Listing 1: SimpleSLL.hpp

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

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

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

```
180     size_t n;
181 };
182 
183 //-----------------------------------------------------------------------------
184 // function implementations
185 //-----------------------------------------------------------------------------
186 
187 /*
188  * Add d to the front of the list.
189  */
190 template <class T>
191 void SimpleSLL<T>::add(const T &d) {
192     // create the new node
193     Node *pN = new Node(d, pHead);
194 
195     // change head pointer to point to the new node
196     pHead = pN;
197 
198     // increment size
199     n++;
200 }
201 
202 // doctest unit test for the add method
203 TEST_CASE("testing␣SimpleSLL<T>::add") {
204     SimpleSLL<int> list;
205 
206     // each addition should happen at the front, and the size should go up by
207     // one each time
208     list.add(1);
209     CHECK(list.get(0) == 1);
210     CHECK(list.size() == 1u);
211 
212     list.add(2);
213     CHECK(list.get(0) == 2);
214     CHECK(list.size() == 2u);
215 }
216 
217 /*
218  * Delete all list nodes.
219  */
220 template <class T>
221 void SimpleSLL<T>::clear() {
222     // create cursors
223     Node *pCurr = pHead, *pPrev = 0;
224 
225     // iterate thru list, deleting each node
226     while(pCurr != 0) {
227         // "inchworm" up to next node
228         pPrev = pCurr;
229         pCurr = pCurr->pNext;
230 
231         // delete previous node
232         delete pPrev;
233     }
234 
235     // reset head pointer and size
236     pHead = 0;
237     n = 0u;
238 }
239 
```

```
240 // doctest unit test for the clear method
241 TEST_CASE("testing␣SimpleSLL<T>::clear") {
242     SimpleSLL<int> list;
243 
244     // add some list elements
245     for(int i = 0; i < 100; i++) {
246         list.add(i);
247     }
248 
249     // clear should make size equal zero
250     list.clear();
251     CHECK(list.size() == 0u);
252 }
253 
254 /*
255  * Search the list for value d.
256  */
257 template <class T>
258 int SimpleSLL<T>::contains(const T &d) const {
259     // create cursors
260     int idx = -1;
261     Node *pCurr = pHead;
262 
263     // iterate until we find d or end of list
264     while(pCurr != 0) {
265         idx++;
266 
267         // found it? return its index
268         if(pCurr->data == d) {
269             return idx;
270         }
271 
272         pCurr = pCurr->pNext;
273     }
274 
275     // not found? return flag value
276     return -1;
277 }
278 
279 // doctest unit test for the contains method
280 TEST_CASE("testing␣SimpleSLL<T>::contains") {
281     SimpleSLL<char> list;
282 
283     // populate the list
284     for(char c = 'A'; c <= 'Z'; c++) {
285         list.add(c);
286     }
287 
288     // search for 1st element in list
289     CHECK(list.contains('Z') == 0);
290 
291     // search for last element in list
292     CHECK(list.contains('A') == 25);
293 
294     // search for something in the middle
295     CHECK(list.contains('M') == 13);
296 
297     // search for something not in list
298     CHECK(list.contains('a') == -1);
299 }
```

```
300
301 /*
302  * Get the value at location idx.
303  */
304 template <class T>
305 T SimpleSLL<T>::get(size_t idx) const {
306     // if the idx is past list end, throw an exception
307     if(idx >= n) {
308         throw std::out_of_range("Index out of range in SimpleSLL::get()");
309     }
310
311     // initialize cursor
312     Node *pCurr = pHead;
313
314     // iterate cursor to position
315     for(size_t i = 0u; i < idx; i++) {
316         pCurr = pCurr->pNext;
317     }
318
319     // return requested value
320     return pCurr->data;
321 }
322
323 // doctest unit test for the get method
324 TEST_CASE("testing SimpleSLL<T>::get") {
325     SimpleSLL<char> list;
326
327     // populate list
328     for(char c = 'A'; c <= 'Z'; c++) {
329         list.add(c);
330     }
331
332     // get first element
333     CHECK(list.get(0) == 'Z');
334
335     // get last element
336     CHECK(list.get(25) == 'A');
337
338     // get something in the middle
339     CHECK(list.get(13) == 'M');
340
341     // check exception handling when access is beyond list
342     bool flag = true;
343     try {
344         list.get(26); // list element 26 does not exist
345         flag = false; // this line should not be reached, due to an exception
346     } catch(std::out_of_range oor) {
347         // verify flag wasn't modified
348         CHECK(flag);
349     }
350 }
351
352 /*
353  * Remove node at location idx.
354  */
355 template <class T>
356 T SimpleSLL<T>::remove(size_t idx) {
357     // if the idx is past list end, throw an exception
358     if(idx >= n) {
359         throw std::out_of_range("Index out of range in SimpleSLL::remove()");
```

```
360     }
361
362     // initialize cursors
363     Node *pCurr = pHead, *pPrev = 0;
364
365     // iterate cursors to position
366     for(size_t i = 0u; i < idx; i++) {
367         pPrev = pCurr;
368         pCurr = pCurr->pNext;
369     }
370
371     // save value so we can return it
372     T d = pCurr->data;
373
374     // first element? change head pointer
375     if(pCurr == pHead) {
376         pHead = pCurr->pNext;
377     } else {
378         // general case: "wire around" node
379         pPrev->pNext = pCurr->pNext;
380     }
381
382     // remove node and decrement size
383     delete pCurr;
384     n--;
385
386     // send back removed value
387     return d;
388 }
389
390 // doctest unit test for the remove method
391 TEST_CASE("testing␣SimpleSLL<T>::remove") {
392     SimpleSLL<char> list;
393
394     // populate list
395     for(char c = 'A'; c <= 'Z'; c++) {
396         list.add(c);
397     }
398
399     // remove first element
400     CHECK(list.remove(0) == 'Z');
401     CHECK(list.size() == 25);
402     CHECK(list.get(0) == 'Y');
403
404     // remove last element
405     CHECK(list.remove(24) == 'A');
406     CHECK(list.size() == 24);
407     CHECK(list.get(23) == 'B');
408
409     // remove something in the middle
410     CHECK(list.remove(12) == 'M');
411     CHECK(list.size() == 23);
412     CHECK(list.get(12) == 'L');
413
414     // check exception handling when access is beyond end of the list
415     bool flag = true;
416     try {
417         list.remove(26);    // illegal access; element 26 doesn't exist
418         flag = false;       // this line should not be reached due to exception
419     } catch(std::out_of_range oor) {
```

```
420         CHECK(flag);
421     }
422 }
423 
424 /*
425  * Print the list to standard output.
426  */
427 template <class T>
428 void SimpleSLL<T>::print() const {
429     using namespace std;
430 
431     cout << "[";
432 
433     // initialize cursor
434     Node *pCurr = pHead;
435 
436     // iterate through list
437     while(pCurr != 0) {
438         cout << pCurr->data;
439 
440         // no comma for last node
441         if(pCurr->pNext != 0) {
442             cout << ",␣";
443         }
444 
445         pCurr = pCurr->pNext;
446     }
447 
448     cout << "]" << endl;
449 }
450 
451 /*
452  * Change the value at location idx to d.
453  */
454 template <class T>
455 void SimpleSLL<T>::set(size_t idx, const T &d) {
456     // if the idx is past list end, throw an exception
457     if(idx >= n) {
458         throw std::out_of_range("Index␣out␣of␣range␣in␣SimpleSLL::set()");
459     }
460 
461     // initialize cursor
462     Node *pCurr = pHead;
463 
464     // iterate to location
465     for(size_t i = 0u; i < idx; i++) {
466         pCurr = pCurr->pNext;
467     }
468 
469     // change data in location idx to d
470     pCurr->data = d;
471 }
472 
473 // doctest unit test for the set method
474 TEST_CASE("testing␣SimpleSLL<T>::set") {
475     SimpleSLL<char> list;
476 
477     // populate the list
478     for(char c = 'A'; c <= 'Z'; c++) {
479         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
    }
}
```

