```
/*****************************************************************************
 * AUTHOR           : Nick Reardon
 * Extra Credit #1  : Extendable Array
 * CLASS            : CS1D
 * SECTION          : MW - 2:30p
 * DUE DATE         : 02 / 12 / 20
 *****************************************************************************/
#ifndef _EXTENDABLEARRAY_H_
#define _EXTENDABLEARRAY_H_

#include <string>
#include <exception>
#include "Except.h"

/**
 * @enum    ERROR_TYPE
 * @brief   Values that represent ExtendableArray error types
 */
enum ERROR_TYPE
{
    DEFUALT,
    FULL,
    EMPTY,
    OUT_OF_RANGE
};


 /**
  * @class  ExtendableArray ExtendableArray.h ExtendableArray.h
  *
  * @brief  Extendable circular array template class
  *
  * @author Nick Reardon
  * @date   11/25/2020
  *
  * @tparam Type    template datatype.
  */
template<class Type>
class ExtendableArray
{
private:
    Type* typeAr;       /** @brief  Template datatype pointer for array
      allocation */

    int capacity;       /** @brief  The current reserved memory locations of the
      array */
    int currentSize;    /** @brief  count of used indeces in the array */

    int frontIndex;     /** @brief  tracks the first index of the circular array
      */
    int endIndex;       /** @brief  tracks the last index of the circular array
      */
```

```
49
50
51  protected:
52
53  #ifndef Protected_Methods //-------------------------------------------------
54
55      /**
56       * @fn  void ExtendableArray::shiftRight_Outward(const int givenIndex)
57       * @brief   Shift array members right/up, wrapping around to the end
58       *
59       * @exception   Except  Exception thrown when class is empty.
60       *
61       * @param   givenIndex  int value passed from outside of the object index is ↵
           adjusted for use in
62       *                      the method.
63       */
64      void shiftRight_Outward(const int givenIndex)
65      {
66
67          if (empty())
68          {
69              throw Except("Array is empty", EMPTY, 5);
70          }
71
72          if (full())
73          {
74              expand();
75          }
76
77
78          if (endIndex == capacity - 1)
79          {
80              endIndex = 0;
81          }
82          else
83          {
84              endIndex++;
85          }
86
87
88          int tempIndex = endIndex;
89          for (int i = capacity - 1; i > givenIndex; i--)
90          {
91              if (tempIndex == 0)
92              {
93                  typeAr[tempIndex] = typeAr[capacity - 1];
94
95                  tempIndex = capacity - 1;
96              }
97              else
98              {
99                  typeAr[tempIndex] = typeAr[tempIndex - 1];
```

```
100
101                 tempIndex--;
102             }
103         }
104     }
105
106     /**
107      * @fn  void ExtendableArray::shiftLeft_Outward(const int givenIndex)
108      * @brief   Shift array members left/down, wrapping around to the end
109      *
110      * @exception   Except  Exception thrown when class is empty.
111      *
112      * @param   givenIndex  int value passed from outside of the object index is
       adjusted for use in
113      *                      the method.
114      */
115     void shiftLeft_Outward(const int givenIndex)
116     {
117
118         if (empty())
119         {
120             throw Except("Array is empty", EMPTY, 5);
121         }
122
123         if (full())
124         {
125             expand();
126         }
127
128         if (frontIndex == 0)
129         {
130             frontIndex = capacity - 1;
131         }
132         else
133         {
134             frontIndex--;
135         }
136
137         int tempIndex = frontIndex;
138         for (int i = 0; i < givenIndex; i++)
139         {
140             if (tempIndex == capacity - 1)
141             {
142                 typeAr[tempIndex] = typeAr[0];
143
144                 tempIndex = 0;
145             }
146             else
147             {
148                 typeAr[tempIndex] = typeAr[tempIndex + 1];
149
150                 tempIndex++;
```

```
151             }
152         }
153     }
154
155     /**
156      * @fn  void ExtendableArray::shiftLeft_Inward(const int givenIndex)
157      * @brief   Shift array members left/down, wrapping around to the end
158      *
159      * @exception   Except  Exception thrown when class is empty.
