Listing 1: IteratorSLL.hpp

```
#pragma once

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

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

/**
 * @brief CMP 246 Module 4 generic singly-linked list, supporting iterators.
 *
 * IteratorSLL is a generic singly-linked list data structure. It
 * allows inserting at the front of the list, and supports index-based
 * get, set, and remove operations. The list also provides a contains method,
 * and the administrative methods clear, isEmpty, and size. IteratorSLL also
 * has a copy constructor, and overrides the assignment and stream insertion
 * operators. IteratorSLL provides front and end methods to access iterators
 * that move through the list from front to back.
 */
template <class T> class IteratorSLL {

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

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

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

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

public:
    /**
     * @brief IteratorSLL iterator.
     *
     * This class allows IteratorSLL users to iterate through the list, from
     * front to back, without exposing the pointer structure of the list, and
```

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

```
151     /**
152      * @brief Default list constructor.
153      *
154      * Made an initially empty list.
155      */
156     IteratorSLL() : pHead(0), n(0u) { }
157
158     /**
159      * @brief Copy construstor.
160      *
161      * Make a new, deep-copy list, just like the parameter list.
162      *
163      * @param otherList Reference to the IteratorSLL to copy.
164      */
165     IteratorSLL(const IteratorSLL<T> &otherList) :
166         pHead(0), n(0u) { copy(otherList); }
167
168     /**
169      * @brief Destructor.
170      *
171      * Free the memory used by this list.
172      */
173     ~IteratorSLL() { clear(); }
174
175     /**
176      * @brief Add a value to the front of the list.
177      *
178      * @param d Value to add to the list.
179      */
180     void add(const T &d);
181
182     /**
183      * @brief Clear the list.
184      *
185      * Remove all the elements from the list.
186      */
187     void clear();
188
189     /**
190      * @brief Search the list for a specified value.
191      *
192      * Searches for a value and returns an iterator on the first occurrence
193      * of the value in the list, or end if the value is not in the list.
194      *
195      * @param d Value to search for.
196      *
197      * @return Iterator on the first occurrence of d in the list, or end if
198      * d is not in the list.
199      */
200     Iterator contains(const T &d) const;
201
202     /**
203      * @brief Get an Iterator representing the end of the list.
204      *
205      * @return An Iterator positioned after the last node in the list.
206      */
207     Iterator end() const { return Iterator(0); }
208
209     /**
210      * @brief Get an Iterator on the first node of the list.
211      *
212      * @return An Iterator positioned on the first node of the list.
213      */
214     Iterator front() const { return Iterator(pHead); }
215
216     /**
217      * @brief Get a value.
218      *
219      * Get the value at a specified index in the list.
220      *
221      * @param idx Index of the value to get.
222      *
223      * @throws std::out_of_range if the index is past the end of the list.
224      *
225      * @return Value at location idx in the list.
226      */
```

```
227     T get (unsigned idx) const;
228
229     /**
230      * @brief Determine if the list is empty.
231      *
232      * Convenience method to test if the list contains no elements.
233      *
234      * @return true if the list is empty, false otherwise.
235      */
236     bool isEmpty() const { return size() == 0u; }
237
238     /**
239      * @brief Remove an element.
240      *
241      * Remove the value at a specified index in the list.
242      *
243      * @param idx Index of the element to remove.
244      *
245      * @throws std::out_of_range if the index is past the end of the list.
246      *
247      * @return Value that was at location idx.
248      */
249     T remove(unsigned idx);
250
251     /**
252      * @brief Change a list element.
253      *
254      * Change the value at a specified index to another value.
255      *
256      * @param idx Index of the value to change.
257      *
258      * @throws std::out_of_range if the index is past the end of the list.
259      *
260      * @param d New value to place in position idx.
261      */
262     void set(unsigned idx, const T &d);
263
264     /**
265      * @brief Get list size.
266      *
267      * Get the number of integers in the list.
268      *
269      * @return The number of integers in the list.
270      */
271     unsigned size() const { return n; }
272
273     /**
274      * @brief Assignment operator.
275      *
276      * Override of the assignment operator to work with IteratorSLL objects.
