user for the starting capacity of the vector and create a vector object by invoking the non-default constructor of *BasicVector*
    - Enter a looping prompt that parses the following commands with the appropriate parameters, and uses the vector created in the previous step to invoke the appropriate member functions:
        - *at <index>*
            - Invoke the *at* function, passing the *index* as its parameter, print result
        - *get <index>*
            - Invoke the *operator[]* function, passing *index* as its parameter, print result
        - *front*
            - Invoke the *front* function, print the result
        - *back*
            - Invoke the *back* function, print the result
        - *insert <index> <value>*
            - Invoke the *insert* function, passing *index* and *value* as its two parameters
        - *push <value>*
            - Invoke the *push_back* function, passing *value* its parameter
        - *pop*
            - Invoke the *pop_back* function.
        - *size*
            - Invoke the *size* function, print the returned value
        - *capacity*
            - Invoke the *capacity* function, print the returned value
        - *print*
            - Invoke the print command
        - *quit*
            - Break out of loop, exit program normally.

Expected prompt/input with example inputs:

```
Enter starting capacity of double vector: 10

Now accepting commands (quit to exit program):
> push 10.4
> size
1
> capacity
16
> print
elements(1): 10.4
> quit

Exiting Program.
```

Important: The commands in the list above need to be implemented exactly as written. Points will be taken off if you modify the syntax/naming convention.

Program 2. (20 points) Generic Vector

Take the code from the *integer_vector.cpp* program, and modify it to create a program (called *generic_vector.cpp*) that does that following:

1. Uses templates in order to allow the *BasicVector* class to not be limited to just containing integers
    - The template will allow you to substitute a placeholder label in the following parts of the code:
        - The data type of the *data* member field
        - The value parameter in the *push_back* and *insert* functions
        - The return type of the *at*, *front*, *back*, and *[]* functions
        - Logic in the *resize* and the constructors that dynamically allocate *data*
2. Modify the interactive prompt to use templates in the following way:
    - Before asking for the starting capacity, prompt the user to specify what data type they want the vector to store in *data*
        - 1 for int
        - 2 for float
        - 3 for double
        - 4 for string
        - 5 for bool

    Expected prompt/input with sample output

    ```
    Specify what data type to store in vector:
          1) int
          2) float
    ```

```
        3) double
        4) string
        5) bool

> 3
Enter starting capacity of double vector: 10

Now accepting commands (quit to exit program):
> push 10.4
> size
1
> capacity
16
> print
elements(1): 10.4
> quit

Exiting program.
```

Template Program Files

Two template files (*integer_vector.cpp* and *generic_vector.cpp)* will be published on Canvas, and will be available for copying (cp) on the Sunlab computer from the following directory:

```
/home/cmpsc122/s18/hw2/
```

Compiling the Program with g++

Use the following command to compile your classes. This is the command I personally use on the Sunlab machines to compile and grade your programs:

```
g++ -Wall -o <output_name> <program_name.cpp>
```

Example:

```
g++ -Wall -o matrix matrix.cpp
```

Remember: Your code must successfully compile without any errors, or a zero will be given for the assignment.

Following Instructions

You are expected to follow the directives laid out in this assignment and the course syllabus. Points will be deducted for incorrectly named files, missing/incorrectly filled out banner comments, not supplying hardcopy submissions in a pocket folder with the appropriate information, and failing to implement features/functionality specified in the assignment. It is expected that you will utilize the provided template files for this assignment, and that you do not modify the names of class members/functions that are provided in the template.

If you have questions about any portions of the assignment or what is expected, contact me for clarification.

CMPSC 122.1
February 15, 2018

Homework 2

Due: March 5, 2018

Submission

- Electronic Submission (Due: One minute before midnight, 11:59 PM, Sunday March 4, 2018)

  - Your two source code files (*integer_vector.cpp, template_vector.cpp*)
  - Submitted using the **mail122** command on the Sunlab machines

- Hardcopy Submission (Due: Beginning of class, 6:00 PM, Monday March 5, 2018)

  - Printed hardcopies of each of the two source code files
  - The pages of each program should be stapled together
  - Submitted in a pocket folder with your name, the course, and the section number on the front