Listing 2: User.h

```
#pragma once

#include <iostream>
#include <string>

/**
 * @brief CMP 246 Module 2 class representing an online user.
 *
 * Simple class holding the name and password of a fictional online user.
 * Has a copy constructor and overrides assignment (=), equality testing (==),
 * and stream insertion (<<).
 */
class User {
public:
    /**
     * @brief Default constructor.
     *
     * Builds a default user, with empty-string name and password fields.
     */
    User() : name(""), password("") { }

    /**
     * @brief Initializing constructor.
     *
     * Builds a user with the specified name and password.
     *
     * @param n std::string containing the user's username.
     *
     * @param p std::string containing the user's password.
     */
    User(std::string n, std::string p) : name(n), password(p) { }

    /**
     * @brief Copy constructor.
```

```
35        *
36        * Builds a user just like another user.
37        *
38        * @param other User object to copy name and password from.
39        */
40       User(const User &other);
41
42       /**
43        * @brief Name accessor.
44        *
45        * Get this user's name.
46        *
47        * @return std::string containing this user's name.
48        */
49       std::string getName() { return name; }
50
51       /**
52        * @brief Password accessor.
53        *
54        * Get this user's password.
55        *
56        * @return std::string containing this user's password.
57        */
58       std::string getPassword() { return password; }
59
60       /**
61        * @brief Assignment operator.
62        *
63        * Overridden assignment operator, allowing safe assignment of one User
64        * object to another.
65        *
66        * @param other Reference to the user to copy name and password from.
67        *
68        * @return Reference to this object.
69        */
70       User& operator=(const User &other);
71
72       /**
73        * @brief Equality operator.
74        *
75        * Overridden equality testing operator, for comparing two User objects.
76        *
77        * @param other Reference to the other user to compare against.
78        *
79        * @return true if this user has same name and password as the other user,
80        * false otherwise.
81        */
82       bool operator==(const User &other);
83
84       /**
85        * @brief Stream insertion operator.
86        *
87        * Overridden stream insertion operator, defined as a friend to the User
88        * class.
89        *
90        * @param out Reference to the output stream to write to.
91        *
92        * @param user Reference to the User object to write to the output stream.
93        *
94        * @return Reference to the output stream.
```

```
95       */
96      friend std::ostream& operator<<(std::ostream &out, const User &user) {
97          out << user.name << "␣(" << user.password << ")";
98          return out;
99      }
100
101 private:
102     /** Name of the user. */
103     std::string name;
104     /** Password of the user. */
105     std::string password;
106 };
```

