Question 0: Lecture 3 Are the preconditions for the following functions correct? If not, where does it go wrong, and how would you rewrite them?

(a)

```
/* Normalize a vector in place
 *@param[in] _v_ stl vector of doubles to normalize
 *
 *@post the sum of all the elements of _v_ is 1
*/
void normalize(std::vector<double>& v){
  double sum = 0.0;
  for(auto it = v.begin(); it != v.end(); it++){
    sum += *it;
  }
  for(auto it = v.begin(); it != v.end(); it++){
    *it/=sum;
  }

}
```

@pre v is a nonempty vector and the sum of all its elements remain nonzero
(or we can restrict every ele in vector to be nonnegative and vector has at least one positive element)

→ if sum fails to increment from zero and remains zero, problem would entail when we enter the second for loop — value would be divided by zero.

(b)

```
/* Compute the cosine similarity between two vectors
* @param[in] _v_ stl vector of doubles
* @param[in] _w_ stl vector of doubles
* @return the cosine of the angle between _v_ and _w_
*
* @pre _v_.size() == _w_.size()
* @pre for all 0<i<_v_.size() _v_[i]!=0 and _w_[i]!=0
*/

double cosine_sim(const std::vector<double>& v, const std::vector<double>& w){
  double num = 0.0;
  double norm_v = 0.0;
  double norm_w = 0.0;

  if(v.size() != w.size()){
    std::cout<<"vectors are not the same length"<<std::endl;
    return 2.0;
  }
  for(unsigned int i=0; i<v.size(); i++){
    num += v[i]*w[i];
    norm_w += w[i] * w[i];
    norm_v += v[i] * v[i];
  }

  return num/(sqrt(norm_w)*sqrt(norm_v));
}
```

Don't need to be "all"

error! This condition needs to be revised. Notice that on the last line, we take square root on norm of vectors v and w.

revised @pre

This means that we just need to ensure that each norm is positive. i.e. there exists some i, $0 \le i < v.size()$ s.t. $v[i] \neq 0 \Rightarrow$ norm of $v > 0$
there exists some j $0 \le j < w.size()$ s.t. $w[j] \neq 0 \Rightarrow$ norm of $w > 0$
would be an ideal precondition.

Question 1: Lecture 2

```
/** Find the first element in a sorted array that does not
 *  satisfy a predicate.
 * @param[in] _a_          Array to search.
 * @param[in] _low_,_high_  Search in the index range [_low_, _high_).
 * @param[in] _pred_       Predicate to consider
 * @return  An index into array _a_, or _high_.
 *
 * @tparam T     Type of the elements.
 * @tparam Pred  Type of the predicate with signature:
 *                bool operator()(const T&)
 *
 * @pre  0 <= _low_ <= _high_ <= Size of _a_.
 * @pre  There exists k in [_low_, _high_] such that
 *   for all i with _low_ <= i < k, _pred_(_a_[i]), and
 *   for all j with k <= j < _high_, !_pred_(_a_[j]).
 *
 * @post For all i,j with _low_ <= i < result <= j < _high_,
 *           _pred_(_a_[i]) and !_pred_(_a_[j]).
 *
 * Performs O(log(_high_ - _low_)) operations.
 */
template <typename T, typename Pred>
int lower_bound_pred(const T* a, int low, int high, const Pred& pred) {
  // Someone has implemented this for you.
}
```

Suppose you are given an implementation of the `lower_bound_pred` function with the above specification. Using it, implement a `lower_bound` function with the specification below; **(a)** first, with a functor, and **(b)** second, with a lambda function.

```
//  CME212, Winter 2020
//  Paper Exercise 1 - Question 1: Lecture 2
//
//  Created by Chih-Hsuan Kao on 1/25/20.
//  Copyright © 2020 Chih-Hsuan Kao. All rights reserved.
//

#include <iostream>
class Pred {
    int value;

    Pred(int val)
        : value(val)
    {}

    bool operator<(int another_val) const {
        return value < another_val;
    }
};

/**
 * Find the first element in a sorted array that is not less than a value.
 * @param[in] a Sorted array to search.
 * @param[in] low Search in the index range [_low_, _high_).
 * @param[in] high Search in the index range [_low_, _high_).
 * @param[in] v value to search for
 * @return an index into array _a_, or _high_.
 *
 * @tparam T Type of the elements.
 *          T is comparable with 'bool operator <(T,T)'.
 *          Operator < is irreflexive , i.e. operator <(x, x) returns false.
 *
 * @pre 0 <= _low_ <= _high_ <= Size of _a_.
 * @pre For all i,j with _low_ <= i < j < _high_, !(_a_[j] < _a_[i])
 *
 * @post For all i,j with _low_ <= i < result <= j < _high_, _a_[i] < _v_ and !(_a_[j] < _v_).
 *
 * Performs O(log(_high_ - _low_)) operations.
 */

/** Implement a lower_bound function with a functor */
template <typename T>
int lower_bound(const T* a, int low, int high, const T& v) {
    lower_bound_pred(a,low,high,Pred(v));
}

/** Implement a lower_bound function with a lambda function */
template <typename T>
int lower_bound(const T* a, int low, int high, const T& v) {
    lower_bound_pred(a,low,high,[v](int anotherval){return v < anotherval;});
}
```

Question 2: Lecture 4

(a) What are the 4 operators and the 2 functions an iterator needs to implement?

(b) As long as an iterator class has the above methods defined, it can be used with any of the `stl` methods that use iterators. We will write an iterator class for a book that is represented as a `std::vector<std::vector<string>>`. In this case, each inner `std::vector<string>` represents a sentence of the book, split into individual words. Write a wrapper iterator that iterates over all the words in a `Book`. The starter code is given below as well as in `q2_starter.cpp`

(c) Write a functor that implements a comparator between two strings based on the length of the strings, returning `true` if and only if the first string is shorter than the second string.

