Listing 1: DLL.hpp

```
#pragma once

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

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

/**
 * @brief CMP 246 Module 5 generic doubly-linked list, supporting iterators.
 *
 * DLL is a generic doubly-linked list data structure. It allows inserting at
 * the front or back 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. DLL also has a copy constructor, and
 * overrides the assignment and stream insertion operators. DLL provides front,
 * back, and end methods to access iterators that move through the list from
 * front to back or back to front.
 */
template <class T> class DLL {

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

        /**
         * @brief Initializing constructor.
         *
         * Make a new node with the specified data and previous and next
         * pointer values.
         *
         * @param d Data value for the node.
         * @param pP Pointer to the previous node in the list, or 0 if this is
         * the first node in the list.
         * @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 *pP, Node *pN) : data(d), pPrev(pP), 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 Previous node pointer.
         *
         * Pointer to the previous node in the list, or 0 if this is the first
         * node.
         */
        Node *pPrev;

        /**
```

```
75          * @brief Next node pointer.
76          *
77          * Pointer to the next node in the list, or 0 if this is the last node.
78          */
79         Node *pNext;
80     };
81
82 public:
83     /**
84      * @brief DLL iterator.
85      *
86      * This class allows DLL users to iterate through the list, from front to
87      * back or back to front, without exposing the pointer structure of the
88      * list, and without incurring the time complexity of successive calls to
89      * the index-based get() and set() methods.
90      */
91     class Iterator {
92     public:
93         /**
94          * @brief Iterator dereferencing operator.
95          *
96          * This override of the dereferencing operator allows DLL
97          * users to access and / or change the payload of a node in the list.
98          *
99          * @throws std::out_of_range if the iterator is past either end of
100         * the list.
101         */
102        T &operator*();
103
104        /**
105         * @brief Iterator equality operator.
106         *
107         * This override of the equality operator allows DLL users to
108         * compare two iterators, to determine if they refer to the same node
109         * in the list.
110         */
111        bool operator==(const Iterator &other) const {
112            return pCurr == other.pCurr;
113        }
114
115        /**
116         * @brief Interator inequality operator.
117         *
118         * This override of the inequality operator allows DLL users to
119         * compare two iterators, to determine if they refer to different nodes
120         * in the list.
121         */
122        bool operator!=(const Iterator &other) const {
123            return pCurr != other.pCurr;
124        }
125
126        /**
127         * @brief Iterator decrement operator.
128         *
129         * This override of the postfix decrement operator allows DLL
130         * users to move an iterator from one node to the previous node in the
131         * list.
132         *
133         * @param dummy Dummy parameter that indicates we are overriding
134         * the postfix decrement operator.
135         *
136         * @throws std::out_of_range if the iterator is past either end of
137         * the list.
138         */
139        Iterator &operator--(int dummy);
140
141        /**
142         * @brief Iterator increment operator.
143         *
144         * This override of the postfix increment operator allows DLL
145         * users to move an iterator from one node to the next node in the
146         * list.
147         *
148         * @param dummy Dummy parameter that indicates we are overriding
149         * the postfix increment operator.
150         *
```

```
151             * @throws std::out_of_range if the iterator is past either end of
152             * the list.
153             */
154            Iterator &operator++(int dummy);
155
156            // Make DLL a friend class, so it can access the private constructor
157            friend class DLL;
158
159        private:
160            /**
161             * @brief Initializing constructor.
162             *
163             * This constructor makes an iterator that refers to the Node at the
164             * end of the specified pointer. The constructor is private, so that
165             * the only way to get an iterator is via the DLL front(), back(),
166             * and end() methods.
167             *
168             * @param pC Node this iterator should refer to.
169             */
170            Iterator(Node *pC) : pCurr(pC) { }
171
172            /**
173             * Pointer to the Node this iterator refers to.
174             */
175            Node *pCurr;
176        };
177
178        /**
179         * @brief Default list constructor.
180         *
181         * Made an initially empty list.
182         */
183        DLL() : pHead(0), pTail(0), n(0u) { }
184
185        /**
186         * @brief Copy construstor.
187         *
188         * Make a new, deep-copy list, just like the parameter list.
189         *
190         * @param otherList Reference to the DLL to copy.
191         */
192        DLL(const DLL<T> &otherList) : pHead(0), pTail(0), n(0u) { copy(otherList); }
193
194        /**
195         * @brief Destructor.
196         *
197         * Free the memory used by this list.
198         */
199        ~DLL() { clear(); }
200
201        /**
202         * @brief Add a value to the front of the list.
203         *
204         * @param d Value to add to the list.
205         */
206        void addFirst(const T &d);
207
208        /**
209         * @brief Add a value to the back of the list.
210         *
211         * @param d Value to add to the list.
212         */
213        void addLast(const T &d);
214
215        /**
216         * @brief Get an Iterator on the last node of the list.
217         *
218         * @return An Iterator positioned on the last node of the list.
219         */
220        Iterator back() const { return Iterator(pTail); }
221
222        /**
223         * @brief Clear the list.
224         *
225         * Remove all the elements from the list.
226         */
```

```
227     void clear();
228
229     /**
230      * @brief Search the list for a specified value.
231      *
232      * Searches for a value and returns the index of the first occurrence
233      * of the value in the list, or -1 if the value is not in the list.
234      *
235      * @param d Value to search for.
236      *
237      * @return Index of the first occurrence of d in the list, or -1 if it is
238      * not in the list.
239      */
240     int contains(const T &d) const;
241
