```cpp
//lab 11

// Generic.cpp
#include <algorithm>
#include <numeric>
#include <utility>
#include <vector>
#include <string>
#include <iostream>
#include <cmath>
#include "Generic.hpp"

using namespace std;

bool Compare(const std::pair<string, int> &a, const std::pair<string, int> &b){
  if((a.second >= 600 && b.second >= 600) || (a.second < 600 && b.second < 600)){
    return a.first < b.first;
  }
  else if(a.second >= 600 && b.second < 600){
    return true;
  }
  else{
    return false;
  }
}

void PassOrFail(vector<pair<string, int>> &v){
  std::stable_sort(v.begin(), v.end(), Compare);
}

void ShiftRange(vector<int> &v, int left, int right){
  std::stable_sort(v.begin(),v.end(), [left,right](int a,int b){
    if((a>= left && a<=right) && (b>= left && b<=right)){
      return a<b;
    }
    else if(!(a>= left && a<=right) && !(b>= left && b<=right)){
      return a<b;
    }
    else if((a>= left && a<=right) && !(b>= left && b<=right)){
      return false;
    }
    return true;
  });
}

vector<int> Fibonacci(int n){
  std::vector<int> vec(n);
  std::iota(vec.begin(), vec.end(), 0);
  std::transform(vec.begin(), vec.end(), vec.begin(), [&vec](int i){
    if (i == 0) return 1;
    if (i == 1) return 1;
    return vec[i - 1] + vec[i - 2];
  });
  return vec;
}


int BinaryToInt(const string &binary_str){
  return std::accumulate(binary_str.begin(), binary_str.end(), 0, [](int acc, char bit){
    return acc * 2 + (bit - '0');
  });
}

//generic.hpp
#pragma once
#include <iostream>
#include <string>
#include <utility>
#include <vector>

void PassOrFail(std::vector<std::pair<std::string, int>> &v);

void ShiftRange(std::vector<int> &v, int left, int right);

std::vector<int> Fibonacci(int n);

int BinaryToInt(const std::string &binary_str);


//lab 9

// matrix

#pragma once

#include <vector>
#include <iostream>

template <typename T>
class Matrix {
```

```
private:
  using row_t = std::vector<T>;
  std::vector< row_t > matrix;

public:
  Matrix() { }  // default constructor to make and empty matrix

  // initialization constructor specifying size, and (optional) default value
  Matrix(size_t rows, size_t cols, T init_val=T{})
    : matrix(rows, std::vector<T>(cols, init_val))
  { }

  Matrix(const Matrix<T> &) = default;  // copy constructor to make default work (rule of 5)

  Matrix<T>& operator=(const Matrix<T> &) = default; // and operator method

  // size accessors allow us to retrieve the width or height attributes
  size_t GetWidth() const {   // const value to avoid memory overloading
    if (matrix.empty()) {
      return 0;
    } else {
      return matrix[0].size();
    }
  }
  size_t GetHeight() const { return matrix.size(); }  // initialization constructor for size method, rule of 3 implied from (5 earlier)

  // data accessors allow us to access values at specific rows and columns
  T & GetValue(size_t row, size_t col) { return matrix[row][col]; }
  T GetValue(size_t row, size_t col) const { return matrix[row][col]; }

  // IsSameSize MEMBER FUNCTION
  template <typename P>
  bool IsSameSize(const Matrix<P>& other) const {
    return GetHeight() == other.GetHeight() && GetWidth() == other.GetWidth();
  }

  // operator[] MEMBER FUNCTION
  row_t& operator[](size_t index) {
    return matrix[index];
  }

  const row_t& operator[](size_t index) const {
    return matrix[index];
  }

  // Write MEMBER FUNCTION
  void Write(std::ostream& os) const {
    for (const auto& row : matrix) {
      os << "{";
      for (const auto& val : row) {
        os << val << ",";
      }
      os << "},";
    }
  }

  // operator+ MEMBER FUNCTION
  Matrix<T> operator+(const Matrix<T>& other) const {
    if (!IsSameSize(other)) {
      return Matrix<T>();
    }

    size_t rows = GetHeight();
    size_t cols = GetWidth();

    Matrix<T> result(rows, cols);

    for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) {
        result[i][j] = matrix[i][j] + other[i][j];
      }
    }

    return result;
  }

  // operator* MEMBER FUNCTION (Matrix * scalar)
  Matrix<T> operator*(const T& scalar) const {
    size_t rows = GetHeight();
    size_t cols = GetWidth();

    Matrix<T> result(rows, cols);

    for (size_t i = 0; i < rows; ++i) {