(d) The code below reads in Moby Dick into the book class. Use the iterator class, the functor and an stl algorithm from the algorithms package to find the longest word in Moby Dick.

(a)

	ReturnType	OperatorName	Argument(s) (if any)
Forward Increment:	Iterator	operator++()	
Equality comparison:	bool	operator==()	another iterator
Inequality comparison:	bool	operator!=()	another iterator
Dereferencing:	value	operator*()	

	Function	ReturnType
First element it iterates	begin()	Iterator (pointing to beginning)
Last element it iterates	end()	Iterator (pointing to past-the-end)

(b)(c) see code from next page

(d)

```
chkao831@rice10:/farmshare/user_data/chkao831/cme211-chkao831/remote$ g++ -std=c++11 main.cpp -o main
chkao831@rice10:/farmshare/user_data/chkao831/cme211-chkao831/remote$ ./main
matches?-tinder?-gunpowder?-what
```

```cpp
#include <algorithm>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

//  CME212, Winter 2020
//  Paper Exercise 1 - Question 2: Lecture 4
//
//  Created by Chih-Hsuan Kao on 1/27/20.
//  Copyright © 2020 Chih-Hsuan Kao. All rights reserved.
//

//alias pointer for convenience
using book_ptr = std::vector<std::vector<std::string>>*;

//Helper function to split string into a vector based on a splitting character.
//This function is used to load the book.
const std::vector<std::string> split(const std::string& s, const char& c)
{
    std::string buff{""};
    std::vector<std::string> v;

    //iterate character by character and push the buffer to the vector if the
     splitting char is reached.
    for(auto n:s)
    {
        if(n != c) buff+=n; else
        if(n == c && buff != "") { v.push_back(buff); buff = ""; }
    }
    if(buff != "") v.push_back(buff);

    return v;
}

// Forward declaration of Book class.
class Book;

/** Iterator for Book */
class BookIter
{
    public:

    /** Increments to the next word in the book class. */
    BookIter operator++()
    {
        //if reached pass-the-end, return the end Iterator with nullptr
        if(idx_out == (*ptr_vec).size()){
            return BookIter(ptr_vec,idx_out,0);
        //otherwise, not yet til the end
        } else {
```

```
        //inside of nested vector, if haven't reached the end
        if(idx_inn + 1 < (*ptr_vec)[idx_out].size()){
            //increment the inner index
            idx_inn++;
        //If reached the end inside of nested vector, increment outer one
        } else {
            //do-while: execute at least once
            //as long as not reached outer end, skipping non-empty outer
             vec
            do {
                idx_out++;
            } while (idx_out < (*ptr_vec).size() &&
             (*ptr_vec)[idx_out].empty());
            //go to start of inner vector
            idx_inn = 0;
        }
        return *this;
    }
}

/** Defines equality between two iterators */
bool operator==(BookIter book_iter)
{
    return (ptr_vec == book_iter.ptr_vec) &&
    (idx_out == book_iter.idx_out) &&
    (idx_inn == book_iter.idx_inn);
}

/** Defines inequality between two iterators */
bool operator!=(BookIter book_iter)
{
    return !(*this == book_iter);
}

/** Dereference operator */
std::string operator*()
{
    return (*ptr_vec)[idx_out][idx_inn];
}

private:
    friend class Book;

    //Private constructor that can be accessed by the Book class.
    BookIter(book_ptr ptrvec,
            int idxout,
            int idxinn)
        : ptr_vec(ptrvec),
          idx_out(idxout),
          idx_inn(idxinn)
    {
```

```
        }
        book_ptr ptr_vec = nullptr;
        int idx_out = 0;
        int idx_inn = 0;
}; //end BookIter

/** This class represents a book as a vector of vector of strings,
where each vector of strings is sentence in the book. */
class Book
{

    std::vector<std::string> temp_book_;
    std::vector<std::vector<std::string>> book_;

    public:
    //Constructor for the book class.
    //Takes in a .txt file and splits it into a
    //std::vector<std::vector<string>>
    Book(std::string filename)
    {
        //Read in the book as a vector of strings
        //where each string is a sentence.
        std::string sent;
        std::ifstream book_file(filename);
        std::vector<std::string> temp;

        //Read the file line by line.
        while(std::getline(book_file, sent))
        {
            temp_book_.push_back(sent);
        }
        book_file.close();

        //Split each sentence by ' '
        for (unsigned int i = 0; i< temp_book_.size(); i++)
        {
            //Skip empty lines
            if (temp_book_[i].length()==0){
                continue;
            }
            else{
                temp = split(temp_book_[i], ' ');
                book_.push_back(temp);
            }
        }
    }

    /** return an iterator pointing at the start of the book */
    BookIter begin()
    {
        return BookIter(&book_,0,0);
```

```
    }

    /** return an iterator pointing at the end of the book (pass-the-end) */
    BookIter end()
    {
        return BookIter(&book_,(unsigned)book_.size(),0);
    }

};//end Book

/**
 * A functor that compares two strings based on length of strings
 * its operator returns true iff the first string shorter than the second one.
 */
class compare_string_length {
    public:
        bool operator()(std::string str1, std::string str2) const{
            return str1.size() < str2.size();
        }
};

//testing
int main()
{
    //Read in the book
    Book moby_dick("moby_dick.txt");

    //find the longest word in the book.
    std::string longest_word;
    compare_string_length my_functor;
    longest_word =
     *std::max_element(moby_dick.begin(),moby_dick.end(),my_functor);

    //print to command line
    std::cout<< longest_word << std::endl;
    return 0;
}
```