242     /**
243      * @brief Get an Iterator representing the end of the list.
244      *
245      * @return An Iterator representing one past the back, or one before the
246      * front, of the list.
247      */
248     Iterator end() const { return Iterator(0); }
249
250     /**
251      * @brief Get an Iterator on the first node of the list.
252      *
253      * @return An Iterator positioned on the first node of the list.
254      */
255     Iterator front() const { return Iterator(pHead); }
256
257     /**
258      * @brief Get a value.
259      *
260      * Get the value at a specified index in the list.
261      *
262      * @param idx Index of the value to get.
263      *
264      * @throws std::out_of_range if the index is past either end of the list.
265      *
266      * @return Value at location idx in the list.
267      */
268     T get (unsigned idx) const;
269
270     /**
271      * @brief Get first value.
272      *
273      * Get the value at the front of the list.
274      *
275      * @throws std::out_of_range if the list is empty.
276      *
277      * @return Value at the front of the list.
278      */
279     T getFirst() const;
280
281     /**
282      * @brief Get last value.
283      *
284      * Get the value at the back of the list.
285      *
286      * @throws std::out_of_range if the list is empty.
287      *
288      * @return Value at the back of the list.
289      */
290     T getLast() const;
291
292     /**
293      * @brief Determine if the list is empty.
294      *
295      * Convenience method to test if the list contains no elements.
296      *
297      * @return true if the list is empty, false otherwise.
298      */
299     bool isEmpty() const { return size() == 0u; }
300
301     /**
302      * @brief Remove an element.
```

```
303         *
304         * Remove the value at a specified index in the list.
305         *
306         * @param idx Index of the element to remove.
307         *
308         * @throws std::out_of_range if the index is past either end of the list.
309         *
310         * @return Value that was at location idx.
311         */
312        T remove(unsigned idx);
313
314        /**
315         * @brief Remove first element.
316         *
317         * Remove the element from the front of the list.
318         *
319         * @throws std::out_of_range if the list is empty.
320         *
321         * @return Value that was at the front of the list.
322         */
323        T removeFirst();
324
325        /**
326         * @brief Remove last element.
327         *
328         * Remove the element from the back of the list.
329         *
330         * @throws std::out_of_range if the list is empty.
331         *
332         * @return Value that was at the back of the list.
333         */
334        T removeLast();
335
336        /**
337         * @brief Change a list element.
338         *
339         * Change the value at a specified index to another value.
340         *
341         * @param idx Index of the value to change.
342         *
343         * @throws std::out_of_range if the index is past either end of the list.
344         *
345         * @param d New value to place in position idx.
346         */
347        void set(unsigned idx, const T &d);
348
349        /**
350         * @brief Change the first list element.
351         *
352         * @param d New value to replace front value in the list.
353         *
354         * @throws std::out_of_range if the list is empty.
355         */
356        void setFirst(const T &d);
357
358        /**
359         * @brief Change the last list element.
360         *
361         * @param d New value to replace back value in the list.
362         *
363         * @throws std::out_of_range if the list is empty.
364         */
365        void setLast(const T &d);
366
367        /**
368         * @brief Get list size.
369         *
370         * Get the number of integers in the list.
371         *
372         * @return The number of integers in the list.
373         */
374        unsigned size() const { return n; }
375
376        /**
377         * @brief Assignment operator.
378         *
```

```
379       * Override of the assignment operator to work with DLL objects.
380       * Makes this list a deep-copy, identical structure as the parameter
381       * DLL.
382       *
383       * @param list DLL to copy from
384       *
385       * @return Reference to this object.
386       */
387      DLL<T> &operator=(const DLL<T> &otherList);
388
389  private:
390      /**
391       * @brief Copy helper method.
392       *
393       * This private helper method is used to deep-copy all of the elements from
394       * the parameter list to this list. Any existing elements in this list are
395       * safely removed before the copy.
396       *
397       * @param otherList Reference to the DLL object to copy from.
398       */
399      void copy(const DLL<T> &otherList);
400
401      /**
402       * Pointer to the first Node in the list, or 0 if the list is empty.
403       */
404      Node *pHead;
405
406      /**
407       * Pointer to the last Node in the list, or 0 if the list is empty.
408       */
409      Node *pTail;
410
411      /**
412       * Number of integers in the list.
413       */
414      size_t n;
415  };
416
417  //-----------------------------------------------------------------------------
418  // function implementations
419  //-----------------------------------------------------------------------------
420
421  /*
422   * Iterator dereferencing operator override.
423   */
424  template <class T>
425  T &DLL<T>::Iterator::operator*() {
426      // if the iterator is past either end of the list, throw an exception
427      if(pCurr == 0) {
428          throw std::out_of_range("Dereferencing␣beyond␣list␣end␣in␣"
429                                  "Iterator::*()");
430      }
431
432      return pCurr->data;
433  }
434
435  // doctest unit test for iterator dereferencing
436  TEST_CASE("testing␣DLL<T>::Iterator␣dereferencing") {
437      DLL<int> list;
438
439      list.addFirst(1);
440      list.addFirst(0);
441      DLL<int>::Iterator it = list.front();
442
443      // first element should be 0
444      CHECK(*it == 0);
445
446      // last element should be 1
447      it++;
448      CHECK(*it == 1);
449
450      // check exception handling when dereferencing past end of list
451      it++;
452      bool flag = true;
453      try {
454          *it;            // should throw an exception
```

```
455         flag = false;   // this should never happen
456     } catch (std::out_of_range oor) {
457         CHECK(flag);
458     }
459 }
460
461 /*
462  * Iterator decrement operator overload.
463  */
464 template <class T>
465 typename DLL<T>::Iterator &DLL<T>::Iterator::operator--(int dummy) {
466     // if the iterator is past the end of the list, throw an exception
467     if(pCurr == 0) {
468         throw std::out_of_range("Increment␣beyond␣list␣end␣in␣"
469                                 "Iterator::--()");
470     }
471
