Chap 16 C– Functors and Lambdas

1. Using generate_n to generate 20 numbers from 0 to 19 in a vector of ints.
   i. Pass a function pointer to generate_n
   ii. Pass a functor object to generate_n
   iii. Pass a lambda setting the implicit reference in the capture clause

2. The template function *remove* will **pseudo** remove values from a container. I say pseudo because no template function can resize a container class. The remove() will simply move the elements around so that it looks like the values have been removed.

   For example let's say we have a vector and we want to remove all the elements that equal 1.

```
vector<int> vi{ 1,2,3 };
remove(vi.begin(), vi.end(), 1);
```

   When we call this the values 2 and 3 will be copy forward over the 1, but size of the vector will not change so its value will equal {2,3,3}. The 2 and 3 were copied forward to replace the 1 but the size of the container is still a size of three so the last element from the original state of vi is still there.

   The good news is remove() returns an iterator that points to the **new** end of the container. So we can call the container's erase() method to physically remove the unwanted elements.

```
vector<int> vi{ 1,2,3 };
auto newend = remove(vi.begin(), vi.end(), 1);
vi.erase(newend,vi.end());       // vi={2,3}
```

   The remove_if() template function works the exact same way except it takes a lambda/function/function object and removes all elements where the true is returned. For example to remove all odd numbers we do the following:

```
vector<int> vi{ 1,2,3,4 };
auto newend = remove_if(vi.begin(), vi.end(), [](int x) {return x % 2; });
vi.erase(newend,vi.end());       // vi={2,4}
```

   So using remove_if and erase perform the following:

   a. Where vector<int> vi = {0,1,2,3,4,5,6,7,8,9} remove all numbers NOT divisible by 3
   b. Prompt the user for a sentence and populate each word in a vector of strings. Remove all strings whose length is greater than 4 and print out the remainders.
      Example: enter a string: a dog and elephant walked into a bar
      Result: a dog and into a bar

c. You can use remove_if/erase on a string as well (strings have begin() and end() methods like other containers. Prompt the user for a string and remove all whitespaces using remove_if/erase.

d. The template function remove_copy_if() acts like remove_if() except it doesn't remove any elements. It makes a copy of the container with the elements removed. Using the lambda from problem a. make a copy of the vi vector that contains only the number divisible by 3.

3. Using generate_n generate the first 20 Fibonacci numbers in a vector of ints.
4. A **high order function** is a function that can either take as an argument other functions or return as a value a function.

   Any function that takes a function/functor/lambda is a high order function (remove_if, for_each, generate, etc…).

   How does the STL make these high order functions that can take either a function, functor, or lambda? Well remember to make generic functions in C++ you use templates.

   Make your own version of the Algorithm's library **for_each()** that takes a begin and end iterator and a function/functor/lambda and applies the lambda to each element.

5. High order functions can also return a function object. In C++ we can return a lambda from a function. For example:

```
auto equals(int i) {
        return [i](int x) { return x == i; };
}

void testHighOrder() {
        vector<int> vi{ 1,2,3,4 };

        auto newend = remove_if(vi.begin(), vi.end(), equals(2));
        vi.erase(newend, vi.end());     //  vi== {1,3,4}

        newend = remove_if(vi.begin(), vi.end(), equals(3));
        vi.erase(newend, vi.end());     //  vi== {1,4}
}
```

   Create your own high order function called divby(int x) that returns a lambda function that returns true if a passed integer to it is divisible by x.

   Use this to remove numbers from a vector of ints that are divisible by 3.

6. If you wanted to time how long a function took in milliseconds to run you could write the following:

```
auto start = chrono::high_resolution_clock::now();//time before calling function
functor();
auto stop = chrono::high_resolution_clock::now(); //time after calling function
long long duration = chrono::duration_cast<milliseconds>(stop - start).count();
// duration is the number of milliseconds
```

high_resolution_clock is a class from chrono include file. Duration_cast will cast the difference to a particular time type (seconds, milliseconds, microseconds, etc…).

Make a generic (that means ***template)*** function called **TimeMe** that takes as an argument a functor, lambda or function pointer and returns how long it took to execute it in milliseconds.

In other words we want something that looks like this: duration = TimeMe(*lambda*);

Example:

```
auto len = 2 * 1E4;         // 2 * 10000 = 20000
vector<int> vi;

// I want to time how long it takes to fill the vector
// with the value 42. So I make a lambda function that calls fill_n
// and pass that lambda to TimeMe.

const auto duration = TimeMe(
         [&vi,len]() {fill_n(back_inserter(vi), len, 42); }
);

cout << "Fill took " << duration << " milliseconds\n";
```

Output should be something like:

```
Fill took 9 mS milliseconds
```