Lab 10

Comp 11 - Summer Session — My Vector

10.1 Description

In this lab we will implement the vector data structure. Much of the code has been implemented for you–but you need to practice typing in the code and building the class first and foremost.

There are only **3 places** where you will make code changes, all in the push_back() member function.

The step by step of what is going on is listed below that are missing are described below.

Our objectives are the following:

- Implement a class called myVector.
- Implement the push_back member function.
- push_back() should dynamically allocate memory, and reallocate memory if more space is needed. The initial memory allocation should allocate 10 items(this number is arbitrary).
- 10 thus becomes our initial capacity for how many items we can store in our vector before needing to allocate more memory.
  - You have private member variables called pos(where you are) and capacity that monitor how much space you have.
  - If you run out of space, you will allocate a new array twice as big(make sure to multiply and update capacity), copy all of the elements to it, and then point your old array to this new array.
  - You will also remember to delete your old memory to avoid memory leaks!
- Convince yourself why these member variables and member functions need to be public or private.

10.2 Files

You may use the following code to help get you started. You will notice that this class is also templated for you, so you can use CompVector on whatever data type you like!

```
#include <iostream>
#include <vector>

// In this lab, you are going to re-implement vector.
template <class T>
class CompVector{
private:
    int pos;  // Keeps track of where to insert new elements in
    vector
    int capacity; // How big is our dynamic array
    T* elements; // Our dynamic array
public:
    // Default Constructor
    // Note that we set pos to 0
    // elements is NULL initially
    CompVector(){
        pos =0;
        elements = NULL;
    }

    // Default Destructor
    // Always remember to delete memory, and we use []'s because
    // it is an array
    ~CompVector(){
        delete[] elements;
    }

    // push_back -- add an item
    void push_back(T element){
        if(elements == NULL){
            // (1) Allocate memory and update capacity

            // ... your code here .. 2 lines
        }
        else if(pos > capacity-1){
           // Create a new temporary array
           T* new_elements = new T[capacity*2];
           // (2) Copy all of the elements into the temporary
           // array using a loop.

           // .... your code here ... 1 loop


           // Update capacity
            capacity = capacity * 2;
            // (3) Delete our old memory
            // Ask yourself, which memory are we deleting?
            // Is it elements or new_elements?

            // Point elements
            // Why do we need to do this?
```

```
            // Answer: ----------------------
            elements = new_elements;
        }

        // Set element to position
        elements[pos] = element;
        // Update position
        pos++;
    }


    // Return the size of the vector
    int size() const {
        return pos;
    }

    T at(int index) {
        return elements[index];
    }

};



int main(){

    // Create our version of vector and the standard libraries
    std::vector<int> cppVector;
    CompVector<int> ourVector;

    // Add some items
    for(int i =0; i < 10; ++i){
        cppVector.push_back(i);
        ourVector.push_back(i);
    }

    std::cout << "cppVector size:\t" << cppVector.size() << "\n";
    std::cout << "ourVector size:\t" << ourVector.size() << "\n";
    std::cout << "cppVector[9]:\t" << cppVector.at(9) << "\n";
    std::cout << "ourVector[9]:\t" << ourVector.at(9) << "\n";

    // Add some more items again!
    for(int i =0; i < 10; ++i){
        cppVector.push_back(i+10);
        ourVector.push_back(i+10);
    }

    std::cout << "cppVector size:\t" << cppVector.size() << "\n";
    std::cout << "ourVector size:\t" << ourVector.size() << "\n";
    std::cout << "cppVector[19]:\t" << cppVector.at(19) << "\n";
    std::cout << "ourVector[19]:\t" << ourVector.at(19) << "\n";

    return 0;
}
```

Listing 10.1: CompVector starter code

10.3 Output

```
mike:Lab10$ ./vector
cppVector size: 10
ourVector size: 10
cppVector[9]:   9
ourVector[9]:   9
cppVector size: 20
ourVector size: 20
cppVector[19]:  19
ourVector[19]:  19
```

10.4 Refresher

You can use std::vector to understand the behavior of the data structor.

10.5 Submission

```
provide comp11 lab10 your_file.cpp README
```

Listing 10.2: Submit Assignment

10.6 Going Further

Did you enjoy this lab? Want to try out some additional commands to go further?

- Try implementing an erase or pop method that removes the past element from the vector. Think about where your indexes can go out of bounds.
- If your internal array becomes small again (after implementing the above), you may also reallocate a smaller array to save memory (that is, if you allocate 100000 items, then delete all of them, you do not need that storage).