Listing 1: SLL.hpp

```
#pragma once

#include <doctest.h>
#include <iostream>
#include <stdexcept>
#include <sstream>

/*---------------------------------------------------------------------------
 * class definition
 *---------------------------------------------------------------------------*/

/**
 * @brief CMP 246 Module 3 generic singly-linked list.
 *
 * SLL is a generic singly-linked list data structure. It
 * allows inserting at the front of the list, and supports index-based
 * get, set, and remove operations. The list also provides a contains method,
 * and the administrative methods clear, isEmpty, and size. SLL also has a copy
 * constructor, and overrides the assignment and stream insertion operators.
 */
template <class T> class SLL {

private:
    /**
     * @brief Node in the singly-linked list.
     *
     * Node is a private inner class of SLL. The class represents a
     * single node in the list. Each node has a payload of type T and a
     * pointer to the next node in the list.
     */
    class Node {
    public:
        /**
         * @brief Default constructor.
         *
         * Make a new Node with default data and next pointer set to zero.
         */
        Node() : data(), pNext(0) { }

        /**
         * @brief Initializing constructor.
         *
         * Make a new node with the specified data and next pointer values.
         *
         * @param d Data value for the node.
         * @param pN Pointer to the next node in the list, or 0 if this is the
         * last Node in the list.
         */
        Node(const T &d, Node *pN) : data(d), pNext(pN) { }

        /**
         * @brief Node payload.
         *
         * Type T payload of the node. Assumed to support assignment, equality
         * testing, copy constructor, and stream insertion.
         */
        T data;

        /**
         * @brief Next node pointer.
         *
         * Pointer to the next node in the list, or 0 if this is the last node.
         */
        Node *pNext;
    };

public:
    /**
     * @brief Default list constructor.
     *
     * Made an initially empty list.
     */
    SLL() : pHead(0), n(0u) { }

    /**
     * @brief Copy construstor.
     *
     * Make a new, deep-copy list, just like the parameter list.
     *
     * @param otherList Reference to the SLL to copy.
     */
    SLL(const SLL<T> &otherList) : pHead(0), n(0u) { copy(otherList); }

    /**
     * @brief Destructor.
     *
     * Free the memory used by this list.
     */
    ~SLL();

    /**
     * @brief Add a value to the front of the list.
     *
     * @param d Value to add to the list.
     */
    void add(const T &d);

    /**
     * @brief Clear the list.
     *
     * Remove all the elements from the list.
     */
    void clear();

    /**
     * @brief Search the list for a specified value.
     *
     * Searches for a value and returns the index of the first occurrence
     * of the value in the list, or -1 if the value is not in the list.
     *
     * @param d Value to search for.
     *
     * @return Index of the first occurrence of d in the list, or -1 if it is
     * not in the list.
     */
    int contains(const T &d) const;

    /**
     * @brief Get a value.
     *
     * Get the value at a specified index in the list.
     *
     * @param idx Index of the value to get.
     *
     * @throws std::out_of_range if the index is past the end of the list.
     *
     * @return Value at location idx in the list.
     */
    T get (unsigned idx) const;

    /**
     * @brief Determine if the list is empty.
     *
     * Convenience method to test if the list contains no elements.
     *
     * @return true if the list is empty, false otherwise.
     */
    bool isEmpty() const { return size() == 0u; }

    /**
     * @brief Remove an element.
     *
     * Remove the value at a specified index in the list.
     *
     * @param idx Index of the element to remove.
     *
     * @throws std::out_of_range if the index is past the end of the list.
     *
     * @return Value that was at location idx.
     */
    T remove(unsigned idx);

    /**
     * @brief Change a list element.
     *
     * Change the value at a specified index to another value.
     *
     * @param idx Index of the value to change.
     *
     * @throws std::out_of_range if the index is past the end of the list.
     *
     * @param d New value to place in position idx.
     */
    void set(unsigned idx, const T &d);

    /**
     * @brief Get list size.
     *
     * Get the number of integers in the list.
     *
     * @return The number of integers in the list.
     */
    unsigned size() const { return n; }

    /**
     * @brief Assignment operator.