472     pCurr = pCurr->pPrev;
473     return *this;
474 }
475
476 // doctest unit test for iterator decrement overload
477 TEST_CASE("testing␣DLL<T>::Iterator␣postfix␣decrement") {
478     DLL<char> list;
479
480     // populate with a - z
481     for(char c = 'a'; c <= 'z'; c++) {
482         list.addFirst(c);
483     }
484
485     // verify that iterating moves through the list backwards
486     DLL<char>::Iterator it = list.back();
487     for(char c = 'a'; c <= 'z'; c++) {
488         CHECK(*it == c);
489         it--;
490     }
491
492     // check exception handling when incrementing beyond list end
493     bool flag = true;
494     try {
495         it--;           // this should throw an exception
496         flag = false;   // this should never happen
497     } catch(std::out_of_range oor) {
498         CHECK(flag);
499     }
500 }
501
502 /*
503  * Iterator increment operator overload.
504  */
505 template <class T>
506 typename DLL<T>::Iterator &DLL<T>::Iterator::operator++(int dummy) {
507     // if the iterator is past the end of the list, throw an exception
508     if(pCurr == 0) {
509         throw std::out_of_range("Increment␣beyond␣list␣end␣in␣"
510                                 "Iterator::++()");
511     }
512
513     pCurr = pCurr->pNext;
514     return *this;
515 }
516
517 // doctest unit test for iterator increment overload
518 TEST_CASE("testing␣DLL<T>::Iterator␣postfix␣increment") {
519     DLL<char> list;
520
521     // populate with a - z
522     for(char c = 'a'; c <= 'z'; c++) {
523         list.addFirst(c);
524     }
525
526     // verify that iterating moves through the list
527     DLL<char>::Iterator it = list.front();
528     for(char c = 'z'; c >= 'a'; c--) {
529         CHECK(*it == c);
530         it++;
```

```
531     }
532
533     // check exception handling when incrementing beyond list end
534     bool flag = true;
535     try {
536         it++;           // this should throw an exception
537         flag = false;   // this should never happen
538     } catch(std::out_of_range oor) {
539         CHECK(flag);
540     }
541 }
542
543 // doctest unit test for the copy constructor
544 TEST_CASE("testing␣DLL<T>␣copy␣constructor") {
545     DLL<int> list1;
546
547     // populate the original list
548     for(int i = 0; i < 5; i++) {
549         list1.addFirst(i);
550     }
551
552     // make a new list like original
553     DLL<int> list2(list1);
554
555     // does it have the right size?
556     CHECK(list2.size() == list1.size());
557
558     // does it have the right elements?
559     for(int i = 0; i < 5; i++) {
560         CHECK(list2.get(i) == (4 - i));
561     }
562
563     // try it again with dynamic allocation
564     DLL<int> *pList = new DLL<int>(list1);
565
566     // does it have the right size?
567     CHECK(pList->size() == list1.size());
568
569     // does it have the right elements?
570     for(int i = 0; i < 5; i++) {
571         CHECK(pList->get(i) == (4 - i));
572     }
573
574     delete pList;
575 }
576
577 /*
578  * Add d to the front of the list.
579  */
580 template <class T>
581 void DLL<T>::addFirst(const T &d) {
582     Node *pN = new Node(d, 0, pHead);
583
584     if(pHead == 0) {
585         // empty list case
586         pHead = pTail = pN;
587     } else {
588         // non-empty list case
589         pHead = pN;
590         pHead->pNext->pPrev = pHead;
591     }
592
593     n++;
594 }
595
596 // doctest unit test for addFirst
597 TEST_CASE("testing␣DLL<T>::addFirst") {
598     DLL<int> list;
599
600     list.addFirst(0);
601
602     // is it there?
603     DLL<int>::Iterator it = list.front();
604     CHECK(*it == 0);
605
606     // try with another element
```

```
607     list.addFirst(1);
608     it = list.front();
609     CHECK(*it == 1);
610     it = list.back();
611     CHECK(*it == 0);
612 }
613 
614 /*
615  * Add do the back of the list.
616  */
617 template <class T>
618 void DLL<T>::addLast(const T &d) {
619     Node *pN = new Node(d, pTail, 0);
620 
621     if(pHead == 0) {
622         // empty list case
623         pHead = pTail = pN;
624     } else {
625         // non-empty list case
626         pTail = pN;
627         pTail->pPrev->pNext = pTail;
628     }
629 
630     n++;
631 }
632 
633 // doctest unit test for addLast
634 TEST_CASE("testing DLL<T>::addLast") {
635     DLL<int> list;
636 
637     list.addLast(0);
638 
639     // is it there?
640     DLL<int>::Iterator it = list.back();
641     CHECK(*it == 0);
642 
643     // try with another element
644     list.addLast(1);
645     it = list.back();
646     CHECK(*it == 1);
647     it = list.front();
648     CHECK(*it == 0);
649 }
650 
651 // doctest unit test for the back method
652 TEST_CASE("testing DLL<T>::back") {
653     DLL<int> list;
654     list.addLast(0);
655 
656     // is the back iterator the first element?
657     DLL<int>::Iterator it = list.back();
658     CHECK(*it == list.get(0));
659 
660     // add another and repeat the check
661     list.addLast(1);
662     it = list.back();
663     CHECK(*it == list.get(1));
664 }
665 
666 /*
667  * Delete all list nodes.
668  */
669 template <class T>
670 void DLL<T>::clear() {
671     // create cursors
672     Node *pCurr = pHead, *pPrev = 0;
673 
674     // iterate thru list, deleting each node
675     while(pCurr != 0) {
676         // "inchworm" up to next node
677         pPrev = pCurr;
678         pCurr = pCurr->pNext;
679 
680         // delete previous node
681         delete pPrev;
682     }
```

```
683
684     // reset head, tail pointer and size
685     pHead = 0;
686     pTail = 0;
687     n = 0u;
688 }
689
690 // doctest unit test for the clear method
691 TEST_CASE("testing␣DLL<T>::clear") {
692     DLL<int> list;
693
694     // add some list elements
695     for(int i = 0; i < 100; i++) {
696         list.addFirst(i);
697     }
698
699     // clear should make size equal zero
700     list.clear();
701     CHECK(list.size() == 0u);
702 }
703
704 /*
705  * Search the list for value d.
706  */
707 template <class T>
708 int DLL<T>::contains(const T &d) const {
709     // create cursors
710     int idx = -1;
711     Node *pCurr = pHead;
712
713     // iterate until we find d or end of list
714     while(pCurr != 0) {
715         idx++;
716
717         // found it? return its index
718         if(pCurr->data == d) {
719             return idx;
720         }
721
722         pCurr = pCurr->pNext;
723     }
724
725     // not found? return flag value
726     return -1;
727 }
728
729 // doctest unit test for the contains method
730 TEST_CASE("testing␣DLL<T>::contains") {
731     DLL<char> list;
732
733     // populate the list
734     for(char c = 'A'; c <= 'Z'; c++) {
735         list.addFirst(c);
736     }
737
738     // search for 1st element in list
739     CHECK(list.contains('Z') == 0);
740
741     // search for last element in list
742     CHECK(list.contains('A') == 25);
743
744     // search for something in the middle
745     CHECK(list.contains('M') == 13);
746
747     // search for something not in list
748     CHECK(list.contains('a') == -1);
749 }
750