Listing 3: User.cpp

```
1  #include "User.h"
2
3  // function definitions for the User class.
4
5  // copy constructor
6  User::User(const User &other) {
7      name = other.name;
8      password = other.password;
9  }
10
11 // assignment operator
12 User& User::operator=(const User &other) {
13     name = other.name;
14     password = other.password;
15
16     return *this;
17 }
18
19 // equality operator
20 bool User::operator==(const User &other) {
21     return name == other.name && password == other.password;
22 }
```

Listing 4: UserAuth.cpp

```
1  #include <cstdlib>
2  #include <fstream>
3  #include <iostream>
4  #include "../1-SimpleSLL/SimpleSLL.hpp"
5  #include "User.h"
6
7  /**
8   * @brief CMP 246 Module 2 main program to exercise the SimpleSLL class.
9   *
10  * This program loads the username / password pairs from 'users.txt' into a
11  * SimpleSLL of User objects. Then, the program prompts for a username and
12  * password, and checks to see if that pair is in the list -- i.e., if the
13  * user has been authenticated or not.
14  */
15 int main() {
16     // read users.txt into a list of User objects
17     SimpleSLL<User> userList;
18
19     std::ifstream inFile("users.txt");
20     std::string name, password;
```

```
21      while(inFile >> name) {
22          inFile >> password;
23          userList.add(User(name, password));
24      }
25
26      // prompt for username and password, then authenticate
27      std::cout << "Enter␣username␣(q␣to␣quit):␣";
28      std::cin >> name;
29      while(name != "q") {
30          std::cout << "Enter␣password:␣";
31          std::cin >> password;
32
33          User u(name, password);
34
35          // username and password correct?
36          if(userList.contains(u) != -1) {
37              std::cout << "WELCOME␣TO␣OUR␣SITE!" << std::endl;
38              std::cout << "...logged␣off." << std::endl;
39          } else {
40              std::cout << "ACCESS␣DENIED." << std::endl;
41          }
42
43          // prompt for next username
44          std::cout << "Enter␣username␣(q␣to␣quit):␣";
45          std::cin >> name;
46      }
47
48      return EXIT_SUCCESS;
49  }
```