     *
     * Override of the assignment operator to work with SLL objects. Makes
     * this list a deep-copy, identical structure as the parameter SLL.
     *
     * @param list SLL to copy from
     *
     * @return Reference to this object.
     */
    SLL<T> &operator=(const SLL<T> &otherList);

    /**
     * @brief Stream insertion operator.
     *
     * Override of the stream insertion operator to work with SLL objects.
     * Outputs each element of the list, in bracketed, comma-separated format,
     * to the provided std::ostream. This function is a friend of the SLL<T>
     * class.
     *
     * @param out std::ostream reference to write the list contents to.
     *
     * @param list SLL reference, with the list to write out.
     *
     * @return A reference to the out std::ostream object.
     */
    template <class T1>
    friend std::ostream &operator<<(std::ostream &out, const SLL<T1> &list);

private:
    /**
     * @brief Copy helper method.
     *
     * This private helper method is used to deep-copy all of the elements from
     * the parameter list to this list. Any existing elements in this list are
     * safely removed before the copy.
     *
     * @param otherList Reference to the SLL object to copy from.
     */
    void copy(const SLL<T> &otherList);

    /**
     * Pointer to the first Node in the list, or 0 if the list is empty.
     */
    Node *pHead;

    /**
     * Number of integers in the list.
     */
    size_t n;
};

```

```cpp
//----------------------------------------------------------------------------
// function implementations
//----------------------------------------------------------------------------

// doctest unit test for the copy constructor
TEST_CASE("testing SLL<T> copy constructor") {
    SLL<int> list1;

    // populate the original list
    for(int i = 0; i < 5; i++) {
        list1.add(i);
    }

    // make a new list like original
    SLL<int> list2(list1);

    // does it have the right size?
    CHECK(list2.size() == list1.size());

    // does it have the right elements?
    for(int i = 0; i < 5; i++) {
        CHECK(list2.get(i) == (4 - i));
    }

    // try it again with dynamic allocation
    SLL<int> *pList = new SLL<int>(list1);

    // does it have the right size?
    CHECK(pList->size() == list1.size());

    // does it have the right elements?
    for(int i = 0; i < 5; i++) {
        CHECK(pList->get(i) == (4 - i));
    }

    delete pList;
}

/*
 * Delete all list nodes when the list is destroyed.
 */
template <class T>
SLL<T>::~SLL() {
    clear();
}

/*
 * Add d to the front of the list.
 */
template <class T>
void SLL<T>::add(const T &d) {
    // create the new node
    Node *pN = new Node(d, pHead);

    // change head pointer to point to the new node
    pHead = pN;

    // increment size
    n++;
}

// doctest unit test for the add method
TEST_CASE("testing SLL<T>::add") {
    SLL<int> list;

    // each addition should happen at the front, and the size should go up by
    // one each time
    list.add(1);
    CHECK(list.get(0) == 1);
    CHECK(list.size() == 1u);

    list.add(2);
    CHECK(list.get(0) == 2);
    CHECK(list.size() == 2u);
}

```

```
/*
 * Delete all list nodes.
 */
template <class T>
void SLL<T>::clear() {
    // create cursors
    Node *pCurr = pHead, *pPrev = 0;

    // iterate thru list, deleting each node
    while(pCurr != 0) {
        // "inchworm" up to next node
        pPrev = pCurr;
        pCurr = pCurr->pNext;

        // delete previous node
        delete pPrev;
    }

    // reset head pointer and size
    pHead = 0;
    n = 0u;
}

// doctest unit test for the clear method
TEST_CASE("testing␣SLL<T>::clear") {
    SLL<int> list;

    // add some list elements
    for(int i = 0; i < 100; i++) {
        list.add(i);
    }

    // clear should make size equal zero
    list.clear();
    CHECK(list.size() == 0u);
}

/*
 * Search the list for value d.