160      *
161      * @param   givenIndex  int value passed from outside of the object index is
         adjusted for use in
162      *                      the method.
163      */
164     void shiftLeft_Inward(const int givenIndex)
165     {
166
167         int tempIndex = adjustedIndex(givenIndex);
168
169         if (empty())
170         {
171             throw Except("Array is empty", EMPTY, 5);
172         }
173
174         if (frontIndex == capacity - 1)
175         {
176             frontIndex = 0;
177         }
178         else
179         {
180             frontIndex++;
181         }
182
183
184         for (int i = givenIndex; i > 0; i--)
185         {
186
187             if (tempIndex == 0)
188             {
189                 tempIndex = capacity - 1;
190
191                 typeAr[0] = typeAr[tempIndex];
192             }
193             else
194             {
195                 typeAr[tempIndex] = typeAr[tempIndex - 1];
196
197                 tempIndex--;
198             }
199         }
200     }
201
```

```
202     /**
203      * @fn  void ExtendableArray::shiftRight_Inward(const int givenIndex)
204      * @brief   Shift array members right/up, wrapping around to the end
205      *
206      * @exception   Except  Exception thrown when class is empty.
207      *
208      * @param   givenIndex  int value passed from outside of the object index is adjusted for use in
209      *                      the method.
210      */
211     void shiftRight_Inward(const int givenIndex)
212     {
213
214         int tempIndex = adjustedIndex(givenIndex);
215
216         if (empty())
217         {
218             throw Except("Array is empty", EMPTY, 5);
219         }
220
221         if (endIndex == 0)
222         {
223             endIndex = capacity - 1;
224         }
225         else
226         {
227             endIndex--;
228         }
229
230
231         for (int i = givenIndex; i < capacity - 1; i++)
232         {
233
234             if (tempIndex == capacity - 1)
235             {
236                 tempIndex = capacity - 1;
237
238                 typeAr[tempIndex] = typeAr[0];
239             }
240             else
241             {
242                 typeAr[tempIndex] = typeAr[tempIndex + 1];
243
244                 tempIndex++;
245             }
246         }
247     }
248
249
250     /**
251      * @fn  void ExtendableArray::expand(const int minIncrease = 1)
252      * @brief   Expands the array to accomodate a given minimum increase
```

```
253         *
254         * @param   minIncrease (Optional) The minimum increase. Supports larger
              increases (e.g. adding
255         *                      multiple instances at once )
256         */
257        void expand(const int minIncrease = 1)
258        {
259            int newCapacity = capacity;
260            do
261            {
262                newCapacity = newCapacity * 2;
263
264            } while (newCapacity < currentSize + minIncrease);
265
266            Type* tempAr = new Type[newCapacity];
267
268            int tempIndex = frontIndex;
269            for (int i = 0; i < capacity; i++)
270            {
271                tempAr[i] = typeAr[tempIndex];
272
273                if (tempIndex == capacity - 1)
274                {
275                    tempIndex = 0;
276                }
277                else
278                {
279                    tempIndex++;
280                }
281            }
282
283            Type* hold = typeAr;
284            typeAr = tempAr;
285
286            delete[] hold;
287            hold = nullptr;
288            tempAr = nullptr;
289
290            frontIndex = 0;
291            endIndex = currentSize - 1;
292
293            capacity = newCapacity;
294        }
295
296        /**
297         * @fn  int ExtendableArray::adjustedIndex(const int givenIndex) const
298         * @brief   Given a normal index ( 0 .. capacity ) return the adjusted index
              to accomodate the
299         *          circular array
300         *
301         * @exception   Except  Exception thrown when class is empty Except Exception
              index is out of
```

```
302       *                       range.
303       *
304       * @param   givenIndex  Zero-based index of the given.
305       * @returns int the adjusted index.