751 /*
752  * Make this list a deep copy of another list.
753  */
754 template <class T>
755 void DLL<T>::copy(const DLL<T> &list) {
756     // remove any existing data
757     clear();
758
```

```
759     // using iterator and addLast simplifies this method compared with the
760     // equivalent in previous SLLs
761     for(DLL<T>::Iterator i = list.front(); i != list.end(); i++) {
762         addLast(*i);
763     }
764 }
765
766 // since copy is private, it's tested indirectly in copy constructor and
767 // assignment operator tests
768
769 // doctest unit test for the end method
770 TEST_CASE("testing DLLT<T>::end") {
771     DLL<double> list;
772
773     // iterating through empty list should not happen
774     DLL<double>::Iterator it = list.front();
775     int count = 0;
776     for(; it != list.end(); it++) {
777         count++;
778     }
779     CHECK(count == 0);
780
781     // iterating through a list w/ 5 elements
782     for(int i = 0; i < 5; i++) {
783         list.addFirst(i);
784     }
785     it = list.front();
786     count = 0;
787     for(; it != list.end(); it++) {
788         count++;
789     }
790     CHECK(count == 5);
791 }
792
793 // doctest unit test for the front method
794 TEST_CASE("testing DLL<T>::front") {
795     DLL<int> list;
796     list.addFirst(0);
797
798     // is the front iterator the first element?
799     DLL<int>::Iterator it = list.front();
800     CHECK(*it == list.get(0));
801
802     // add another and repeat the check
803     list.addFirst(1);
804     it = list.front();
805     CHECK(*it == list.get(0));
806 }
807
808 /*
809  * Get the value at location idx.
810  */
811 template <class T>
812 T DLL<T>::get(unsigned idx) const {
813     // if the idx is past list end, throw an exception
814     if(idx >= n) {
815         throw std::out_of_range("Index out of range in DLL::get()");
816     }
817
818     // initialize cursor
819     Node *pCurr = pHead;
820
821     // iterate cursor to position
822     for(unsigned i = 0u; i < idx; i++) {
823         pCurr = pCurr->pNext;
824     }
825
826     // return requested value
827     return pCurr->data;
828 }
829
830 // doctest unit test for the get method
831 TEST_CASE("testing DLL<T>::get") {
832     DLL<char> list;
833
834     // populate list
```

```
    for(char c = 'A'; c <= 'Z'; c++) {
        list.addFirst(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);
    }
}

/*
 * Get the front value.
 */
template <class T>
T DLL<T>::getFirst() const {
    // if list is empty, throw an exception
    if(pHead == 0) {
        throw std::out_of_range("Empty list in DLL::getFirst()");
    }

    return pHead->data;
}

// doctest unit test for the getFirst method
TEST_CASE("testing DLL<T>::getFirst") {
    DLL<int> list;
    for(int i = 0; i < 5; i++) {
        list.addFirst(i);
        CHECK(list.getFirst() == i);
    }

    // test exception generation
    list.clear();
    bool flag = true;
    try {
        list.getFirst();    // this should cause an exception
        flag = false;       // this should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Get the back value.
 */
template <class T>
T DLL<T>::getLast() const {
    // if list is empty, throw an exception
    if(pTail == 0) {
        throw std::out_of_range("Empty list in DLL::getLast()");
    }

    return pTail->data;
}

// doctest unit test for the getLast method
TEST_CASE("testing DLL<T>::getLast") {
    DLL<int> list;
    for(int i = 0; i < 5; i++) {
        list.addLast(i);
        CHECK(list.getLast() == i);
    }
```

```
911
912     // test exception generation
913     list.clear();
914     bool flag = true;
915     try {
916         list.getLast();     // this should cause an exception
917         flag = false;       // this should never happen
918     } catch(std::out_of_range oor) {
919         CHECK(flag);
920     }
921 }
922
923 /*
924  * Remove node at location idx.
925  */
926 template <class T>
927 T DLL<T>::remove(unsigned idx) {
928     // if the idx is past list end, throw an exception
929     if(idx >= n) {
930         throw std::out_of_range("Index␣out␣of␣range␣in␣DLL::remove()");
931     }
932
933     // handle special cases with other methods
934     if(idx == 0u) {
935         return removeFirst();
936     } else if(idx == n - 1u) {
937         return removeLast();
938     }
939
940     // handle the general case
941     Node *pCurr = pHead;
942
943     // iterate cursor to position
944     for(unsigned i = 0u; i < idx; i++) {
945         pCurr = pCurr->pNext;
946     }
947
948     // save value so we can return it
949     T d = pCurr->data;
950
951     // wire around the node to be removed
952     pCurr->pPrev->pNext = pCurr->pNext;
953     pCurr->pNext->pPrev = pCurr->pPrev;
954
955     // remove node and decrement size
956     delete pCurr;
957     n--;
958
959     // send back removed value
960     return d;
961 }
962
963 // doctest unit test for the remove method
964 TEST_CASE("testing␣DLL<T>::remove") {
965     DLL<char> list;
966
967     // populate list
968     for(char c = 'A'; c <= 'Z'; c++) {
969         list.addFirst(c);
970     }
971
972     // remove first element
973     CHECK(list.remove(0) == 'Z');
974     CHECK(list.size() == 25);
975     CHECK(list.get(0) == 'Y');
976
977     // remove last element
978     CHECK(list.remove(24) == 'A');
979     CHECK(list.size() == 24);
980     CHECK(list.get(23) == 'B');
981
982     // remove something in the middle
983     CHECK(list.remove(12) == 'M');
984     CHECK(list.size() == 23);
985     CHECK(list.get(12) == 'L');
986
```

```
    // 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);
    }
}

/*
 * Remove front element
 */
template <class T>
T DLL<T>::removeFirst() {
    if(pHead == 0) {
        throw std::out_of_range("Empty list in DLL::removeFirst()");
    }

    // save data in front node, and pointer to the node
    T d = pHead->data;
    Node *pTemp = pHead;

    // update head pointer
    pHead = pHead->pNext;
    if(pHead != 0) {
        // if there are more elements, mark new front node prev pointer
        // as left end of the list
        pHead->pPrev = 0;
    } else {
        // if there are no more elements, update tail pointer
        pTail = 0;
    }

    // update size, free former front node memory
    n--;
    delete pTemp;

    // send back value from former front node
    return d;
}

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

    // populate list
    for(int i = 0; i < 10; i++) {
        list.addLast(i);
    }

    // check removing from front
    for(int i = 0; i < 10; i++) {
        CHECK(list.removeFirst() == i);
        CHECK(list.size() == 9 - i);
    }

    // check removing from empty list
    bool flag = true;
    try {
        list.removeFirst();     // list is empty, so this is an error
        flag = false;           // should never happen due to exception
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Remove back element.
 */
template <class T>
T DLL<T>::removeLast() {
    if(pHead == 0) {
        throw std::out_of_range("Empty list in DLL::removeLast()");
    }

