Listing 1: SLL.hpp

```
1  #pragma once
2
3  #include <doctest.h>
4  #include <iostream>
5  #include <stdexcept>
6  #include <sstream>
7
8  /*-----------------------------------------------------------------------------
9   * class definition
10  *----------------------------------------------------------------------------*/
11
12 /**
13  * @brief CMP 246 Module 3 generic singly-linked list.
14  *
15  * SLL is a generic singly-linked list data structure. It
16  * allows inserting at the front of the list, and supports index-based
17  * get, set, and remove operations. The list also provides a contains method,
18  * and the administrative methods clear, isEmpty, and size. SLL also has a copy
19  * constructor, and overrides the assignment and stream insertion operators.
20  */
21 template <class T> class SLL {
22
23 private:
24     /**
25      * @brief Node in the singly-linked list.
26      *
27      * Node is a private inner class of SLL. The class represents a
28      * single node in the list. Each node has a payload of type T and a
29      * pointer to the next node in the list.
30      */
31     class Node {
32     public:
33         /**
34          * @brief Default constructor.
35          *
36          * Make a new Node with default data and next pointer set to zero.
37          */
38         Node() : data(), pNext(0) { }
39
40         /**
41          * @brief Initializing constructor.
42          *
43          * Make a new node with the specified data and next pointer values.
44          *
45          * @param d Data value for the node.
46          * @param pN Pointer to the next node in the list, or 0 if this is the
47          * last Node in the list.
48          */
49         Node(const T &d, Node *pN) : data(d), pNext(pN) { }
50
51         /**
52          * @brief Node payload.
53          *
54          * Type T payload of the node. Assumed to support assignment, equality
55          * testing, copy constructor, and stream insertion.
56          */
57         T data;
58
59         /**
```

```
60          * @brief Next node pointer.
61          *
62          * Pointer to the next node in the list, or 0 if this is the last node.
63          */
64         Node *pNext;
65     };
66
67 public:
68     /**
69      * @brief Default list constructor.
70      *
71      * Made an initially empty list.
72      */
73     SLL() : pHead(0), n(0u) { }
74
75     /**
76      * @brief Copy construstor.
77      *
78      * Make a new, deep-copy list, just like the parameter list.
79      *
80      * @param otherList Reference to the SLL to copy.
81      */
82     SLL(const SLL<T> &otherList) : pHead(0), n(0u) { copy(otherList); }
83
84     /**
85      * @brief Destructor.
86      *
87      * Free the memory used by this list.
88      */
89     ~SLL();
90
91     /**
92      * @brief Add a value to the front of the list.
93      *
94      * @param d Value to add to the list.
95      */
96     void add(const T &d);
97
98     /**
99      * @brief Clear the list.
100     *
101     * Remove all the elements from the list.
102     */
103    void clear();
104
105    /**
106     * @brief Search the list for a specified value.
107     *
108     * Searches for a value and returns the index of the first occurrence
109     * of the value in the list, or -1 if the value is not in the list.
110     *
111     * @param d Value to search for.
112     *
113     * @return Index of the first occurrence of d in the list, or -1 if it is
114     * not in the list.
115     */
116    int contains(const T &d) const;
117
118    /**
119     * @brief Get a value.
```

```
120       *
121       * Get the value at a specified index in the list.
122       *
123       * @param idx Index of the value to get.
124       *
125       * @throws std::out_of_range if the index is past the end of the list.
126       *
127       * @return Value at location idx in the list.
128       */
129      T get (unsigned idx) const;
130
131      /**
132       * @brief Determine if the list is empty.
133       *
134       * Convenience method to test if the list contains no elements.
135       *
136       * @return true if the list is empty, false otherwise.
137       */
138      bool isEmpty() const { return size() == 0u; }
139
140      /**
141       * @brief Remove an element.
142       *
143       * Remove the value at a specified index in the list.
144       *
145       * @param idx Index of the element to remove.
146       *
147       * @throws std::out_of_range if the index is past the end of the list.
148       *
149       * @return Value that was at location idx.
150       */
151      T remove(unsigned idx);
152
153      /**
154       * @brief Change a list element.
155       *
156       * Change the value at a specified index to another value.
157       *
158       * @param idx Index of the value to change.
159       *
160       * @throws std::out_of_range if the index is past the end of the list.
161       *
162       * @param d New value to place in position idx.
163       */
164      void set(unsigned idx, const T &d);
165
166      /**
167       * @brief Get list size.
168       *
169       * Get the number of integers in the list.
170       *
171       * @return The number of integers in the list.
172       */
173      unsigned size() const { return n; }
174
175      /**
176       * @brief Assignment operator.