 */
template <class T>
int SLL<T>::contains(const T &d) const {
    // create cursors
    int idx = -1;
    Node *pCurr = pHead;

    // iterate until we find d or end of list
    while(pCurr != 0) {
        idx++;

        // found it? return its index
        if(pCurr->data == d) {
            return idx;
        }

        pCurr = pCurr->pNext;
    }

    // not found? return flag value
    return -1;
}

// doctest unit test for the contains method
TEST_CASE("testing␣SLL<T>::contains") {
    SLL<char> list;

    // populate the list
    for(char c = 'A'; c <= 'Z'; c++) {
        list.add(c);
    }

    // search for 1st element in list
    CHECK(list.contains('Z') == 0);

    // search for last element in list
    CHECK(list.contains('A') == 25);

    // search for something in the middle
    CHECK(list.contains('M') == 13);

    // search for something not in list
    CHECK(list.contains('a') == -1);
}

/*
 * Make this list a deep copy of another list.
 */
template <class T>
void SLL<T>::copy(const SLL<T> &otherList) {
    // remove any existing data
    clear();

    // initialize two cursors: one for this list, one for the other list
    Node *pCurr = pHead, *pOtherCurr = otherList.pHead;

    // iterate through the nodes in the other list
    while(pOtherCurr != 0) {
        // special case: the first node changes the head pointer
        if(pHead == 0) {
            pHead = new Node(pOtherCurr->data, 0);
            pCurr = pHead;
        } else {
            // general case: add new node to end of this list
            pCurr->pNext = new Node(pOtherCurr->data, 0);
            pCurr = pCurr->pNext;
        }

        // move to next node in other list, and increment our size
        pOtherCurr = pOtherCurr->pNext;
        n++;
    }
}

// since copy is private, it's tested indirectly in copy constructor and
// assignment operator tests

/*
 * Get the value at location idx.
 */
template <class T>
T SLL<T>::get(unsigned idx) const {
    // if the idx is past list end, throw an exception
    if(idx >= n) {
        throw std::out_of_range("Index␣out␣of␣range␣in␣SLL::get()");
    }

    // initialize cursor
    Node *pCurr = pHead;

    // iterate cursor to position
    for(unsigned i = 0u; i < idx; i++) {
        pCurr = pCurr->pNext;
    }

    // return requested value
    return pCurr->data;
}

// doctest unit test for the get method
TEST_CASE("testing␣SLL<T>::get") {
    SLL<char> list;

    // populate list
    for(char c = 'A'; c <= 'Z'; c++) {
        list.add(c);
    }

    // get first element
    CHECK(list.get(0) == 'Z');

    // get last element
    CHECK(list.get(25) == 'A');

    // get something in the middle
    CHECK(list.get(13) == 'M');

    // check exception handling when access is beyond list
    bool flag = true;
    try {
        list.get(26); // list element 26 does not exist
        flag = false; // this line should not be reached, due to an exception
    } catch(std::out_of_range oor) {
        // verify flag wasn't modified
        CHECK(flag);
    }
}

/*
 * Remove node at location idx.
 */
template <class T>
T SLL<T>::remove(unsigned idx) {
    // if the idx is past list end, throw an exception
    if(idx >= n) {
        throw std::out_of_range("Index out of range in SLL::remove()");
    }

    // initialize cursors
    Node *pCurr = pHead, *pPrev = 0;

    // iterate cursors to position
    for(unsigned i = 0u; i < idx; i++) {
        pPrev = pCurr;
        pCurr = pCurr->pNext;
    }

    // save value so we can return it
    T d = pCurr->data;

    // first element? change head pointer
    if(pCurr == pHead) {
        pHead = pCurr->pNext;
    } else {
        // general case: "wire around" node
        pPrev->pNext = pCurr->pNext;
    }

    // remove node and decrement size
    delete pCurr;
    n--;

    // send back removed value
    return d;
}

// doctest unit test for the remove method
TEST_CASE("testing SLL<T>::remove") {
    SLL<char> list;

    // populate list
    for(char c = 'A'; c <= 'Z'; c++) {
        list.add(c);
    }

    // remove first element
    CHECK(list.remove(0) == 'Z');
    CHECK(list.size() == 25);
    CHECK(list.get(0) == 'Y');

    // remove last element
    CHECK(list.remove(24) == 'A');
    CHECK(list.size() == 24);
    CHECK(list.get(23) == 'B');

    // remove something in the middle
    CHECK(list.remove(12) == 'M');
    CHECK(list.size() == 23);
    CHECK(list.get(12) == 'L');

    // check exception handling when access is beyond end of the list
    bool flag = true;
    try {
        list.remove(26);    // illegal access; element 26 doesn't exist
        flag = false;       // this line should not be reached due to exception
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Change the value at location idx to d.