```

```
1063      // save data in front node, and pointer to the node
1064      T d = pTail->data;
1065      Node *pTemp = pTail;
1066
1067      // update head pointer
1068      pTail = pTail->pPrev;
1069      if(pTail != 0) {
1070          // if there are more elements, mark new last node next pointer
1071          // as right end of the list
1072          pTail->pNext = 0;
1073      } else {
1074          // if there are no more elements, update head pointer
1075          pHead = 0;
1076      }
1077
1078      // update size, free former last node memory
1079      n--;
1080      delete pTemp;
1081
1082      // send back value from former last node
1083      return d;
1084  }
1085
1086  // doctest unit test for the removeLast method
1087  TEST_CASE("testing␣DLL<T>::removeLast") {
1088      DLL<int> list;
1089
1090      // populate list
1091      for(int i = 0; i < 10; i++) {
1092          list.addFirst(i);
1093      }
1094
1095      // test removeLast
1096      for(int i = 0; i < 10; i++) {
1097          CHECK(list.removeLast() == i);
1098          CHECK(list.size() == 9 - i);
1099      }
1100
1101      // test exception handling
1102      bool flag = true;
1103      try {
1104          list.removeLast();      // should not be legal; list is empty
1105          flag = false;           // should never happen due to exception
1106      } catch(std::out_of_range oor) {
1107          CHECK(flag);
1108      }
1109  }
1110
1111  /*
1112   * Change the value at location idx to d.
1113   */
1114  template <class T>
1115  void DLL<T>::set(unsigned idx, const T &d) {
1116      // if the idx is past list end, throw an exception
1117      if(idx >= n) {
1118          throw std::out_of_range("Index␣out␣␣of␣range␣in␣DLL::set()");
1119      }
1120
1121      // initialize cursor
1122      Node *pCurr = pHead;
1123
1124      // iterate to location
1125      for(unsigned i = 0u; i < idx; i++) {
1126          pCurr = pCurr->pNext;
1127      }
1128
1129      // change data in location idx to d
1130      pCurr->data = d;
1131  }
1132
1133  // doctest unit test for the set method
1134  TEST_CASE("testing␣DLL<T>::set") {
1135      DLL<char> list;
1136
1137      // populate the list
1138      for(char c = 'A'; c <= 'Z'; c++) {
```

```
1139         list.addFirst(c);
1140     }
1141
1142     // set first element
1143     list.set(0, 'z');
1144     CHECK(list.get(0) == 'z');
1145
1146     // set last element
1147     list.set(25, 'a');
1148     CHECK(list.get(25) == 'a');
1149
1150     // set something in the middle
1151     list.set(13, 'm');
1152     CHECK(list.get(13) == 'm');
1153
1154     // check exception handling for index beyond end of list
1155     bool flag = true;
1156     try {
1157         list.set(26, 'X');  // this is illegal; index doesn't exist
1158         flag = false;       // this should never be reached, due to exception
1159     } catch(std::out_of_range oor) {
1160         CHECK(flag);    // if exception was handled properly, should be true
1161     }
1162 }
1163
1164 /*
1165  * Change the value at the front to d.
1166  */
1167 template <class T>
1168 void DLL<T>::setFirst(const T &d) {
1169     // throw an exception if the list is empty
1170     if(isEmpty()) {
1171         throw std::out_of_range("Empty␣list␣in␣DLL<T>::setFirst()");
1172     }
1173
1174     pHead->data = d;
1175 }
1176
1177 // doctest unit test for setFirst
1178 TEST_CASE("testing␣DLL<T>::setFirst") {
1179     DLL<char> list;
1180
1181     // populate list
1182     for(char c = 'a'; c <= 'z'; c++) {
1183         list.addLast(c);
1184     }
1185
1186     // test setFirst
1187     list.setFirst('A');
1188     CHECK(list.getFirst() == 'A');
1189     list.clear();
1190     list.addFirst('A');
1191     list.setFirst('a');
1192     CHECK(list.getLast() == 'a');
1193
1194     // test exception handling
1195     bool flag = true;
1196     list.clear();
1197     try {
1198         list.setFirst('Q');     // should cause an exception
1199         flag = false;           // should never happen
1200     } catch(std::out_of_range oor) {
1201         CHECK(flag);
1202     }
1203 }
1204
1205 /*
1206  * Change the value at the back to d.
1207  */
1208 template <class T>
1209 void DLL<T>::setLast(const T &d) {
1210     // throw an exception if the list is empty
1211     if(isEmpty()) {
1212         throw std::out_of_range("Empty␣list␣in␣DLL<T>::setLast()");
1213     }
1214
```

```
1215      pTail->data = d;
1216  }
1217
1218  // doctest unit test for setLast
1219  TEST_CASE("testing␣DLL<T>::setLast") {
1220      DLL<char> list;
1221
1222      // populate list
1223      for(char c = 'a'; c <= 'z'; c++) {
1224          list.addLast(c);
1225      }
1226
1227      // test setFirst
1228      list.setLast('A');
1229      CHECK(list.getLast() == 'A');
1230      list.clear();
1231      list.addFirst('A');
1232      list.setLast('a');
1233      CHECK(list.getFirst() == 'a');
1234
1235      // test exception handling
1236      bool flag = true;
1237      list.clear();
1238      try {
1239          list.setLast('Q');      // should cause an exception
1240          flag = false;           // should never happen
1241      } catch(std::out_of_range oor) {
1242          CHECK(flag);
1243      }
1244  }
1245
1246  /*
1247   * Assignment operator.
1248   */
1249  template <class T>
1250  DLL<T> & DLL<T>::operator=(const DLL<T> &otherList) {
1251      // remove any existing contents first
1252      clear();
1253
1254      // copy other list contents to this object
1255      copy(otherList);
1256
1257      return *this;
1258  }
1259
1260  // doctest unit test for the assignment operator
1261  TEST_CASE("testing␣DLL<T>␣assignment") {
1262      DLL<int> list1, list2;
1263
1264      // populate lists
1265      for(int i = 0; i < 5; i++) {
1266          list1.addFirst(i);
1267          if(i % 2 == 0) {
1268              list2.addFirst(i);
1269          }
1270      }
1271
1272      // do the assignment
1273      list1 = list2;
1274
1275      // right size?
1276      CHECK(list1.size() == list2.size());
1277
1278      // same contents?
1279      for(unsigned i = 0; i < list1.size(); i++) {
1280          CHECK(list1.get(i) == list2.get(i));
1281      }
1282  }
1283
1284  /*
1285   * Override of the stream insertion operator. Using iterators removes
1286   * the need for this to be a friend of the DLL class.
1287   */
1288  template <class T>
1289  std::ostream &operator<<(std::ostream &out, const DLL<T> &list) {
1290
```

```
    out << "[";

    // iterate through the list using an iterator
    typename DLL<T>::Iterator i = list.front();

    while(i != list.end()) {

        out << *i;

        // output comma for all but last element
        i++;
        if(i != list.end()) {
            out << ", ";
        }
    }

    out << "]";

    return out;
}

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

    for(int i = 0; i < 5; i++) {
        list.addFirst(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: Stack.hpp

