PIC 10B Section 1 - Homework # 3 (due Friday, April 19, by 6 pm)

You should upload each .cpp and .h file separately and submit them to CCLE before the due date/time! Your work will otherwise not be considered for grading. Do not submit a zipped up folder or any other type of file besides .h and .cpp.
Be sure you upload files with the precise name requested of you and that it matches the names you used in your editor environment otherwise there may be linker and other errors when your homeworks are compiled on a different machine. Also be sure your code compiles and runs on Visual Studio 2017.

Do not submit a main routine for this homework.

WRITING A VECTOR CLASS

In this homework, you'll get to write your very own vector-type class. The class will behave as a **std::vector<std::string>** object, but you are *not allowed to use the standard library vector templated class or any other container classes of the C++ standard besides std::string.*
In doing this assignment, you'll be able to better understand how **std::vector** works behind the scenes.
Please refer to the syllabus for how the work will be graded: note that more than half of the marks come from good coding practices and code documentation.

You should write a class object of type **VectorString**, belonging to a **pic10b** namespace, that can store objects of type **std::string**. The class should have the following:

- member variables **vec_size**, representing the number of elements in the vector, and **vec_capacity**, storing the size of the underlying dynamic array in memory, which must always be at least as big as **vec_size** (see later remarks);
- a member variable storing a smart pointer variable to the dynamic array of **std::string**;
- a default constructor **VectorString()** that allocates a dynamic array of **std::string** of one element (so **vec_capacity** is 1), but which has a **vec_size** of 0;
- a constructor **VectorString(size_t)** that allocates a dynamic array of *twice* the input size: **vec_size** should be set to the input and **vec_capacity** is twice the input with all of the string variables set to the empty string;
- a constructor accepting a size and input string value, that does the same as above with the input size and capacity, but which initializes all the string variables to the input string;
- a copy constructor **VectorString(const VectorString&)** and move constructor **VectorSring(VectorString&&)**;
- a copy assignment **VectorString& operator=(const VectorString&)** and move assignment operator **VectorString& operator=(VectorString&&)**;
- a **size** member function returning the size of the vector;
- a **capacity** member function returning the capacity of the vector;
- a **push_back** member function accepting a **std::string** that adds an element to the end of the vector if the capacity allows and otherwise creates a new dynamic array of twice the former capacity, moving all the elements over and adding the new element;
- a **pop_back** member function that ”removes” the last element of the vector by updating its **vec_size** value (the value will remain there in memory but by changing the size, you can logically overwrite it if there is another insertion);
- a **deleteAt** member function accepting an index value that removes the element at the given index from the vector, shifting elements backwards as appropriate (once again, the ”last” value will still be there in memory but can be overwritten);
- an **insertAt** member function accepting an index and **std::string** value that inserts an element at the input position, moving elements (including the element previously there) forward and when this would push the vector size above the capacity then all the elements should be moved over to a new dynamic array of twice the former capacity before this process; and

- an **at** member function, **overloaded on const**, accepting an index value that returns the element at the given index *by reference / reference to const*. Do not worry about throwing exceptions as the real **std::vector::at** function does. Not yet...

Note that when the size exceeds the capacity then the elements are relocated to a larger block of memory with twice the former capacity! The vector is indexed from 0.

Your job is to write a header file **VectorString.h** and accompanying file **VectorString.cpp** with the implementations so that a **.cpp** file can use the **VectorString** class. An example set of code and output are given below: your actual homework will be graded against a different **.cpp** file.

Do not submit a main routine!

The only header files that you may use are: <**string**>, <**memory**>. **Note in particular that <vector>, <deque>, etc., are not allowed. The entire purpose of this assignment is that you implement for yourself what a vector would do for you.**

Once again: do not submit a main routine!

You may assume the user will only enter valid parameters, i.e., use indices that are within range with **at**; they won't **pop_back** on an empty vector, etc.

Example:

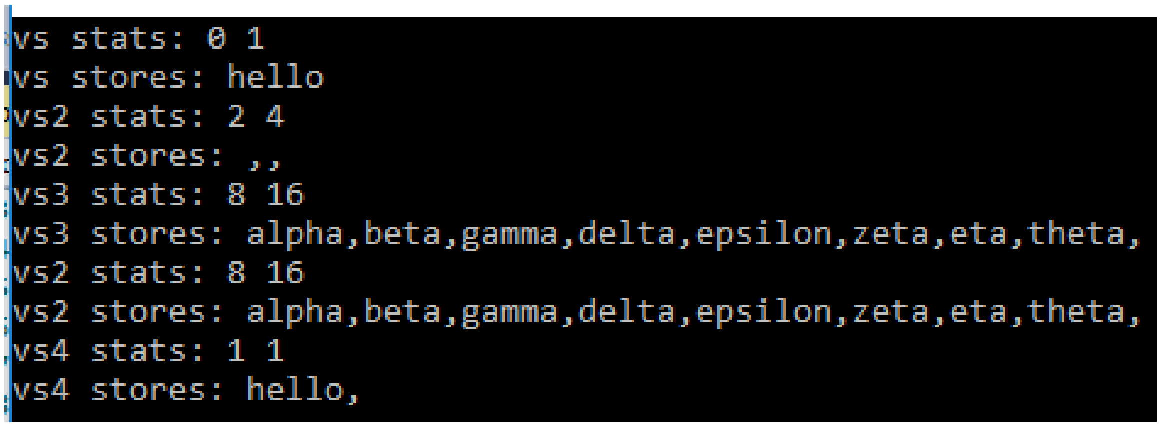
Example code using this class is below

```
#include"VectorString.h"
#include<iostream>
#include<string>

pic10b::VectorString foo() {
  return pic10b::VectorString(2);
}

int main() {
  pic10b::VectorString vs;
  std::cout << "vs stats: " << vs.size() << " " << vs.capacity() << '\n';
  vs.push_back("hello");
  std::cout << "vs stores: " << vs.at(0) << '\n';

  auto foo_out = foo();
```

```
pic10b::VectorString vs2(std::move(foo_out));
std::cout << "vs2 stats: " << vs2.size() << " " << vs2.capacity() << '\n';

std::cout << "vs2 stores: ";
for (std::size_t i = 0, total_size = vs2.size(); i < total_size; ++i) {
  std::cout << vs2.at(i) << ",";
}
std::cout << '\n';

pic10b::VectorString vs3(4, "beta");
vs3.pop_back();
vs3.push_back("delta");
vs3.push_back("epsilon");
vs3.at(1) = "gamma";
vs3.insertAt(0, "alpha");
vs3.push_back("zeta");
vs3.push_back("theta");
vs3.insertAt(vs3.size() - 1, "eta");
vs3.deleteAt(3);

std::cout << "vs3 stats: " << vs3.size() << " " << vs3.capacity() << '\n';
std::cout << "vs3 stores: ";
for (std::size_t i = 0, total_size = vs3.size(); i < total_size; ++i) {
  std::cout << vs3.at(i) << ",";
}
std::cout << '\n';

vs2 = vs3;

std::cout << "vs2 stats: " << vs2.size() << " " << vs2.capacity() << '\n';
std::cout << "vs2 stores: ";
for (std::size_t i = 0, total_size = vs2.size(); i < total_size; ++i) {
  std::cout << vs2.at(i) << ",";
}
std::cout << '\n';

const auto vs4 = std::move(vs);

std::cout << "vs4 stats: " << vs4.size() << " " << vs4.capacity() << '\n';
std::cout << "vs4 stores: ";
for (std::size_t i = 0, total_size = vs4.size(); i < total_size; ++i) {
  std::cout << vs4.at(i) << ",";
}
```

```
  std::cout << '\n';

  std::cin.get();
  return 0;
}
```

generating the following output:



```
vs stats: 0 1
vs stores: hello
vs2 stats: 2 4
vs2 stores: ,,
vs3 stats: 8 16
vs3 stores: alpha,beta,gamma,delta,epsilon,zeta,eta,theta,
vs2 stats: 8 16
vs2 stores: alpha,beta,gamma,delta,epsilon,zeta,eta,theta,
vs4 stats: 1 1
vs4 stores: hello,
```

Remarks: Besides the facts that we haven't templated the class (which will be covered later) and that our dynamic memory management can be improved (also later), this is very similar to how the real **std::vector** works. With a real **std::vector**, a special class object called an allocator allocates unconstructed heap memory, i.e., the memory cells, unless filled with legitimate vector elements, are not even constructed by a default constructor.

Implementing the copy constructor is a little delicate. A real copy constructor making use of smart pointers needs to take into account not only the memory managed but also the "deleter". This will be a detail this isn't pursued in this assignment. For the copy constructor, we will just worry about copying values in memory.

Some initial guidance to get you on your way...

1. First make sure you can create a dynamic array of **std::string**s and work with them with a smart pointer as a wrapper.
2. You are working with smart pointers so you don't need to worry about the destructor unless you do something really strange with your memory.
3. To copy, you'll need to create a new dynamic array (with a smart pointer wrapping it) of the appropriate size and then copy each element over one at a time.
4. For **push_back**, if the new size does not go beyond the capacity, all you need to do is subscript your smart pointer at the new size value and increase the size; if you need to create more memory, be sure to keep track of the location in memory where your smart pointer used to point (you may need to use **release**).
5. For **pop_back**, you need only decrease the size. This effectively chops off the last element.
6. Draw yourself a diagram of how the **insertAt** and **deleteAt** functions work. You'll have to shuffle memory around the old fashioned way instead of using the **insert** and **erase** functions provided by **std::vector**s. The PIC 10A notes on vectors may be helpful.