177       *
178       * Override of the assignment operator to work with SLL objects. Makes
179       * this list a deep-copy, identical structure as the parameter SLL.
```

```
180       *
181       * @param list SLL to copy from
182       *
183       * @return Reference to this object.
184       */
185      SLL<T> &operator=(const SLL<T> &otherList);
186
187      /**
188       * @brief Stream insertion operator.
189       *
190       * Override of the stream insertion operator to work with SLL objects.
191       * Outputs each element of the list, in bracketed, comma-separated format,
192       * to the provided std::ostream. This function is a friend of the SLL<T>
193       * class.
194       *
195       * @param out std::ostream reference to write the list contents to.
196       *
197       * @param list SLL reference, with the list to write out.
198       *
199       * @return A reference to the out std::ostream object.
200       */
201      template <class T1>
202      friend std::ostream &operator<<(std::ostream &out, const SLL<T1> &list);
203
204  private:
205      /**
206       * @brief Copy helper method.
207       *
208       * This private helper method is used to deep-copy all of the elements from
209       * the parameter list to this list. Any existing elements in this list are
210       * safely removed before the copy.
211       *
212       * @param otherList Reference to the SLL object to copy from.
213       */
214      void copy(const SLL<T> &otherList);
215
216      /**
217       * Pointer to the first Node in the list, or 0 if the list is empty.
218       */
219      Node *pHead;
220
221      /**
222       * Number of integers in the list.
223       */
224      size_t n;
225  };
226
227  //-----------------------------------------------------------------------------
228  // function implementations
229  //-----------------------------------------------------------------------------
230
231  // doctest unit test for the copy constructor
232  TEST_CASE("testing␣SLL<T>␣copy␣constructor") {
233      SLL<int> list1;
234
235      // populate the original list
236      for(int i = 0; i < 5; i++) {
237          list1.add(i);
238      }
239
```

```
240     // make a new list like original
241     SLL<int> list2(list1);
242 
243     // does it have the right size?
244     CHECK(list2.size() == list1.size());
245 
246     // does it have the right elements?
247     for(int i = 0; i < 5; i++) {
248         CHECK(list2.get(i) == (4 - i));
249     }
250 
251     // try it again with dynamic allocation
252     SLL<int> *pList = new SLL<int>(list1);
253 
254     // does it have the right size?
255     CHECK(pList->size() == list1.size());
256 
257     // does it have the right elements?
258     for(int i = 0; i < 5; i++) {
259         CHECK(pList->get(i) == (4 - i));
260     }
261 
262     delete pList;
263 }
264 
265 /*
266  * Delete all list nodes when the list is destroyed.
267  */
268 template <class T>
269 SLL<T>::~SLL() {
270     clear();
271 }
272 
273 /*
274  * Add d to the front of the list.
275  */
276 template <class T>
277 void SLL<T>::add(const T &d) {
278     // create the new node
279     Node *pN = new Node(d, pHead);
280 
281     // change head pointer to point to the new node
282     pHead = pN;
283 
284     // increment size
285     n++;
286 }
287 
288 // doctest unit test for the add method
289 TEST_CASE("testing␣SLL<T>::add") {
290     SLL<int> list;
291 
292     // each addition should happen at the front, and the size should go up by
293     // one each time
294     list.add(1);
295     CHECK(list.get(0) == 1);
296     CHECK(list.size() == 1u);
297 
298     list.add(2);
299     CHECK(list.get(0) == 2);
```

```
300     CHECK(list.size() == 2u);
301 }
302 
303 /*
304  * Delete all list nodes.
305  */
306 template <class T>
307 void SLL<T>::clear() {
308     // create cursors
309     Node *pCurr = pHead, *pPrev = 0;
310 
311     // iterate thru list, deleting each node
312     while(pCurr != 0) {
313         // "inchworm" up to next node
314         pPrev = pCurr;
315         pCurr = pCurr->pNext;
316 
317         // delete previous node
318         delete pPrev;
319     }
320 
321     // reset head pointer and size
322     pHead = 0;
323     n = 0u;
324 }
325 
326 // doctest unit test for the clear method
327 TEST_CASE("testing␣SLL<T>::clear") {
328     SLL<int> list;
329 
330     // add some list elements
331     for(int i = 0; i < 100; i++) {
332         list.add(i);
333     }
334 
335     // clear should make size equal zero
336     list.clear();
337     CHECK(list.size() == 0u);
338 }
339 
340 /*
341  * Search the list for value d.