      for (size_t j = 0; j < cols; ++j) {
        result[i][j] = matrix[i][j] * scalar;
      }
    }

    return result;
```

```
  }

  // template <typename T>
  // Matrix<T> operator**(Matrix<T>& other) {
  //     size_t rows_a = GetHeight();
  //     size_t cols_a = GetWidth();

  //     size_t rows_b = other.GetHeight();
  //     size_t cols_b = other.GetWidth();

  //     if (cols_a != rows_b) {
  //         // Return an empty matrix if dimensions don't allow multiplication
  //         return Matrix<T>();
  //     }

  //     Matrix<T> result(rows_a, cols_b);

  //     for (size_t i = 0; i < rows_a; ++i) {
  //         for (size_t j = 0; j < cols_b; ++j) {
  //             T sum = T();  // Initialize to zero for summation
  //             for (size_t k = 0; k < cols_a; ++k) {
  //                 sum += (*this)[i][k] * other[k][j];
  //             }
  //             result[i][j] = sum;
  //         }
  //     }

  //     return result;
  // }


};

// operator<< REGULAR FUNCTION
template <typename T>
std::ostream& operator<<(std::ostream& os, const Matrix<T>& mat) {
  mat.Write(os);
  return os;
}

// Non-member operator* function to handle scalar * Matrix
template <typename T>
Matrix<T> operator*(const T& scalar, const Matrix<T>& mat) {
  return mat * scalar;
}

//lab 7
// table.cpp
#include "Table.hpp"
#include <random>
#include <vector>
#include <string>
#include <iostream>

using namespace std;

Table::Table(size_t width, size_t height, int val)
// PLACE A ':' HERE FOLLOWED BY ANY MEMBER VARIABLE INITIALIZIONS
{
  width_ = width;
  height_ = height;
  row_t row_vec (width_ , val);
  vector<row_t> two_d_vec (height_, row_vec);
  table_ = two_d_vec;
}

// WRITE THE DEFINITION FOR Table::PrintTable
void Table::PrintTable(std::ostream & out) const{
  // vector<vector<int>> two_d_vec;
  // two_d_vec = Table(width_, height_, val);
  for(size_t i = 0; i < table_.size(); ++i){
    for(size_t j = 0; j < table_.at(i).size(); ++j){
      out << table_.at(i).at(j) << ", ";
    }
    out << endl;
  }
}

// // WRITE THE DEFINITION FOR Table::FillRandom
void Table::FillRandom(int low, int high, int seed){
  // std::random_device device;
  std::mt19937 generator(seed);
  std::uniform_int_distribution<int> distribution(low, high);

  // when we need the random value, we use distribution(generator)
  for(size_t i = 0; i < table_.size(); ++i){
    for(size_t j = 0; j < table_.at(i).size(); ++j){
      table_.at(i).at(j) = distribution(generator);
    }
  }
```

```
}


// // WRITE THE DEFINITION FOR Table::SetValue
bool Table::SetValue(size_t col, size_t row, int val){
  if(((col) < width_  && static_cast<int>(col) >= 0) && ((row) < height_ && static_cast<int>(row) >= 0 )){
    table_.at(row).at(col) = val;
    return true;
  }else{
    //throw std::out_of_range("Gone bro");
    return false;
  }
}


// // WRITE THE DEFINITION FOR Table::GetValue
int Table::GetValue(size_t col, size_t row) const{
  if(((col) < width_  && static_cast<int>(col) >= 0) && ((row) < height_ && static_cast<int>(row) >= 0 )){
    return table_.at(row).at(col);
  }else{
    throw std::out_of_range("This is not in my table!");
  }
}

//table.hpp
#pragma once

#include <iostream>
#include <vector>

// A Table of integer values.
class Table {
 private:
  using row_t = std::vector<int>;  // Each row is a regular vector of int.
  std::vector<row_t> table_;       // Table is a vector of rows.
  size_t width_;                   // How wide is table_ (how many columns)?
  size_t height_;                  // How high is table_ (how many rows)?

 public:
  // table will be width x height, default val is 0
  Table(size_t width, size_t height, int val = 0);

  // Pre-made member functions to access the width and height variables.
  size_t GetWidth() const { return width_; }
  size_t GetHeight() const { return height_; }

  // Function to print the tables's contents to a provided output stream.
  void PrintTable(std::ostream &) const;

  // range from low to high, seed has default of 0
  void FillRandom(int low, int high, int seed = 0);

  // Accessors to get and set values.
  bool SetValue(size_t col, size_t row, int val);
  int GetValue(size_t col, size_t row) const;
};
```