277      * Makes this list a deep-copy, identical structure as the parameter
278      * IteratorSLL.
279      *
280      * @param list IteratorSLL to copy from
281      *
282      * @return Reference to this object.
283      */
284     IteratorSLL<T> &operator=(const IteratorSLL<T> &otherList);
285
286 private:
287     /**
288      * @brief Copy helper method.
289      *
290      * This private helper method is used to deep-copy all of the elements from
291      * the parameter list to this list. Any existing elements in this list are
292      * safely removed before the copy.
293      *
294      * @param otherList Reference to the IteratorSLL object to copy from.
295      */
296     void copy(const IteratorSLL<T> &otherList);
297
298     /**
299      * Pointer to the first Node in the list, or 0 if the list is empty.
300      */
301     Node *pHead;
302
```

```
303     /**
304      * Number of integers in the list.
305      */
306     size_t n;
307 };
308 
309 //------------------------------------------------------------------------------
310 // function implementations
311 //------------------------------------------------------------------------------
312 
313 /*
314  * Iterator dereferencing operator override.
315  */
316 template <class T>
317 T &IteratorSLL<T>::Iterator::operator*() {
318     // if the iterator is past the end of the list, throw an exception
319     if(pCurr == 0) {
320         throw std::out_of_range("Dereferencing␣beyond␣list␣end␣in␣"
321                                 "Iterator::*()");
322     }
323 
324     return pCurr->data;
325 }
326 
327 // doctest unit test for iterator dereferencing
328 TEST_CASE("testing␣IteratorSLL<T>::Iterator␣dereferencing") {
329     IteratorSLL<int> list;
330 
331     list.add(0);
332     IteratorSLL<int>::Iterator it = list.front();
333 
334     // first element should be 0
335     CHECK(*it == 0);
336 
337     // check exception handling when dereferencing past end of list
338     it = list.end();
339     bool flag = true;
340     try {
341         *it;            // should throw an exception
342         flag = false;   // this should never happen
343     } catch (std::out_of_range oor) {
344         CHECK(flag);
345     }
346 }
347 
348 /*
349  * Iterator increment operator overload.
350  */
351 template <class T>
352 typename IteratorSLL<T>::Iterator
353     &IteratorSLL<T>::Iterator::operator++(int dummy) {
354 
355     // if the iterator is past the end of the list, throw an exception
356     if(pCurr == 0) {
357         throw std::out_of_range("Increment␣beyond␣list␣end␣in␣"
358                                 "Iterator::++()");
359     }
360 
361     pCurr = pCurr->pNext;
362     return *this;
363 }
364 
365 // doctest unit test for iterator increment overload
366 TEST_CASE("testing␣IteratorSLL<T>::Iterator␣postfix␣increment") {
367     IteratorSLL<char> list;
368 
369     // populate with a - z
370     for(char c = 'a'; c <= 'z'; c++) {
371         list.add(c);
372     }
373 
374     // verify that iterating moves through the list
375     IteratorSLL<char>::Iterator it = list.front();
376     for(char c = 'z'; c >= 'a'; c--) {
377         CHECK(*it == c);
378         it++;
```

```
379     }
380
381     // check exception handling when incrementing beyond list end
382     bool flag = true;
383     try {
384         it++;           // this should throw an exception
385         flag = false;   // this should never happen
386     } catch(std::out_of_range oor) {
387         CHECK(flag);
388     }
389 }
390
391 // doctest unit test for the copy constructor
392 TEST_CASE("testing␣IteratorSLL<T>␣copy␣constructor") {
393     IteratorSLL<int> list1;
394
395     // populate the original list
396     for(int i = 0; i < 5; i++) {
397         list1.add(i);
398     }
399
400     // make a new list like original
401     IteratorSLL<int> list2(list1);
402
403     // does it have the right size?
404     CHECK(list2.size() == list1.size());
405
406     // does it have the right elements?
407     for(int i = 0; i < 5; i++) {
408         CHECK(list2.get(i) == (4 - i));
409     }
410
411     // try it again with dynamic allocation
412     IteratorSLL<int> *pList = new IteratorSLL<int>(list1);
413
414     // does it have the right size?
415     CHECK(pList->size() == list1.size());
416
417     // does it have the right elements?