342  */
343 template <class T>
344 int SLL<T>::contains(const T &d) const {
345     // create cursors
346     int idx = -1;
347     Node *pCurr = pHead;
348 
349     // iterate until we find d or end of list
350     while(pCurr != 0) {
351         idx++;
352 
353         // found it? return its index
354         if(pCurr->data == d) {
355             return idx;
356         }
357 
358         pCurr = pCurr->pNext;
359     }
```

```
360
361     // not found? return flag value
362     return -1;
363 }
364
365 // doctest unit test for the contains method
366 TEST_CASE("testing SLL<T>::contains") {
367     SLL<char> list;
368
369     // populate the list
370     for(char c = 'A'; c <= 'Z'; c++) {
371         list.add(c);
372     }
373
374     // search for 1st element in list
375     CHECK(list.contains('Z') == 0);
376
377     // search for last element in list
378     CHECK(list.contains('A') == 25);
379
380     // search for something in the middle
381     CHECK(list.contains('M') == 13);
382
383     // search for something not in list
384     CHECK(list.contains('a') == -1);
385 }
386
387 /*
388  * Make this list a deep copy of another list.
389  */
390 template <class T>
391 void SLL<T>::copy(const SLL<T> &otherList) {
392     // remove any existing data
393     clear();
394
395     // initialize two cursors: one for this list, one for the other list
396     Node *pCurr = pHead, *pOtherCurr = otherList.pHead;
397
398     // iterate through the nodes in the other list
399     while(pOtherCurr != 0) {
400         // special case: the first node changes the head pointer
401         if(pHead == 0) {
402             pHead = new Node(pOtherCurr->data, 0);
403             pCurr = pHead;
404         } else {
405             // general case: add new node to end of this list
406             pCurr->pNext = new Node(pOtherCurr->data, 0);
407             pCurr = pCurr->pNext;
408         }
409
410         // move to next node in other list, and increment our size
411         pOtherCurr = pOtherCurr->pNext;
412         n++;
413     }
414 }
415
416 // since copy is private, it's tested indirectly in copy constructor and
417 // assignment operator tests
418
419 /*
```

```
420  * Get the value at location idx.
421  */
422 template <class T>
423 T SLL<T>::get(unsigned idx) const {
424     // if the idx is past list end, throw an exception
425     if(idx >= n) {
426         throw std::out_of_range("Index␣out␣of␣range␣in␣SLL::get()");
427     }
428 
429     // initialize cursor
430     Node *pCurr = pHead;
431 
432     // iterate cursor to position
433     for(unsigned i = 0u; i < idx; i++) {
434         pCurr = pCurr->pNext;
435     }
436 
437     // return requested value
438     return pCurr->data;
439 }
440 
441 // doctest unit test for the get method
442 TEST_CASE("testing␣SLL<T>::get") {
443     SLL<char> list;
444 
445     // populate list
446     for(char c = 'A'; c <= 'Z'; c++) {
447         list.add(c);
448     }
449 
450     // get first element
451     CHECK(list.get(0) == 'Z');
452 
453     // get last element
454     CHECK(list.get(25) == 'A');
455 
456     // get something in the middle
457     CHECK(list.get(13) == 'M');
458 
459     // check exception handling when access is beyond list
460     bool flag = true;
461     try {
462         list.get(26); // list element 26 does not exist
463         flag = false; // this line should not be reached, due to an exception
464     } catch(std::out_of_range oor) {
465         // verify flag wasn't modified
466         CHECK(flag);
467     }
468 }
469 
470 /*
471  * Remove node at location idx.
