PIC 10B: Homework 3

Due February 22

Submitting Your Homework

The zip file you extracted to find this pdf also includes the files: `vector.hpp`, `vector.cpp`, `my_main.cpp`, and `std_main.cpp`. In this assignment, you will edit and submit `vector.cpp` so that it builds and runs with `vector.hpp` and `my_main.cpp`. The output from building and running `std_main.cpp` should be similar, but may vary from compiler to compiler.

- Upload `vector.cpp` to Gradescope before the deadline.
- Name the file exactly as specified, with no extra spaces and no capital letters.
- Do **not** upload `vector.hpp`, `my_main.cpp`, or `std_main.cpp` to Gradescope.
- Do not enclose `vector.cpp` in a folder or zip it. You should be submitting exactly one file and it should have the extension `.cpp`.
- When working on your homework assignment, do **not** edit `vector.hpp`.
- Be sure that your code builds and runs with my files on Visual Studio 2022.

Vectors of `int`s

A `my::vector` consists of three member variables:

- `siz` - the *size* of the vector; accessible using `size()`.
- `cap` - the *capacity* of the vector; accessible using `capacity()`.
- `ptr` - a *pointer* to the data stored on the heap.

This should allow us to model `my::vector` based on a `std::vector<int>`. Recall that for a `std::vector<int>`, the *capacity* is the size of the storage space currently allocated for the vector, expressed in terms of the number of `int`s. This is not necessarily equal to the vector's size! However, at any time, the following inequality is guaranteed to be true.

$$\text{size} \leq \text{capacity}$$

The potential for extra space accommodates growth (via `push_back`) without the need to reallocate on each individual insertion.

Notice that the capacity does not suppose a limit on the size of the vector. When more capacity is needed, it is increased automatically. The capacity of a vector can be explicitly altered by calling the `reserve` member function. The code in `std_main.cpp` demonstrates how these concepts work thoroughly for `std::vector<int>`. The following code gives the idea as quick as possible. This shorter piece of code is included as `siz_cap.cpp`.

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

void print_info(const std::vector<int>& v, size_t N) {
    std::cout << "            size: " << v.size()     << '\n';
    std::cout << "        capacity: " << v.capacity() << '\n';

    if (N < 10) { std::cout << ' '; }
    std::cout << "first " << N << " elements: ";

    for (size_t i = 0; i < N; ++i) {
        std::cout << *(v.data() + i) << ' ';
    }
    std::cout << '\n' << std::endl;
}

int main() {
    std::vector<int> v;
    int* ptr = v.data();          // store a copy of the underlying pointer;
    print_info(v, v.capacity()); // see that v has 0 size and capacity

    for (int i = 1; i <= 9; ++i) {
        v.push_back(i * 11);     // we'll push_back 11, 22, 33, ..., 88, 99

        if (ptr != v.data()) {
            std::cout << "there's been a memory reallocation" << std::endl;
            ptr = v.data();
        }
        print_info(v, v.capacity());
        // fine since we use *(v.data() + i) instead of v[i]
    }
    return 0;
}
```

This is the behavior we want to mimic for our own vector class `my::vector`.

Tasks

0. I've already defined some member functions of `my::vector` for you:

    - `size_t size() const` and `size_t capacity() const`,
    - `bool empty() const` and `void pop_back()`,
    - `void new_capacity(size_t)`, `const int* data() const`, and `int* data()`.

    I've also defined the free function `void my::swap(vector&, vector&)`.

1. Define the default constructor `vector()` and destructor `~vector()`.

    `std_main.cpp` demonstrates what the default constructor should accomplish, and the destructor should free the memory associated with `ptr`.

2. Define overloaded `operator[]`s so that we can access and modify elements as normal.

3. Define the constructors:

    - `vector(size_t);`
      Value initialization will be most efficient for the array on the heap.
    - `vector(size_t, int);`
      One may as well use default initialization for the array on the heap before assigning the correct value to the elements.
    - `vector(std::initializer_list<int>);`
      For a `std::initializer_list<int> il`, `il.size()` does what you expect; you can loop through the elements using a range-based `for` loop.

4. Define the member function `void clear()`.

    `std_main.cpp` demonstrates what it should accomplish.

    Like `pop_back`, it has a very simple definition.

5. Define the following member functions (all of which are demonstrated in `std_main.cpp`). The existence of the private member function `new_capacity` should help you a lot!

    - `void reserve(size_t);`
      This function should increase the capacity if necessary.
    - `void push_back(int);`
      When the capacity is 0, this function will need to update the capacity to 1. Otherwise, when `siz == cap`, it should double the capacity.
    - `void resize(size_t n, int i);`

- When `n > size`, you will need to write some new elements.
- Before that, you will need to check if it is necessary to increase capacity. If it is necessary to increase capacity,
  - when doubling the capacity suffices, do that;
  - otherwise, simply set the capacity to be equal to the new size.

Compilers vary on how they implement this member function. When increasing the capacity, XCode updates the new capacity according to the old capacity, OnlineGDB updates the new capacity according to the old size, and Visual Studio does something different. Because it is the simplest, we will follow XCode, choosing to update the new capacity according to the old capacity, not the old size. We want the output from lines 134 to 167 of `my_vector.cpp` to be as follows, except with the question marks replaced by random numbers.

```
there's been a memory reallocation
             size: 7
         capacity: 10
first 10 elements: 11 22 33 44 55 66 77 ? ? ?

there's been a memory reallocation
             size: 16
         capacity: 20
first 20 elements: 11 22 33 44 55 66 77 8 8 8 8 8 8 8 8 8 ? ? ? ?

there's been a memory reallocation
             size: 24
         capacity: 40
 first 0 elements:

there's been a memory reallocation
             size: 81
         capacity: 81
 first 0 elements:
```

6. Define:

- the member function `void swap(vector&);`
- the move constructor `vector(vector&&);`
- the move assignment operator `vector& operator=(vector&&);`
- the copy constructor `vector(const vector&);`
- the copy assignment operator `vector& operator=(const vector&);`

`std_main.cpp` demonstrates what they should accomplish. In short,

- swapping and moving should steal the resource, and therefore steal `siz` and `cap`;
- when copy constructing, the copy will have `siz == cap`.