418     for(int i = 0; i < 5; i++) {
419         CHECK(pList->get(i) == (4 - i));
420     }
421
422     delete pList;
423 }
424
425 /*
426  * Add d to the front of the list.
427  */
428 template <class T>
429 void IteratorSLL<T>::add(const T &d) {
430     // create the new node
431     Node *pN = new Node(d, pHead);
432
433     // change head pointer to point to the new node
434     pHead = pN;
435
436     // increment size
437     n++;
438 }
439
440 // doctest unit test for the add method
441 TEST_CASE("testing␣IteratorSLL<T>::add") {
442     IteratorSLL<int> list;
443
444     // each addition should happen at the front, and the size should go up by
445     // one each time
446     list.add(1);
447     CHECK(list.get(0) == 1);
448     CHECK(list.size() == 1u);
449
450     list.add(2);
451     CHECK(list.get(0) == 2);
452     CHECK(list.size() == 2u);
453 }
454
```

```
455 /*
456  * Delete all list nodes.
457  */
458 template <class T>
459 void IteratorSLL<T>::clear() {
460     // create cursors
461     Node *pCurr = pHead, *pPrev = 0;
462 
463     // iterate thru list, deleting each node
464     while(pCurr != 0) {
465         // "inchworm" up to next node
466         pPrev = pCurr;
467         pCurr = pCurr->pNext;
468 
469         // delete previous node
470         delete pPrev;
471     }
472 
473     // reset head pointer and size
474     pHead = 0;
475     n = 0u;
476 }
477 
478 // doctest unit test for the clear method
479 TEST_CASE("testing␣IteratorSLL<T>::clear") {
480     IteratorSLL<int> list;
481 
482     // add some list elements
483     for(int i = 0; i < 100; i++) {
484         list.add(i);
485     }
486 
487     // clear should make size equal zero
488     list.clear();
489     CHECK(list.size() == 0u);
490 }
491 
492 /*
493  * Search the list for value d.
494  */
495 template <class T>
496 typename IteratorSLL<T>::Iterator IteratorSLL<T>::contains(const T &d) const {
497     // create cursor
498     Node *pCurr = pHead;
499 
500     // iterate until we find d or end of list
501     while(pCurr != 0) {
502 
503         // found it? return its iterator
504         if(pCurr->data == d) {
505             return Iterator(pCurr);
506         }
507 
508         pCurr = pCurr->pNext;
509     }
510 
511     // not found? return flag value
512     return Iterator(0);
513 }
514 
515 // doctest unit test for the contains method
516 TEST_CASE("testing␣IteratorSLL<T>::contains") {
517     IteratorSLL<char> list;
518 
519     // populate the list
520     for(char c = 'A'; c <= 'Z'; c++) {
521         list.add(c);
522     }
523 
524     // search for 1st element in list
525     CHECK(*(list.contains('Z')) == 'Z');
526 
527     // search for last element in list
528     CHECK(*(list.contains('A')) == 'A');
529 
530     // search for something in the middle
```

```
531     CHECK(*(list.contains('M')) == 'M');
532
533     // search for something not in list
534     CHECK(list.contains('a') == list.end());
535 }
536
537 /*
538  * Make this list a deep copy of another list.
539  */
540 template <class T>
541 void IteratorSLL<T>::copy(const IteratorSLL<T> &otherList) {
542     // remove any existing data
543     clear();
544
545     // initialize two cursors: one for this list, one for the other list
546     Node *pCurr = pHead, *pOtherCurr = otherList.pHead;
547
548     // iterate through the nodes in the other list
549     while(pOtherCurr != 0) {
550         // special case: the first node changes the head pointer
551         if(pHead == 0) {
552             pHead = new Node(pOtherCurr->data, 0);
553             pCurr = pHead;
554         } else {
555             // general case: add new node to end of this list
556             pCurr->pNext = new Node(pOtherCurr->data, 0);
557             pCurr = pCurr->pNext;
558         }
559
560         // move to next node in other list, and increment our size
561         pOtherCurr = pOtherCurr->pNext;
562         n++;
563     }
564 }
565
566 // since copy is private, it's tested indirectly in copy constructor and
567 // assignment operator tests
568
569 // doctest unit test for the end method
570 TEST_CASE("testing␣IteratorSLLT<T>::end") {
571     IteratorSLL<double> list;
572
573     // iterating through empty list should not happen
574     IteratorSLL<double>::Iterator it = list.front();
575     int count = 0;
576     for(; it != list.end(); it++) {
577         count++;
578     }
579     CHECK(count == 0);
580
581     // iterating through a list w/ 5 elements
582     for(int i = 0; i < 5; i++) {
583         list.add(i);
584     }
585     it = list.front();
586     count = 0;
587     for(; it != list.end(); it++) {
588         count++;
589     }
590     CHECK(count == 5);
591 }
592
593 // doctest unit test for the front method
594 TEST_CASE("testing␣IteratorSLL<T>::front") {
595     IteratorSLL<int> list;
596     list.add(0);
597
598     // is the front iterator the first element?