```
#pragma once

#include <doctest.h>
#include <iostream>
#include <stdexcept>
#include <sstream>
#include "../1-DLL/DLL.hpp"

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

/**
 * @brief CMP 246 Module 5 generic stack.
 *
 * Stack is a generic stack data structure. It supports push, pop, and peek
 * methods, and clear, isEmpty, and size administrative methods. It also has
 * a copy constructor, and overrides the assignment and stream insertion
 * operators.
 */
template <class T> class Stack {
public:
    /**
     * @brief Default constructor.
     *
     * Make a new, empty stack.
     */
    Stack() {}

    /**
     * @brief Copy constructor.
     *
     * Make a new stack, just like the parameter.
     *
     * @param stack Constant reference to the stack to copy from.
     */
```

```
37      Stack(const Stack<T> &stack) { copy(stack); }
38
39      /**
40       * @brief Empty the stack.
41       *
42       * Removes all elements from this stack.
43       */
44      void clear() { list.clear(); }
45
46      /**
47       * @brief Determine if the stack is empty.
48       *
49       * @return True if the stack is empty, false otherwise.
50       */
51      bool isEmpty() { return list.isEmpty(); }
52
53      /**
54       * @brief Access top element.
55       *
56       * @return The top element of the stack, without
57       * removing the element from the stack.
58       *
59       * @throws std::out_of_range if the stack is empty.
60       */
61      T peek() const;
62
63      /**
64       * @brief Pop top element.
65       *
66       * Removes and returns the top element of the stack.
67       *
68       * @return Element of type T that was at the top of the stack.
69       *
70       * @throws std::out_of_range if the stack is empty.
71       */
72      T pop();
73
74      /**
75       * @brief Push a new element.
76       *
77       * Pushes a value of type T onto the top of the stack.
78       *
79       * @param v Element to push.
80       */
81      void push(const T &v) { list.addFirst(v); }
82
83      /**
84       * @brief Get the size of the stack.
85       *
86       * @return Number of elements in the stack.
87       */
88      unsigned size() { return list.size(); }
89
90      /**
91       * @brief Assignment operator override.
92       *
93       * @param stack Stack to copy from.
94       *
95       * @return Reference to this stack, after it has copied the parameter,
96       * for chaining purposes.
97       */
98      Stack<T> &operator=(const Stack<T> &stack);
99
100     /**
101      * @brief Stream insertion override.
102      *
103      * @param out ostream object to output to, e.g., cout.
104      *
105      * @param stack Stack to output.
106      *
107      * @return Reference to the out ostream object.
108      */
109     friend std::ostream &operator<<(std::ostream &out, const Stack<T> &stack) {
110         out << stack.list;
111         return out;
112     }
```

```
113
114 private:
115     /**
116      * Doubly-linked list used as the underlying data store for the stack.
117      */
118     DLL<T> list;
119
120     /**
121      * @brief Copy stack contents.
122      *
123      * Private helper method for copy constructor and assignment override.
124      *
125      * @param stack Stack to copy from.
126      */
127     void copy(const Stack<T> &stack) { list = stack.list; }
128 };
129
130 //-----------------------------------------------------------------------------
131 // function implementations
132 //-----------------------------------------------------------------------------
133
134 // doctest unit test for the copy constructor
135 TEST_CASE("testing␣Stack<T>::Stack(stack)␣constructor") {
136     Stack<char> source;
137
138     for(char c = 'a'; c <= 'z'; c++) {
139         source.push(c);
140     }
141
142     Stack<char> stackCopy(source);
143
144     // should have same elements, in the same order
145     while(!stackCopy.isEmpty()) {
146         char a = source.pop();
147         char b = stackCopy.pop();
148         CHECK(a == b);
149     }
150
151     // make sure both are empty
152     bool res = source.isEmpty() && stackCopy.isEmpty();
153     CHECK(res);
154 }
155
156 // doctest unit test for the clear method
157 TEST_CASE("testing␣Stack<T>::clear()") {
158     Stack<int> stack;
159
160     for(int i = 0; i < 10; i++) {
161         stack.push(i);
162     }
163
164     // clear and valdiate size is zero
165     stack.clear();
166
167     CHECK(stack.size() == 0u);
168     CHECK(stack.isEmpty());
169 }
170
171 // doctest unit test for the isEmpty method
172 TEST_CASE("testing␣Stack<T>::isEmpty()") {
173     Stack<double> stack;
174
175     // initial one is empty?
176     CHECK(stack.isEmpty());
177
178     // populate
179     for(int i = 0; i < 10; i++) {
180         stack.push(i);
181     }
182
183     // should not be empty
184     CHECK(!stack.isEmpty());
185
186     // clear
187     stack.clear();
188
```