 */
template <class T>
void SLL<T>::set(unsigned idx, const T &d) {
    // if the idx is past list end, throw an exception
    if(idx >= n) {
        throw std::out_of_range("Index␣out␣of␣range␣in␣SLL::set()");
    }

    // initialize cursor
    Node *pCurr = pHead;

    // iterate to location
    for(unsigned i = 0u; i < idx; i++) {
        pCurr = pCurr->pNext;
    }

    // change data in location idx to d
    pCurr->data = d;
}

// doctest unit test for the set method
TEST_CASE("testing␣SLL<T>::set") {
    SLL<char> list;

    // populate the list
    for(char c = 'A'; c <= 'Z'; c++) {
        list.add(c);
    }

    // set first element
    list.set(0, 'z');
    CHECK(list.get(0) == 'z');

    // set last element
    list.set(25, 'a');
    CHECK(list.get(25) == 'a');

    // set something in the middle
    list.set(13, 'm');
    CHECK(list.get(13) == 'm');

    // check exception handling for index beyond end of list
    bool flag = true;
    try {
        list.set(26, 'X');  // this is illegal; index doesn't exist
        flag = false;       // this should never be reached, due to the exception
    } catch(std::out_of_range oor) {
        CHECK(flag);    // if exception was handled properly, should be true
    }
}

/*
 * Assignment operator.
 */
template <class T>
SLL<T> & SLL<T>::operator=(const SLL<T> &otherList) {
    // copy other list contents to this object
    copy(otherList);

    return *this;
}

// doctest unit test for the assignment operator
```

```
TEST_CASE("testing␣SLL<T>␣assignment") {
    SLL<int> list1, list2, list3;

    // populate lists
    for(int i = 0; i < 5; i++) {
        list1.add(i);
        if(i % 2 == 0) {
            list2.add(i);
        }
    }

    // do the assignment
    list1 = list2;

    // right size?
    CHECK(list1.size() == list2.size());

    // same contents?
    for(unsigned i = 0; i < list1.size(); i++) {
        CHECK(list1.get(i) == list2.get(i));
    }

    // test chained assignments
    list2.clear();
    list1.clear();
    for(unsigned i = 0; i < 5; i++) {
        list1.add(i);
    }
    list2 = list3 = list1;

    // right size?
    CHECK(list1.size() == list2.size());
    CHECK(list2.size() == list3.size());

    // same contents?
    for(unsigned i = 0; i < list1.size(); i++) {
        CHECK(list1.get(i) == list2.get(i));
        CHECK(list1.get(i) == list3.get(i));
    }

    // check deep copies
    for(unsigned i = 0; i < list1.size(); i++) {
        list2.set(i, i + 1);
        list3.set(i, i + 3);
    }

    // different contents?
    for(unsigned i = 0; i < list1.size(); i++) {
        CHECK(list1.get(i) != list2.get(i));
        CHECK(list1.get(i) != list3.get(i));
    }
}

/*
 * Override of stream insertion operator.
 */
template <class T>
std::ostream &operator<<(std::ostream &out, const SLL<T> &list) {
    out << "[";

    // initialize a cursor to the head of the list
    typename SLL<T>::Node *pCurr = list.pHead;

    // iterate until the end
    while (pCurr != 0) {
        out << pCurr->data;

        // output no comma for last element
        if(pCurr->pNext != 0) {
            out << ",␣";
        }

        // update cursor
        pCurr = pCurr->pNext;
    }

    out << "]";

    return out;
}


// doctest unit test for the stream insertion operator
TEST_CASE("testing␣SLL<T>␣stream␣insertion") {
    SLL<int> list;

    for(int i = 0; i < 5; i++) {
        list.add(i);
    }

    // test stream insertion by "printing" to a string
    std::ostringstream oss;

    oss << list;

    // did the output match?