599     IteratorSLL<int>::Iterator it = list.front();
600     CHECK(*it == list.get(0));
601
602     // add another and repeat the check
603     list.add(1);
604     it = list.front();
605     CHECK(*it == list.get(0));
606 }
```

```
607
608 /*
609  * Get the value at location idx.
610  */
611 template <class T>
612 T IteratorSLL<T>::get(unsigned idx) const {
613     // if the idx is past list end, throw an exception
614     if(idx >= n) {
615         throw std::out_of_range("Index␣out␣of␣range␣in␣IteratorSLL::get()");
616     }
617
618     // initialize cursor
619     Node *pCurr = pHead;
620
621     // iterate cursor to position
622     for(unsigned i = 0u; i < idx; i++) {
623         pCurr = pCurr->pNext;
624     }
625
626     // return requested value
627     return pCurr->data;
628 }
629
630 // doctest unit test for the get method
631 TEST_CASE("testing␣IteratorSLL<T>::get") {
632     IteratorSLL<char> list;
633
634     // populate list
635     for(char c = 'A'; c <= 'Z'; c++) {
636         list.add(c);
637     }
638
639     // get first element
640     CHECK(list.get(0) == 'Z');
641
642     // get last element
643     CHECK(list.get(25) == 'A');
644
645     // get something in the middle
646     CHECK(list.get(13) == 'M');
647
648     // check exception handling when access is beyond list
649     bool flag = true;
650     try {
651         list.get(26); // list element 26 does not exist
652         flag = false; // this line should not be reached, due to an exception
653     } catch(std::out_of_range oor) {
654         // verify flag wasn't modified
655         CHECK(flag);
656     }
657 }
658
659 /*
660  * Remove node at location idx.
661  */
662 template <class T>
663 T IteratorSLL<T>::remove(unsigned idx) {
664     // if the idx is past list end, throw an exception
665     if(idx >= n) {
666         throw std::out_of_range("Index␣out␣of␣range␣in␣IteratorSLL::remove()");
667     }
668
669     // initialize cursors
670     Node *pCurr = pHead, *pPrev = 0;
671
672     // iterate cursors to position
673     for(unsigned i = 0u; i < idx; i++) {
674         pPrev = pCurr;
675         pCurr = pCurr->pNext;
676     }
677
678     // save value so we can return it
679     T d = pCurr->data;
680
681     // first element? change head pointer
682     if(pCurr == pHead) {
```

```
683         pHead = pCurr->pNext;
684     } else {
685         // general case: "wire around" node
686         pPrev->pNext = pCurr->pNext;
687     }
688 
689     // remove node and decrement size
690     delete pCurr;
691     n--;
692 
693     // send back removed value
694     return d;
695 }
696 
697 // doctest unit test for the remove method
698 TEST_CASE("testing␣IteratorSLL<T>::remove") {
699     IteratorSLL<char> list;
700 
701     // populate list
702     for(char c = 'A'; c <= 'Z'; c++) {
703         list.add(c);
704     }
705 
706     // remove first element
707     CHECK(list.remove(0) == 'Z');
708     CHECK(list.size() == 25);
709     CHECK(list.get(0) == 'Y');
710 
711     // remove last element
712     CHECK(list.remove(24) == 'A');
713     CHECK(list.size() == 24);
714     CHECK(list.get(23) == 'B');
715 
716     // remove something in the middle
717     CHECK(list.remove(12) == 'M');
718     CHECK(list.size() == 23);
719     CHECK(list.get(12) == 'L');
720 
721     // check exception handling when access is beyond end of the list
722     bool flag = true;
723     try {
724         list.remove(26);    // illegal access; element 26 doesn't exist
725         flag = false;       // this line should not be reached due to exception
726     } catch(std::out_of_range oor) {
727         CHECK(flag);
728     }
729 }
730 
731 /*
732  * Change the value at location idx to d.