306       */
307      int adjustedIndex(const int givenIndex) const
308      {
309
310          if (empty())
311          {
312              throw Except("Array is empty - nothing to return", EMPTY, 5);
313          }
314          else if (givenIndex < 0)
315          {
316              throw Except("Index out of range", OUT_OF_RANGE, 5);
317          }
318          else if (givenIndex > capacity - 1)
319          {
320              throw Except("Index out of range", OUT_OF_RANGE, 5);
321          }
322          else if (givenIndex > currentSize - 1)
323          {
324              throw Except("Index out of range", OUT_OF_RANGE, 5);
325          }
326          else
327          {
328              int realIndex;
329
330              if ((frontIndex + givenIndex) > capacity - 1)
331              {
332                  realIndex = frontIndex + givenIndex - capacity;
333              }
334              else
335              {
336                  realIndex = frontIndex + givenIndex;
337              }
338              return realIndex;
339          }
340
341      }
342
343      /**
344       * @fn  void ExtendableArray::destroy()
345       * @brief   Deletes the array
346       */
347      void destroy()
348      {
349          delete[] typeAr;
350      }
351
352
353  #endif // !Protected_Methods //-------------------------------------------------
```

```
354
355 public:
356
357 #ifndef Constructors_/
      _Deconstructors //-------------------------------------------------
358
359     /**
360      * @fn  ExtendableArray::ExtendableArray<Type>(const int newCapacity = 8)
361      * @brief   Constructor
362      *
363      *          @par    Creates empty array of either a specified capacity or
         uses the default value if
364      *          not provided
365      *
366      * @tparam  Type    Template datatype.
367      * @param   newCapacity (Optional) The new capacity.
368      */
369     ExtendableArray<Type>(const int newCapacity = 8)
370     {
371         frontIndex = -1;
372         endIndex = 0;
373         capacity = newCapacity;
374
375         typeAr = new Type[capacity];
376     }
377
378
379     /**
380      * @fn  ExtendableArray::ExtendableArray<Type>(const ExtendableArray<Type>&
         otherExArray)
381      * @brief   Copy Constructor
382      *
383      *          @par    Copies the values of the contents of another
         ExtendableArray into this one
384      *
385      * @tparam  Type    Template datatype.
386      * @param   otherExArray    The other array.
387      */
388     ExtendableArray<Type>(const ExtendableArray<Type>& otherExArray)
389     {
390         frontIndex = 0;
391         endIndex = otherExArray.currentSize - 1;
392
393         capacity = otherExArray.capacity;
394         currentSize = otherExArray.currentSize;
395
396         typeAr = new Type[capacity];
397
398         int tempIndex = frontIndex;
399         for (int i = 0; i < otherExArray.currentSize - 1; i++)
400         {
401             typeAr[i] = otherExArray.at[tempIndex];
```

```
402
403             if (tempIndex == capacity - 1)
404             {
405                 tempIndex = 0;
406             }
407             else
408             {
409                 tempIndex++;
410             }
411         }
412     }
413
414     /**
415      * @fn  ExtendableArray::ExtendableArray<Type>(const Type* ptrArray, const
        int arSize)
416      * @brief   Constructor
417      *
418      *          @par Copies the values of the contents of a given array into this
         one
419      *
420      * @tparam  Type    Template datatype.
421      * @param   ptrArray    array to copy values from.
422      * @param   arSize      size of the given array.