```
189     // should be empty again
190     CHECK(stack.isEmpty());
191 }
192
193 // copy is private, and so it is tested in copy constructor, assignment tests
194
195 /*
196  * Peek function implementation.
197  */
198 template <class T>
199 T Stack<T>::peek() const {
200     if(list.isEmpty()) {
201         throw std::out_of_range("Empty␣stack␣in␣Stack::peek()");
202     }
203
204     return list.getFirst();
205 }
206
207 // doctest unit test for the peek method
208 TEST_CASE("testing␣Stack<T>::peek()") {
209     Stack<char> stack;
210
211     for(char c = 'a'; c <= 'z'; c++) {
212         stack.push(c);
213     }
214
215     // check access to top element
216     char c = stack.peek();
217
218     CHECK(c == 'z');
219
220     stack.pop();
221
222     c = stack.peek();
223     CHECK(c == 'y');
224
225     // check exception handling
226     stack.clear();
227     bool flag = true;
228     try {
229         c = stack.peek();   // should cause an exception
230         flag = false;       // should never happen
231     } catch(std::out_of_range oor) {
232         // flag should still be true
233         CHECK(flag);
234     }
235 }
236
237 /*
238  * Pop function implementation.
239  */
240 template <class T>
241 T Stack<T>::pop() {
242     if(list.isEmpty()) {
243         throw std::out_of_range("Empty␣stack␣in␣Stack::ppo()");
244     }
245
246     return list.removeFirst();
247 }
248
249 // doctest unit test for pop function
250 TEST_CASE("testing␣Stack<T>::pop()") {
251     Stack<int> stack;
252     for(int i = 0; i < 10; i++) {
253         stack.push(i);
254     }
255
256     // check pop always removes top element
257     for(int i = 9; i >= 0; i--) {
258         CHECK(stack.pop() == i);
259         CHECK(stack.size() == 10 - (10 - i));
260     }
261
262     CHECK(stack.isEmpty());
263
264     // check exception handling
```

```
265     bool flag = true;
266     try {
267         stack.pop();    // should cause an exception
268         flag = false;   // should never happen
269     } catch (std::out_of_range oor) {
270         // flag should still be true
271         CHECK(flag);
272     }
273 }
274
275 // doctest unit test for push function
276 TEST_CASE("testing␣Stack<T>::push()") {
277     Stack<char> stack;
278     stack.push('a');
279     CHECK(stack.peek() == 'a');
280     CHECK(stack.size() == 1u);
281
282     stack.push('b');
283     CHECK(stack.size() == 2u);
284     CHECK(stack.pop() == 'b');
285     CHECK(stack.peek() == 'a');
286 }
287
288 // doctest unit test for size function
289 TEST_CASE("testing␣Stack<T>::size()") {
290     Stack<int> stack;
291     CHECK(stack.size() == 0);
292     for(int i = 0; i < 10; i++) {
293         stack.push(i);
294         CHECK(stack.size() == (i + 1));
295     }
296
297     for(int i = 10; i >= 1; i--) {
298         stack.pop();
299         CHECK(stack.size() == (i - 1));
300     }
301 }
302
303 /*
304  * Assignment operator override.
305  */
306 template <class T>
307 Stack<T> &Stack<T>::operator=(const Stack<T> &stack) {
308     copy(stack);
309     return *this;
310 }
311
312 // doctest unit test for the assignment operator
313 TEST_CASE("testing␣Stack<T>␣assignment␣operator") {
314     Stack<char> source;
315
316     for(char c = 'a'; c <= 'z'; c++) {
317         source.push(c);
318     }
319
320     Stack<char> stackCopy = source;
321
322     // should have same elements, in the same order
323     while(!stackCopy.isEmpty()) {
324         char a = source.pop();
325         char b = stackCopy.pop();
326         CHECK(a == b);
327     }
328
329     // make sure both are empty
330     bool res = source.isEmpty() && stackCopy.isEmpty();
331     CHECK(res);
332 }
333
334 // doctest unit test for the stream insertion operator
335 TEST_CASE("testing␣Stack<T>␣stream␣insertion") {
336     Stack<int> stack;
337
338     for(int i = 0; i < 5; i++) {
339         stack.push(i);
340     }
```

```
341
342     // test stream insertion by "printing" to a string
343     std::ostringstream oss;
344
345     oss << stack;
346
347     // did the output match?
348     CHECK(oss.str() == "[4, 3, 2, 1, 0]");
349 }
```

Listing 3: Queue.hpp

```
1  #pragma once
2
3  #include <doctest.h>
4  #include <iostream>
5  #include <stdexcept>
6  #include <sstream>
7  #include <string>
8  #include "../1-DLL/DLL.hpp"
9
10 /*-----------------------------------------------------------------------------
11  * class definition
12  *----------------------------------------------------------------------------*/
13
14 /**
15  * @brief CMP 246 Module 5 generic queue.
16  *
17  * Queue is a generic queue data structure. It supports enqueue and dequeue
18  * methods, and clear, isEmpty, and size administrative methods. It also has
19  * a copy constructor, and overrides the assignment and stream insertion
20  * operators.
21  */
22 template <class T> class Queue {
23 public:
24     /**
25      * @brief Default constructor.
26      *
27      * Make a new, empty queue.
28      */
29     Queue() { }
30
31     /**
32      * @brief Copy constructor.
33      *
34      * Make a new queue, just like the parameter.
35      *
36      * @param queue Constant reference to the queue to copy from.
37      */
38     Queue(const Queue<T> &queue) { copy(queue); }
39
40     /**
41      * @brief Empty the queue.
42      *
43      * Removes all elements from this queue.
44      */
45     void clear() { list.clear(); }
46
47     /**
48      * @brief Dequeue first element.
49      *
50      * Removes and returns the first element from the queue.
51      *
52      * @return Element of type T that was the first element in the queue.
53      *
54      * @throws std::out_of_range if the queue is empty
55      */
56     T dequeue();
57
58     /**
59      * @brief Enqueue a new element.
60      *
61      * Adds a value of type T to the back of the queue.
62      *
63      * @param v Element to enqueue
64      */
```