733  */
734 template <class T>
735 void IteratorSLL<T>::set(unsigned idx, const T &d) {
736     // if the idx is past list end, throw an exception
737     if(idx >= n) {
738         throw std::out_of_range("Index␣out␣of␣range␣in␣IteratorSLL::set()");
739     }
740 
741     // initialize cursor
742     Node *pCurr = pHead;
743 
744     // iterate to location
745     for(unsigned i = 0u; i < idx; i++) {
746         pCurr = pCurr->pNext;
747     }
748 
749     // change data in location idx to d
750     pCurr->data = d;
751 }
752 
753 // doctest unit test for the set method
754 TEST_CASE("testing␣IteratorSLL<T>::set") {
755     IteratorSLL<char> list;
756 
757     // populate the list
758     for(char c = 'A'; c <= 'Z'; c++) {
```

```
759         list.add(c);
760     }
761
762     // set first element
763     list.set(0, 'z');
764     CHECK(list.get(0) == 'z');
765
766     // set last element
767     list.set(25, 'a');
768     CHECK(list.get(25) == 'a');
769
770     // set something in the middle
771     list.set(13, 'm');
772     CHECK(list.get(13) == 'm');
773
774     // check exception handling for index beyond end of list
775     bool flag = true;
776     try {
777         list.set(26, 'X');  // this is illegal; index doesn't exist
778         flag = false;       // this should never be reached, due to exception
779     } catch(std::out_of_range oor) {
780         CHECK(flag);    // if exception was handled properly, should be true
781     }
782 }
783
784 /*
785  * Assignment operator.
786  */
787 template <class T>
788 IteratorSLL<T> & IteratorSLL<T>::operator=(const IteratorSLL<T> &otherList) {
789     // remove any existing contents first
790     clear();
791
792     // copy other list contents to this object
793     copy(otherList);
794
795     return *this;
796 }
797
798 // doctest unit test for the assignment operator
799 TEST_CASE("testing␣IteratorSLL<T>␣assignment") {
800     IteratorSLL<int> list1, list2;
801
802     // populate lists
803     for(int i = 0; i < 5; i++) {
804         list1.add(i);
805         if(i % 2 == 0) {
806             list2.add(i);
807         }
808     }
809
810     // do the assignment
811     list1 = list2;
812
813     // right size?
814     CHECK(list1.size() == list2.size());
815
816     // same contents?
817     for(unsigned i = 0; i < list1.size(); i++) {
818         CHECK(list1.get(i) == list2.get(i));
819     }
820 }
821
822 /*
823  * Override of the stream insertion operator. Using iterators removes
824  * the need for this to be a friend of the IteratorSLL class.
825  */
826 template <class T>
827 std::ostream &operator<<(std::ostream &out, const IteratorSLL<T> &list) {
828
829     out << "[";
830
831     // iterate through the list using an iterator
832     typename IteratorSLL<T>::Iterator i = list.front();
833
834     while(i != list.end()) {
```

```
835
836         out << *i;
837
838         // output comma for all but last element
839         i++;
840         if(i != list.end()) {
841             out << ",␣";
842         }
843     }
844
845     out << "]";
846
847     return out;
848 }
849
850 // doctest unit test for the stream insertion operator
851 TEST_CASE("testing␣IteratorSLL<T>␣stream␣insertion") {
852     IteratorSLL<int> list;
853
854     for(int i = 0; i < 5; i++) {
855         list.add(i);
856     }
857
858     // test stream insertion by "printing" to a string
859     std::ostringstream oss;
860
861     oss << list;
862
863     // did the output match?