    CHECK(oss.str() == "[4,␣3,␣2,␣1,␣0]");
}
```

Listing 2: Movie.h

```
#pragma once

#include <doctest.h>
#include <iostream>
#include <string>
#include "../1-SLL/SLL.hpp"

/**
 * @brief CMP Module 3 class representing a movie.
 *
 * Simple class representing a movie in a movie database. The movie has fields
 * for the title of the movie, the year of its release, and a list of keywords
 * describing the movie. The class has overrides for stream insertion, and
 * the equality operator, which compares a movie object to a keyword string
 * and returns true if the keyword is in the movie's keyword list.
 */
class Movie {
public:
    /**
     * @brief Initializing constructor.
     *
     * @param inTitle String containing the title of the movie.
     *
     * @param inYear Integer containing the year the movie was released.
     *
     * @param inKeywords SLL of strings holding the keywords for this movie.
     */
    Movie(std::string inTitle, int inYear, const SLL<std::string> &inKeywords)
        : title(inTitle), year(inYear), keywords(inKeywords) { }

    /**
     * Friend function override of stream insertion. Prints the name of the
     * movie and the year it was released.
     *
     * @param out Reference to the std::ostream to write to.
     *
     * @param mov Movie object to print.
     *
     * @return The std::ostream object.
     */
    friend std::ostream &operator<<(std::ostream &out, const Movie &mov);

    /**
     * Equality operator override.
     *
     * @param keyword std::string containing a keyword.
     *
     * @return true if this object's keyword list contains the parameter,
     * false otherwise.
     */
    bool operator==(std::string keyword);

private:
    /** Movie's title */
    std::string title;

    /** Year movie was released */
    int year;

    /** List of keyword strings for the movie */
    SLL<std::string> keywords;
};
```

Listing 3: Movie.cpp

```
#include "Movie.h"

//-------------------------------------------------------------------------------
// Movie class function implementations
//-------------------------------------------------------------------------------

/*
 * Stream insertion friend function.
 */
std::ostream &operator<<(std::ostream &out, const Movie &mov) {
    out << mov.title << "␣(" << mov.year << ")";
    // out << mov.keywords;
    return out;
}

/*
 * Equality operator override.
 */
bool Movie::operator==(std::string keyword) {
    return keywords.contains(keyword) != -1;
}
```

Listing 4: KeywordSearch.cpp

```
#include <cstdio>
#include <cstdlib>
#include <doctest.h>
#include <fstream>
#include <iostream>
#include <set>
#include <string>
#include "Movie.h"
#include "../1-SLL/SLL.hpp"

/**
 * Function to create a list of movie objects from a text file.
 *
 * @param filename String containing the name of the file to read from.
 *
 * @return SLL<Movie> object encapsulating all of the data in the file.
 */
SLL<Movie> makeMovieList(std::string filename) {

    std::ifstream inFile(filename);
    std::string title, keyword;
    int year;
    SLL<Movie> movieList;

    while(!inFile.eof()) {
        std::getline(inFile, title);

        inFile >> year;

        // there are a variable number of keywords per movie
        SLL<std::string> keywords;
        std::getline(inFile, keyword);
        // read until a sentinel value is detected
        while(keyword != "XXX") {
            keywords.add(keyword);
            std::getline(inFile, keyword);
        }

        // build Movie object based on the data, add to the return list
        movieList.add(Movie(title, year, keywords));
    }
    inFile.close();

    return movieList;
}

/**
 * Function to create a list of keyword strings from a text file.
 *
 * @param filename String containing the name of the file to read from.
 *
 * @return SLL<std::string> object encapsulating all of the keywords in the
 * input file.