423      */
424     ExtendableArray<Type>(const Type* ptrArray, const int arSize)
425     {
426         frontIndex = 0;
427         endIndex = arSize - 1;
428
429         currentSize = arSize;
430         capacity = currentSize;
431
432         typeAr = new Type[capacity];
433
434         int tempIndex = frontIndex;
435         for (int i = 0; i < currentSize - 1; i++)
436         {
437             typeAr[i] = ptrArray[tempIndex];
438
439             if (tempIndex == capacity - 1)
440             {
441                 tempIndex = 0;
442             }
443             else
444             {
445                 tempIndex++;
446             }
447         }
448     }
449
450     /**
451      * @fn  ExtendableArray::~ExtendableArray()
```

```
452      * @brief   Destructor
453      */
454     ~ExtendableArray()
455     {
456         delete[] typeAr;
457     }
458
459
460 #endif // !Constructors_/_Deconstructors ----------------------------------------------
461
462 #ifndef Public_Methods //-------------------------------------------------
463
464     /**
465      * @fn  void ExtendableArray::clearAll(const int newCapacity = capacity)
466      * @brief   Deletes current array and allocates a new one with a given capacity
467      *
468      * @param   newCapacity (Optional) The capacity for the new array Can be left out to use current
469      *                      capacity.
470      */
471     void clearAll(const int newCapacity = capacity)
472     {
473         delete[] typeAr;
474
475         frontIndex = -1;
476         endIndex = 0;
477         capacity = newCapacity;
478
479         typeAr = new Type[capacity];
480     }
481
482     /**
483      * @fn  bool ExtendableArray::empty() const
484      * @brief   evaluates if the array is empty or not
485      *
486      * @returns True / False.
487      */
488     bool empty() const
489     {
490         return (currentSize == 0);
491     }
492
493     /**
494      * @fn  bool ExtendableArray::full() const
495      * @brief   evaluates if the array is full or not
496      *
497      * @returns True / False.
498      */
499     bool full() const
500     {
```

```
501
502         return (currentSize == capacity);
503     }
504
505     /**
506      * @fn  int ExtendableArray::size() const
507      * @brief   Gets the size
508      *
509      * @returns An int.
510      */
511     int size() const
512     {
513         return currentSize;
514     }
515
516     /**
517      * @fn  void ExtendableArray::insertAt(const int givenIndex, const Type&
         newItem)
518      * @brief   Inserts the value at the given index ( after adjustment )
519      *
520      * @exception   Except  Exception thrown when class is empty.
521      *
522      * @param   givenIndex  Zero-based index of the array.
523      * @param   newItem     The new value to insert.
524      */
525     void insertAt(const int givenIndex, const Type& newItem)
526     {
527         int adjIndex = adjustedIndex(givenIndex);
528
529         if (full())
530         {
531             expand();
532         }
533
534         if (empty())
535         {
536             throw Except("Array is empty", EMPTY, 5);
537         }
538         else
539         {
540             if (adjIndex <= (currentSize - currentSize / 2))
541             {
542                 shiftLeft_Outward(givenIndex);
543                 adjIndex--;
544             }
545             else
546             {
547                 shiftRight_Outward(givenIndex);
548             }
549
550             typeAr[adjIndex] = newItem;
551
```

```
552             currentSize++;
553 
554         }
555 
556     }
557 
558     /**
559      * @fn  template<class Type> void ExtendableArray::insertFront(const Type&
        newItem)
560      * @brief   Inserts the paramater value at the front of the array
561      *
562      * @tparam  Type    template datatype.
563      * @param   newItem The new item.
564      */
565     void insertFront(const Type& newItem)
566     {
567         if (full())
568         {
569             expand();
570         }
571 
572         if (empty())
573         {
574             frontIndex = 0;
575             endIndex = 0;
576         }
577         else if (frontIndex == 0)
578         {
579             frontIndex = capacity - 1;
580         }
581         else
582         {
583             frontIndex--;
584         }
585 
586         typeAr[frontIndex] = newItem;
587 
588         currentSize++;
589 
590     }
591 
592     /**
593      * @fn  template<class Type> void ExtendableArray::insertBack(const Type&
        newItem)
594      * @brief   Inserts the paramater value at the end of the array
595      *
596      * @tparam  Type    template datatype.
597      * @param   newItem The new item.