472  */
473 template <class T>
474 T SLL<T>::remove(unsigned idx) {
475     // if the idx is past list end, throw an exception
476     if(idx >= n) {
477         throw std::out_of_range("Index␣out␣of␣range␣in␣SLL::remove()");
478     }
479 
```

```
480     // initialize cursors
481     Node *pCurr = pHead, *pPrev = 0;
482 
483     // iterate cursors to position
484     for(unsigned i = 0u; i < idx; i++) {
485         pPrev = pCurr;
486         pCurr = pCurr->pNext;
487     }
488 
489     // save value so we can return it
490     T d = pCurr->data;
491 
492     // first element? change head pointer
493     if(pCurr == pHead) {
494         pHead = pCurr->pNext;
495     } else {
496         // general case: "wire around" node
497         pPrev->pNext = pCurr->pNext;
498     }
499 
500     // remove node and decrement size
501     delete pCurr;
502     n--;
503 
504     // send back removed value
505     return d;
506 }
507 
508 // doctest unit test for the remove method
509 TEST_CASE("testing␣SLL<T>::remove") {
510     SLL<char> list;
511 
512     // populate list
513     for(char c = 'A'; c <= 'Z'; c++) {
514         list.add(c);
515     }
516 
517     // remove first element
518     CHECK(list.remove(0) == 'Z');
519     CHECK(list.size() == 25);
520     CHECK(list.get(0) == 'Y');
521 
522     // remove last element
523     CHECK(list.remove(24) == 'A');
524     CHECK(list.size() == 24);
525     CHECK(list.get(23) == 'B');
526 
527     // remove something in the middle
528     CHECK(list.remove(12) == 'M');
529     CHECK(list.size() == 23);
530     CHECK(list.get(12) == 'L');
531 
532     // check exception handling when access is beyond end of the list
533     bool flag = true;
534     try {
535         list.remove(26);    // illegal access; element 26 doesn't exist
536         flag = false;       // this line should not be reached due to exception
537     } catch(std::out_of_range oor) {
538         CHECK(flag);
539     }
```

```
}

/*
 * 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) {
```

```
600     // copy other list contents to this object
601     copy(otherList);
602 
603     return *this;
604 }
605 
606 // doctest unit test for the assignment operator
607 TEST_CASE("testing␣SLL<T>␣assignment") {
608     SLL<int> list1, list2, list3;
609 
610     // populate lists
611     for(int i = 0; i < 5; i++) {
612         list1.add(i);
613         if(i % 2 == 0) {
614             list2.add(i);
615         }
616     }
617 
618     // do the assignment
619     list1 = list2;
620 
621     // right size?
622     CHECK(list1.size() == list2.size());
623 
624     // same contents?
625     for(unsigned i = 0; i < list1.size(); i++) {
626         CHECK(list1.get(i) == list2.get(i));
627     }
628 
629     // test chained assignments
630     list2.clear();
631     list1.clear();
632     for(unsigned i = 0; i < 5; i++) {
633         list1.add(i);
634     }
635     list2 = list3 = list1;
636 
637     // right size?
638     CHECK(list1.size() == list2.size());
639     CHECK(list2.size() == list3.size());
640 
641     // same contents?
642     for(unsigned i = 0; i < list1.size(); i++) {
643         CHECK(list1.get(i) == list2.get(i));
644         CHECK(list1.get(i) == list3.get(i));
645     }
646 
647     // check deep copies
648     for(unsigned i = 0; i < list1.size(); i++) {
649         list2.set(i, i + 1);
650         list3.set(i, i + 3);
651     }
652 
653     // different contents?
654     for(unsigned i = 0; i < list1.size(); i++) {
655         CHECK(list1.get(i) != list2.get(i));
656         CHECK(list1.get(i) != list3.get(i));
657     }
658 }
659 
```

```
660 /*
661  * Override of stream insertion operator.
662  */
663 template <class T>
664 std::ostream &operator<<(std::ostream &out, const SLL<T> &list) {
665     out << "[";
666 
667     // initialize a cursor to the head of the list
668     typename SLL<T>::Node *pCurr = list.pHead;
669 
670     // iterate until the end
671     while (pCurr != 0) {
672         out << pCurr->data;
673 
674         // output no comma for last element
675         if(pCurr->pNext != 0) {
676             out << ",␣";
677         }
678 
679         // update cursor
680         pCurr = pCurr->pNext;
681     }
682 
683     out << "]";
684 
685     return out;
686 }
687 
688 
689 // doctest unit test for the stream insertion operator
690 TEST_CASE("testing␣SLL<T>␣stream␣insertion") {
691     SLL<int> list;
692 
693     for(int i = 0; i < 5; i++) {
694         list.add(i);
695     }
696 
697     // test stream insertion by "printing" to a string
698     std::ostringstream oss;
699 
700     oss << list;
701 
702     // did the output match?
703     CHECK(oss.str() == "[4,␣3,␣2,␣1,␣0]");
704 }
```

Listing 2: Movie.h

```
1  #pragma once
2  
3  #include <doctest.h>
4  #include <iostream>
5  #include <string>
6  #include "../1-SLL/SLL.hpp"
7  
8  /**
9   * @brief CMP Module 3 class representing a movie.
10  *
11  * Simple class representing a movie in a movie database. The movie has fields
12  * 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();

```

```
44      return movieList;
45  }
46
47  /**
48   * Function to create a list of keyword strings from a text file.
49   *
50   * @param filename String containing the name of the file to read from.
51   *
52   * @return SLL<std::string> object encapsulating all of the keywords in the
53   * input file.