864     CHECK(oss.str() == "[4,␣3,␣2,␣1,␣0]");
865 }
```

Listing 2: PRNGTesting.cpp

```
1  #include <climits>
2  #include <cmath>
3  #include <cstdlib>
4  #include <ctime>
5  #include <iostream>
6  #include <random>
7  #include "../1-IteratorSLL/IteratorSLL.hpp"
8
9  // global integer for use in the RANDU PRNG
10 int vj;
11
12 /**
13  * @brief Infamous RANDU PRNG
14  *
15  * RANDU was provided as the random number generator for IBM mainframes in
16  * the 1960s and early 1970s. It is very flawed, and once the flaws were
17  * discovered, quite a bit of research that used pseudo-random numbers had
18  * to be re-done.
19  *
20  * @see https://en.wikipedia.org/wiki/RANDU
21  *
22  * @return int Pseudo-random integer in the range [INT_MIN, INT_MAX]
23  */
24 int randu() {
25     vj = 65539L * vj % 0x80000000;
26
27     // map vj to [INT_MIN, INT_MAX]
28     long double os = INT_MIN;
29     long double oe = INT_MAX;
30     long double is = 1;
31     long double ie = 2147483647;
32     long double o = os + (oe - os) / (ie - is) * (vj - is);
33     return (int)o;
34 }
35
36 /**
37  * @brief NIST monobit statistical test of PRNG quality
38  *
39  * This is a fundamental test of pseudo-random number generator quality.
40  * Given a sequence of values, this test looks at the proportion of zeroes and
41  * ones in their binary representations. A return value less than 0.01 indicates
42  * that the sequence does not appear to be random.
```

```
 *
 * @see https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf
 *
 * @param numbers List of pseudo-random integers in the range [INT_MIN, INT_MAX]
 *
 * @return double P-value, representing confidence that the sequence appears to
 * be random. A value less than 0.01 indicates that the sequence does not
 * appear to be random.
 */
double monobit(const IteratorSLL<int> &numbers) {
    double S = 0.0;

    // iterate through all values
    IteratorSLL<int>::Iterator i = numbers.front();
    while(i != numbers.end()) {
        // assuming 32-bit integers here
        unsigned mask = 0x00000001;

        // sum up zeroes / ones according to the NIST monobit test
        for(int j = 0; j < 32; j++) {
            int v = *i;
            S += (v & mask) == 0x00000000 ? -1.0 : 1.0;
            mask <<= 1;
        }
        i++;
    }

    // compute P-value according to the NIST monobit test
    double stat = abs(S) / sqrt(numbers.size() * 32.0);
    double PValue = erfc(stat / sqrt(2.0));

    return PValue;
}

/**
 * @brief Application entry point
 *
 * @return int Exit status for the app.
 */
int main() {
    // Mersenne Twister: a high-quality PRNG
    std::mt19937_64 mtPRNG(time(0));

    // Minimum standard PRNG: just good enough
    std::minstd_rand0 minStdPRNG(time(0));

    // distribution object to map values to the proper range
    std::uniform_int_distribution<int> dist(INT_MIN, INT_MAX);

    // list holding pseudo-random integers
    IteratorSLL<int> randInts;

    // number of values to generate / test
    int n = 10000000;

    // test the Mersenne Twister PRNG
    for(int i = 0; i < n; i++) {
        int v = dist(mtPRNG);
        randInts.add(v);
    }
    double PValue = monobit(randInts);
    std::cout << "Mersenne␣Twister␣monobit:␣" << PValue << std::endl;

    // test the minimum standard PRNG
    randInts.clear();
    for(int i = 0; i < n; i++) {
        int v = dist(minStdPRNG);
        randInts.add(v);
    }
    PValue = monobit(randInts);
    std::cout << "Minimum␣standard␣monobit:␣" << PValue << std::endl;

    // test RANDU PRNG
    randInts.clear();
    vj = time(0);
    for(int i = 0; i < n; i++) {
```

```
119             int v = randu();
120             randInts.add(v);
121         }
122         PValue = monobit(randInts);
123         std::cout << "RANDU␣monobit:␣" << PValue << std::endl;
124 
125         return EXIT_SUCCESS;
126     }
```