598      */
599     void insertBack(const Type& newItem)
600     {
601         if (full())
```

```
602         {
603             expand();
604         }
605 
606         if (empty())
607         {
608             frontIndex = 0;
609             endIndex = 0;
610         }
611         else if (endIndex == capacity - 1)
612         {
613             endIndex = 0;
614         }
615         else
616         {
617             endIndex++;
618         }
619 
620         typeAr[endIndex] = newItem;
621 
622         currentSize++;
623 
624 
625     }
626 
627     /**
628      * @fn  void ExtendableArray::eraseAt(const int givenIndex)
629      * @brief   Erases the value at the given index ( after adjustment )
630      *
631      * @exception   Except  Exception thrown when class is empty.
632      *
633      * @param   givenIndex  Zero-based index of the array.
634      */
635     void eraseAt(const int givenIndex)
636     {
637         int adjIndex = adjustedIndex(givenIndex);
638 
639         if (empty())
640         {
641             throw Except("Array is empty", EMPTY, 5);
642         }
643         else
644         {
645             if (adjIndex <= (currentSize - currentSize / 2))
646             {
647                 shiftLeft_Inward(givenIndex);
648             }
649             else
650             {
651                 shiftRight_Inward(givenIndex);
652             }
653 
```

```
654             currentSize--;
655 
656         }
657 
658     }
659 
660     /**
661      * @fn  void ExtendableArray::eraseFront()
662      * @brief   Erase the value at the front of the array
663      *
664      *          @par    Moves the frontIndex value to the right, excluding the value from the valid
665      *          range in the array
666      *
667      * @exception   Except  Thrown when an except error condition occurs.
668      */
669     void eraseFront()
670     {
671         if (empty())
672         {
673             throw Except("Array is empty", EMPTY, 5);
674         }
675         else
676         {
677             if (frontIndex == endIndex)
678             {
679                 frontIndex = -1;
680                 endIndex = -1;
681             }
682             else
683             {
684                 if (frontIndex == capacity - 1)
685                 {
686                     frontIndex = 0;
687                 }
688                 else
689                 {
690                     frontIndex++;
691                 }
692             }
693 
694             currentSize--;
695 
696         }
697     }
698 
699     /**
700      * @fn  void ExtendableArray::eraseBack()
701      * @brief   Erase the value at the end of the array
702      *
703      *          @par    Moves the endIndex value to the right, excluding the value from the valid range
```

```
704         *           in the array
705         *
706         * @exception   Except  Thrown when an except error condition occurs.
707         */
708        void eraseBack()
709        {
710            if (empty())
711            {
712                throw Except("Array is empty", EMPTY, 5);
713            }
714            else
715            {
716                if (frontIndex == endIndex)
717                {
718                    frontIndex = -1;
719                    endIndex = -1;
720                }
721                else if (endIndex == 0)
722                {
723                    endIndex = capacity - 1;
724                }
725                else
726                {
727                    endIndex--;
728                }
729
730                currentSize--;
731            }
732        }
733
734        /**
735         * @fn  Type ExtendableArray::front() const
736         * @brief   Gets the value at the front of the array
737         *
738         * @exception   Except  Exception thrown when class is empty.
739         *
740         * @returns &lt;Type&gt; the front index value.
741         */
742        Type front() const
743        {
744            if (empty())
745            {
746                throw Except("Array is empty - nothing to return", EMPTY, 5);
747            }
748            else
749            {
750                return typeAr[frontIndex];
751            }
752        }
753
754        /**
755         * @fn  Type ExtendableArray::back() const
```

```
756       * @brief   Gets the value at the end of the array
757       *
758       * @exception   Except  Exception thrown when class is empty.
759       *
760       * @returns &lt;Type&gt; the end index value.
761       */
762      Type back() const
763      {
764          if (empty())
765          {
766              throw Except("Array is empty - nothing to return", EMPTY, 5);
767          }
768          else
769          {
770              return typeAr[endIndex];
771          }
772      }
773
774      /**
775       * @fn  Type& ExtendableArray::at(const int index) const
776       * @brief   Accesses the value at the given index ( after adjustment )
777       *
778       * @param   index   Zero-based index of the array.
779       * @returns &lt;Type&gt; the index value.
