CSE 232, Lab Exercise 12

On Your Own

This is going to be an "on your own" lab. There is no lab the week of Thanksgiving, but I don't know how you get Project 10 done if you don't do the work listed here. You tests on Mimir will be the record of whether you did the lab or not. ***Remember , no attendance to labs at all this week!***

The Problem

We are going to work on making our own container class with dynamic memory. We are going to build our own version of a vector that is templated, as it should be.

Namespaces

We have talked a lot about namespaces but really haven't used them ourselves. It turns out we can easily define our own namespaces to separate code that we have defined from code in the standard library. We are going to do that here, define a namespace called `student` in which we will place a class `vector`. Though `vector` has the same name as in the STL, because we store it in our own namespace we can change just a single line of code to use our namespace student version of vector or the stl version.

Making a namespace

Turns out it is rather easy to make a namespace. You can enclose all the code you want to define in a namespace in brackets lead by a declaration of the namespace. For example:

```
namespace student {
   template<typename T>
   class vector{
   …
}
```

Everything in curly brackets is now part of the `student` namespace. You can add things to the student namespace by simply adding other namespace student elements in the code. To use your new namespace (with your new vector class), you would use:

```
student::vector
```

instead of

```
std::vector
```

Multiple mains

We are going to do this in stages. To help with incremental development, we provide multiple `lab12_mainX.cpp` associated with it (`lab12_main1.cpp, lab12_main2.cpp`, etc.) so that you can test locally. You can then test with the various Mimir test cases afterwards.

Task 1

1. Create a `lab12_vector.h` file
    a. because we are creating a templated class, everything goes in the header. We won't have a `vector.cpp`.
2. In `lab12_vector.h` create a namespace `student` as described above.
3. In namespace `student`, create a templated `vector` class with the following private features:
    a. a pointer `data_` of template type which points to dynamically allocated memory
    b. a `size_t capacity_,` default 10, which is the capacity of the vector. Remember, capacity is how many elements it could hold before having to grow.
    c. a `size_t size_,` default 0, the number of elements actually in the vector.

4 methods

1. a 1-arg constructor with argument `size_t capacity`. It will do the following:
    a. set `capacity_` to the argument (or 10 if no argument using the default)
    b. set `size_` to 0
    c. set `data_` to point to a dynamically allocated memory element of template type, and the size of that array will be the `capacity_` value (default 10)
2. a 1-arg constructor with an `initializer_list`. It will do the following:
    a. set `capacity_` and `size_` to `initializer_list.size()`
    b. set `data_` to point to a dynamically allocated memory element of template type, the size of which is the `initializer_list.size()` .
    c. copy the values from the `initializer_list` to `data_`
        i. remember, the `initializer_list` should be of the template type.
3. a method `size_t capacity()`
    a. returns the `capacity_` value
4. a method `size_t size()`
    a. returns the `size_` value

Run this code against the main labeled `lab12_main1.cpp`. You can then try the first test on Mimir to confirm.

Show your TA

Task 2

Create two important methods

- `void push_back(templatetype val);`
- `templatetype& operator[](size_t val);`

Add to `lab12_vector.h` to have these new declarations and create the method in the namespace student

push_back

Takes a single parameter of template type and adds it to the end of the vector.

Remember the special functionality of `push_back`. It adds the provided parameter to the end of the vector. The parameter should be of template type. The end of the vector is the next open space. If `push_back` of the parameter does not exceed `capacity_`, just add it to the end and update `size_`

However, the next `push_back` might exceed the size of your memory. That's fine, remember a vector is never full (well, unless your machine runs out of memory). The `capacity_` of `vector` indicates how many elements it *can hold* before the memory pointed to by `data_` needs to grow. `size_` indicates how many it *actually*, presently holds. If a `push_back` exceeds `capacity_`, (i.e., if `size_` equals or exceeds `capacity_` by adding this new parameter) you must do the following:

1. Create a new memory allocation that is ***twice the size*** of present `capacity_`
    a. call the new pointer `new_data`
2. copy all the elements in `data_` into `new_data`
3. swap `data_` and `new_data` (now `data_` points to the larger memory)
4. delete `new_data` (which now points to the old memory)
5. update `capacity_`
6. `push_back` the new element onto vector
7. update `size_` by 1

operator [ ]

This is the operator that allows us to work with individual elements in our vector. `operator[]` receives a single `size_t` argument which is the ***index*** in the original call and returns a ***reference*** to the element. It needs to be a reference so that what is returned is an lvalue (a location) so that an operation like

`v[2] = 0;` will compile. The code should do one of two things:

- if the `index` requested is within the vector, return a reference to the element at that location
- if it is not, throw a `range_error` (requires `#include<stdexcept>`) with an appropriate message. Look at the last lab.

Run your code with `lab12_main2.cpp` to see if it works. Then test 2 on Mimir to confirm.

Show your TA when you have finished.

Task 3

If you got this far, here is some more fun stuff. First the easy ones. Write the following two functions:

- a `front()` method that returns a **reference** to the front element of the vector

- a `back()` method that returns a **reference** to the back element of the vector
    - what should these do if the vector is empty? We have choices but let's throw another `range_error` in this case.

Then the harder ones. If you do dynamic memory, you should make your own copy constructor and `operator=`. So write the next three functions:

- `vector(vector& v);`
- `vector& operator=(vector& v);`
- `~vector()`

The copy constructor should:

- copy both `capacity_` and `size_` from `v` to the newly created instance
- in the new created instance
    - allocate the new required memory
    - copy the elements from the argument vector to the newly created instance.

The `operator=` should basically do the same thing

- think of using the copy and swap idiom discussed in the videos and in the slides

Then `~vector()`, the destructor to `delete` memory when the created element goes out of scope or is otherwise destroyed.

Run your code with `lab12_main3.cpp` to see if it works. Then test 3 on Mimir to confirm.

- the test on Mimir for the destructor is a little hard to read. Trust me, if it passes it worked and if it didn't then you need to look at your code again.

 Show it to your TA

Task 4

Finally, if you are still here, let's make

- a `clear()` method which removes all the data
    - resets both `capacity_` and `size_`, to 0
    - deletes the existing array memory (if there is any)
- a `pop_back` method. FYI one would only do the following on a system with limited memory, but it is still good practice:
    - it returns a template type, the last element of the vector
    - removes the last element of the vector. The vector `size_` is now one less
    - if after the removal the vector reaches ½ capacity (the `size_` is half of the `capacity_`), do what we did with `push_back` but in reverse.

- make a `new_data` pointer of ***half capacity***
- copy the `data_` to the `new_data`
- clean up as previously

Run your code with `lab12_main4.cpp` to see if it works. Then test 4 on Mimir.

 Show it to your TA