```
65      void enqueue(const T& v) { list.addLast(v); }
66
67      /**
68       * @brief Determine if the queue is empty.
69       *
70       * @return True if the queue is empty, false otherwise.
71       */
72      bool isEmpty() { return list.isEmpty(); }
73
74      /**
75       * @brief Assignment operator override.
76       *
77       * @param queue Queue to copy from.
78       *
79       * @return Reference to this queue, after it has copied the parameter,
80       * for chaining purposes.
81       */
82      Queue<T> &operator=(const Queue<T> &queue);
83
84      /**
85       * @brief Stream insertion override.
86       *
87       * @param out ostream object to output to, e.g., cout.
88       *
89       * @param queue Queue to output.
90       *
91       * @return Reference to the out ostream object.
92       */
93      friend std::ostream &operator<<(std::ostream &out, const Queue<T> &queue) {
94          out << queue.list;
95          return out;
96      }
97
98      /**
99       * @brief Get the size of the queue.
100      *
101      * @return Number of elements in the queue.
102      */
103     unsigned size() { return list.size(); }
104
105 private:
106     /**
107      * Doubly-linked list used as the underlying data store for the queue.
108      */
109     DLL<T> list;
110
111     /**
112      * @brief Copy queue contents.
113      *
114      * Private helper method for copy constructor and assignment override.
115      *
116      * @param queue Queue to copy from.
117      */
118     void copy(const Queue<T> &queue) { list = queue.list; }
119 };
120
121 //-----------------------------------------------------------------------------
122 // function implementations
123 //-----------------------------------------------------------------------------
124
125 // doctest unit tests for copy constructor
126 TEST_CASE("Testing␣Queue<T>::Queue(queue)␣constructor") {
127     Queue<int> q1;
128     for(int i = 1; i <= 5; i++) {
129         q1.enqueue(i);
130     }
131
132     Queue<int> q2(q1);
133
134     for(int i = 1; i <= 5; i++) {
135         CHECK(i == q2.dequeue());
136     }
137 }
138
139 // doctest unit tests for clear method
140 TEST_CASE("Testing␣Queue<T>::clear()") {
```

```
141     Queue<char> q;
142     for(char c = 'a'; c <= 'z'; c++) {
143         q.enqueue(c);
144     }
145     q.clear();
146     CHECK(q.isEmpty());
147 }
148
149 // copy is private, so it is tested in copy constructor and assignment
150 // operator tests
151
152 /*
153  * Dequeue method.
154  */
155 template <class T>
156 T Queue<T>::dequeue() {
157     if(list.isEmpty()) {
158         throw std::out_of_range("dequeue of empty queue");
159     }
160
161     return list.removeFirst();
162 }
163
164 // doctest unit tests for dequeue method
165 TEST_CASE("Testing Queue<T>::dequeue()") {
166     Queue<int> q;
167     for(int i = 0; i < 5; i++) {
168         q.enqueue(i);
169     }
170
171     for(int i = 0; i < 5; i++) {
172         CHECK(i == q.dequeue());
173         CHECK((4 - i) == q.size());
174     }
175
176     bool flag = true;
177     try {
178         q.dequeue();  // should cause an exception
179         flag = false;   // should never happen
180     } catch(std::out_of_range oor) {
181         CHECK(flag);
182     }
183 }
184
185 // doctest unit tests for enqueue method
186 TEST_CASE("Testing Queue<T>::enqueue()") {
187     Queue<double> q;
188
189     for(int i = 1; i <= 5; i++) {
190         q.enqueue(i);
191         CHECK(i == q.size());
192     }
193
194     for(int i = 1; i <= 5; i++) {
195         CHECK(i == q.dequeue());
196     }
197 }
198
199 // doctest unit tests for isEmpty method
200 TEST_CASE("Testing Queue<T>::isEmpty()") {
201     Queue<std::string> q;
202
203     CHECK(q.isEmpty());
204
205     q.enqueue("Bob");
206
207     CHECK(!q.isEmpty());
208
209     q.dequeue();
210
211     CHECK(q.isEmpty());
212 }
213
214 /*
215  * Implementation for assignment operator.
216  */
```

```
217 template <class T>
218 Queue<T> & Queue<T>::operator=(const Queue<T> &queue) {
219     copy(queue);
220     return *this;
221 }
222
223 // doctest unit test for assignment operator
224 TEST_CASE("Testing␣Queue<T>::operator=") {
225     Queue<int> q1, q2;
226
227     for(int i = 0; i < 5; i++) {
228         q1.enqueue(i);
229         q2.enqueue(5 - i);
230     }
231     q2.enqueue(-1);
232
233     q2 = q1;
234
235     CHECK(5 == q2.size());
236     for(int i = 0; i < 5; i++) {
237         CHECK(i == q2.dequeue());
238     }
239 }
240
241 // doctest unit test for the stream insertion operator
242 TEST_CASE("testing␣Queue<T>␣stream␣insertion") {
243     Queue<int> q;
244
245     for(int i = 0; i < 5; i++) {
246         q.enqueue(i);
247     }
248
249     // test stream insertion by "printing" to a string
250     std::ostringstream oss;
251
252     oss << q;
253
254     // did the output match?
255     CHECK(oss.str() == "[0,␣1,␣2,␣3,␣4]");
256 }
257
258 // doctest unit tests for size method
259 TEST_CASE("Testing␣Queue<T>::size") {
260     Queue<char> q;
261
262     CHECK(0 == q.size());
263
264     for(int i = 1; i <= 5; i++) {
265         q.enqueue('a');
266         CHECK(i == q.size());
267     }
268
269     for(int i = 4; i >= 0; i--) {
270         q.dequeue();
271         CHECK(i == q.size());
272     }
273 }
```

Listing 4: RPN.cpp

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <set>
4  #include <string>
5  #include "../2-Stack/Stack.hpp"
6
7  /**
8   * Main program for the Doane CMP 246 Module 5 RPN calculator.
9   */
10 int main() {
11
12     // welcome prompt
13     std::cout << "Welcome␣to␣the␣Doane␣RPN␣Calculator!" << std::endl;
14     std::cout << "Please␣enter␣an␣expression␣in␣postfix,␣EOF␣to␣quit."
15         << std::endl;
16
```

```
17      // prepare stack
18      Stack<double> stack;
19
20      // read string tokens until there is nothing more to read
21      std::string token;
22      double a, b;
23      while(std::cin >> token) {
24          if(token == "E") {
25              // print result of expression
26              std::cout << ">>␣" << stack.pop() << std::endl;
27              stack.clear();
28          } else if(token == "+") {
29              // addition
30              b = stack.pop();
31              a = stack.pop();
32              stack.push(a + b);
33          } else if(token == "-") {
34              // subtraction
35              b = stack.pop();
36              a = stack.pop();
37              stack.push(a - b);
38          } else if(token == "*") {
39              // multiplication
40              b = stack.pop();
41              a = stack.pop();
42              stack.push(a * b);
43          } else if(token == "/") {
44              // division
45              b = stack.pop();
46              a = stack.pop();
47              stack.push(a / b);
48          } else {
49              // numeric entry
50              stack.push(atof(token.c_str()));
51          }
52      }
53
54      // good bye prompt
55      std::cout << "Good␣bye!" << std::endl;
56
57      return EXIT_SUCCESS;
58  }
```