780       */
781      Type& at(const int index) const
782      {
783          return typeAr[adjustedIndex(index)];
784      }
785
786      /**
787       * @fn  Type& ExtendableArray::operator[](const int index) const
788       * @brief   Array indexer operator ( adjusted inside )
789       *
790       * @param   index   Zero-based index of the.
791       * @returns &lt;Type&gt; the index value.
792       */
793      Type& operator[](const int index) const
794      {
795          return typeAr[adjustedIndex(index)];
796      }
797
798      /**
799       * @fn  void ExtendableArray::printAll(std::ostream& output) const
800       * @brief   Print all values in the valid array range line by line
801       *
802       * @exception   Except  Exception thrown when class is empty.
803       *
804       * @param [in,out]  output  ostream object ( cout, file, etc. )
805       */
806      void printAll(std::ostream& output, const std::string& label) const
807      {
```

```
808         if (empty())
809         {
810             throw Except("Array is empty - nothing to print", EMPTY, 5);
811         }
812         else
813         {
814             int tempIndex = frontIndex;
815 
816             output << '\n' << label << '\n';
817 
818             for (int i = 0; i < currentSize; i++)
819             {
820                 output << typeAr[tempIndex] << '\n';
821 
822                 if (tempIndex == capacity - 1)
823                 {
824                     tempIndex = 0;
825                 }
826                 else
827                 {
828                     tempIndex++;
829                 }
830             }
831         }
832     }
833 
834     /**
835      * @fn  void ExtendableArray::oneLinePrintAdjusted(std::ostream& output,
          const std::string& label) const
836      * @brief   Print all values in the valid array, adjusted, in one line
837      *
838      * @exception   Except  Exception thrown when class is empty.
839      *
840      * @param [in,out]  output  ostream object ( cout, file, etc. )
841      * @param           label   The label.
842      */
843     void oneLinePrintAdjusted(std::ostream& output, const std::string& label)
          const
844     {
845         if (empty())
846         {
847             throw Except("Array is empty - nothing to print", EMPTY, 5);
848         }
849         else
850         {
851             int tempIndex = frontIndex;
852 
853             output << label;
854 
855             for (int i = 0; i < currentSize; i++)
856             {
857                 if (i != currentSize - 1)
```

```
858                 {
859                     output << typeAr[tempIndex] << ", ";
860                 }
861                 else
862                 {
863                     output << typeAr[tempIndex] << '\n';
864                 }
865 
866                 if (tempIndex == capacity - 1)
867                 {
868                     tempIndex = 0;
869                 }
870                 else
871                 {
872                     tempIndex++;
873                 }
874             }
875         }
876     }
877 
878     /**
879      * @fn  void ExtendableArray::PrintRealArray(const std::string& label,
          std::ostream& output) const
880      * @brief   Print all values in the real array range [ 0 .. capacity ] line
          by line.
881      *          Also indicates the first and last indeces
882      *
883      * @exception   Except  Exception thrown when class is empty.
884      *
885      * @param           label   The label.
886      * @param [in,out]  output  ostream object ( cout, file, etc. )
887      */
888     void PrintRealArray(const std::string& label, std::ostream& output) const
889     {
890         if (empty())
891         {
892             throw Except("Array is empty - nothing to print", EMPTY, 5);
893         }
894         else
895         {
896             int tempIndex = frontIndex;
897 
898             output << '\n' << label << '\n';
899             for (int i = 0; i < capacity; i++)
900             {
901                 output << "[ " << i << " ] = " << typeAr[i];
902                 if (i == frontIndex)
903                 {
904                     output << "   <- First index";
905                 }
906                 else if (i == endIndex)
907                 {
```

```
908                     output << "   <- Last index";
909                 }
910                 output << '\n';
911             }
912         }
913     }
914
915
916
917
918 #endif // !Public_Methods //-------------------------------------------------
919
920 };
921 #endif // !_EXTENDABLEARRAY_H_
922 //------------------------------------------------------------------------------ ↵
      ----------
923
```