 */
SLL<std::string> makeKeywordList(std::string filename) {
    // a STL set is used to elminate duplicates and sort the keywords
    // alphabetically
    std::set<std::string> keywordSet;

    std::ifstream inFile(filename);
    std::string title, keyword;
    int year;

    // read all the data
    while(!inFile.eof()) {
        // title and year are read, but not used here
        std::getline(inFile, title);
        inFile >> year;

        // variable number of keywords per movie
        std::getline(inFile, keyword);
        // read until sentinel value is encountered
        while(keyword != "XXX") {
            keywordSet.insert(keyword);
            std::getline(inFile, keyword);
        }
    }
    inFile.close();

    // Once we have the set, place its values into the SLL
    SLL<std::string> keywordList;
    // These iterators move through the set backwards, so when
    // added to the SLL the final list will be alpha A to Z
    auto itr = keywordSet.rbegin();
    while(itr != keywordSet.rend()) {
        keywordList.add(*itr);
        ++itr;
    }

    // Remove phantom empty string keyword
    keywordList.remove(0);

    return keywordList;
}

/**
 * Function to build a list of all movies with a specified keyword.
 *
 * @param movies Reference to a SLL<Movie> object containing the movies to
 * search
 *
 * @param keyword String keyword to search for
 *
 * @return SLL<Movie> object will all movies that have the specified keyword
 */
SLL<Movie> findKeywordMatches(const SLL<Movie> &movies, std::string keyword) {
    SLL<Movie> matches;

    for(size_t i = 0; i < movies.size(); i++) {
        Movie m = movies.get(i);
        if(m == keyword) {
            matches.add(m);
        }
    }

    return matches;
}

/**
 * Show the menu on standard output.
 */
void showMenu() {
    std::cout << "\nSelect␣from␣one␣of␣the␣following␣options:" << std::endl;
    std::cout << "\t1)␣Display␣the␣movies␣in␣our␣database" << std::endl;
    std::cout << "\t2)␣Display␣a␣list␣of␣possible␣keywords" << std::endl;
    std::cout << "\t3)␣Perform␣a␣keyword␣search␣of␣the␣database" << std::endl;
    std::cout << "\t9)␣Exit␣the␣application" << std::endl;
    std::cout << "Enter␣selection:␣";
}

/**
 * Display the menu and get the user's selection. Only allows valid inputs.
 */
int getSelection() {
    int choice = -1;

    while(!(choice == 1 || choice == 2 || choice == 3 || choice == 9)) {
        showMenu();
        std::cin >> choice;
    }

    return choice;
}

/**
 * Main method for the Module 3 keyword search application.
 */
int main() {
    // build movie and keyword "databases"
    SLL<Movie> movieList = makeMovieList("movies.txt");
    SLL<std::string> keywordList = makeKeywordList("movies.txt");

    std::cout << "Welcome␣to␣our␣movie␣search␣program!" << std::endl;

    // main menu loop
    int choice = -1;
    while(choice != 9) {
        choice = getSelection();

        switch (choice)
        {
        case 1:
            // option 1: display all of the movies
            for(size_t i = 0; i < movieList.size(); i++) {
                std::cout << "\t" << movieList.get(i) << std::endl;
            }
            break;

        case 2:
            // option 2: display all of the keywords; they are shown in
            // columnar format to make the list shorter
            for(size_t i = 0; i < keywordList.size(); i += 3) {
                printf("%26s␣␣%26s␣␣%26s\n", keywordList.get(i).c_str(),
                    keywordList.get(i + 1).c_str(),
                    keywordList.get(i + 2).c_str());
            }
            break;
        case 3:
            // option 3: get a keyword, search for matches, print results
            std::string keyword;
            std::getline(std::cin, keyword); // eat extra endline
            std::cout << "Enter␣keyword:␣";
            std::getline(std::cin, keyword);
            SLL<Movie> matches = findKeywordMatches(movieList, keyword);
            std::cout << "Movies␣matching␣" << keyword << std::endl;

            if(matches.isEmpty()) {
                std::cout << "\tNONE" << std::endl;
            } else {
                for(size_t i = 0; i < matches.size(); i++) {
                    std::cout << "\t" << matches.get(i) << std::endl;
                }
            }
            break;
        }
    }

    return EXIT_SUCCESS;
}
```