54   */
55  SLL<std::string> makeKeywordList(std::string filename) {
56      // a STL set is used to elminate duplicates and sort the keywords
57      // alphabetically
58      std::set<std::string> keywordSet;
59
60      std::ifstream inFile(filename);
61      std::string title, keyword;
62      int year;
63
64      // read all the data
65      while(!inFile.eof()) {
66          // title and year are read, but not used here
67          std::getline(inFile, title);
68          inFile >> year;
69
70          // variable number of keywords per movie
71          std::getline(inFile, keyword);
72          // read until sentinel value is encountered
73          while(keyword != "XXX") {
74              keywordSet.insert(keyword);
75              std::getline(inFile, keyword);
76          }
77      }
78      inFile.close();
79
80      // Once we have the set, place its values into the SLL
81      SLL<std::string> keywordList;
82      // These iterators move through the set backwards, so when
83      // added to the SLL the final list will be alpha A to Z
84      auto itr = keywordSet.rbegin();
85      while(itr != keywordSet.rend()) {
86          keywordList.add(*itr);
87          ++itr;
88      }
89
90      // Remove phantom empty string keyword
91      keywordList.remove(0);
92
93      return keywordList;
94  }
95
96  /**
97   * Function to build a list of all movies with a specified keyword.
98   *
99   * @param movies Reference to a SLL<Movie> object containing the movies to
100  * search
101  *
102  * @param keyword String keyword to search for
103  *
```

```
104  * @return SLL<Movie> object will all movies that have the specified keyword
105  */
106 SLL<Movie> findKeywordMatches(const SLL<Movie> &movies, std::string keyword) {
107     SLL<Movie> matches;
108 
109     for(size_t i = 0; i < movies.size(); i++) {
110         Movie m = movies.get(i);
111         if(m == keyword) {
112             matches.add(m);
113         }
114     }
115 
116     return matches;
117 }
118 
119 /**
120  * Show the menu on standard output.
121  */
122 void showMenu() {
123     std::cout << "\nSelect␣from␣one␣of␣the␣following␣options:" << std::endl;
124     std::cout << "\t1)␣Display␣the␣movies␣in␣our␣database" << std::endl;
125     std::cout << "\t2)␣Display␣a␣list␣of␣possible␣keywords" << std::endl;
126     std::cout << "\t3)␣Perform␣a␣keyword␣search␣of␣the␣database" << std::endl;
127     std::cout << "\t9)␣Exit␣the␣application" << std::endl;
128     std::cout << "Enter␣selection:␣";
129 }
130 
131 /**
132  * Display the menu and get the user's selection. Only allows valid inputs.
133  */
134 int getSelection() {
135     int choice = -1;
136 
137     while(!(choice == 1 || choice == 2 || choice == 3 || choice == 9)) {
138         showMenu();
139         std::cin >> choice;
140     }
141 
142     return choice;
143 }
144 
145 /**
146  * Main method for the Module 3 keyword search application.
147  */
148 int main() {
149     // build movie and keyword "databases"
150     SLL<Movie> movieList = makeMovieList("movies.txt");
151     SLL<std::string> keywordList = makeKeywordList("movies.txt");
152 
153     std::cout << "Welcome␣to␣our␣movie␣search␣program!" << std::endl;
154 
155     // main menu loop
156     int choice = -1;
157     while(choice != 9) {
158         choice = getSelection();
159 
160         switch (choice)
161         {
162         case 1:
163             // option 1: display all of the movies
```

```
164             for(size_t i = 0; i < movieList.size(); i++) {
165                 std::cout << "\t" << movieList.get(i) << std::endl;
166             }
167             break;
168 
169         case 2:
170             // option 2: display all of the keywords; they are shown in
171             // columnar format to make the list shorter
172             for(size_t i = 0; i < keywordList.size(); i += 3) {
173                 printf("%26s  %26s  %26s\n", keywordList.get(i).c_str(),
174                     keywordList.get(i + 1).c_str(),
175                     keywordList.get(i + 2).c_str());
176             }
177             break;
178         case 3:
179             // option 3: get a keyword, search for matches, print results
180             std::string keyword;
181             std::getline(std::cin, keyword); // eat extra endline
182             std::cout << "Enter keyword: ";
183             std::getline(std::cin, keyword);
184             SLL<Movie> matches = findKeywordMatches(movieList, keyword);
185             std::cout << "Movies matching " << keyword << std::endl;
186 
187             if(matches.isEmpty()) {
188                 std::cout << "\tNONE" << std::endl;
189             } else {
190                 for(size_t i = 0; i < matches.size(); i++) {
191                     std::cout << "\t" << matches.get(i) << std::endl;
192                 }
193             }
194             break;
195         }
196     }
197 
198     return EXIT_SUCCESS;
199 }
```