

```

1 *****
2 * PROGRAMMED BY : Nick Reardon
3 * CLASS          : CS1D
4 * SECTION        : MW - 2:30p
5 * Assignment #4 : Deque To Queue
6 *****

```

Assignment #4 - Deque To Queue

Implement a Queue interface with a class that is based on a Deque using a wrapper. Do not use the STL. Highlight your Deque and Queue classes.

Queue method	Deque Implementation
size()	size()
empty()	empty()
front()	front()
enqueue()	insertBack()
dequeue()	eraseFront()

Test all the member functions (size(), empty(), front(), enqueue, and dequeue) of the queue class with the following data.

```

32 String string1 = "A man, a plan, a canal, Panamaö;
33 String string2 = ôWas it a car or a cat I saw?";
34 String string3 = ôSit on a potato pan, Otis";
35 String string4 = ôNo lemon, no melon";

```

Due February 10th

```

41 *****

```

--- Using Deque implementation ---

deque currently empty - empty() method called in printAll() to show

Printing deque:

Container is empty

Reading into deque from file using insertBack() method

```
53 Printing deque:
54 A man, a plan, a canal, Panama
55 Was it a car or a cat I saw?
56 Sit on a potato pan, Otis
57 No lemon, no melon
58
59 deque - calling size method
60 Size is currently: 4
61
62 deque - Calling front() method
63 front value is currently: A man, a plan, a canal, Panama
64
65 deque - Calling eraseFront() method
66 front value is now currently: Was it a car or a cat I saw?
67
68 deque - Calling eraseFront() method
69
70 Printing deque:
71 Sit on a potato pan, Otis
72 No lemon, no melon
73
74
75 --- Using Queue wrapper of deque implementation ---
76
77 Queue currently empty - empty() method called in printAll() to show
78
79 Printing Queue:
80 Container is empty
81
82 Reading into queue from file using enqueue() method
83
84 Printing Queue:
85 A man, a plan, a canal, Panama
86 Was it a car or a cat I saw?
87 Sit on a potato pan, Otis
88 No lemon, no melon
89
90 queue - calling size method
91 Size is currently: 4
92
93 queue - Calling front() method
94 front value is currently: A man, a plan, a canal, Panama
95
96 queue - Calling dequeue() method
97 front value is now currently: Was it a car or a cat I saw?
98
99 queue - Calling dequeue() method
100
101 Printing Queue:
102 Sit on a potato pan, Otis
103 No lemon, no melon
104
```

105 Press any key to continue . . .

```
1  /*****
2  * AUTHOR          : Nick Reardon
3  * Assignment #4   : Deque To Queue
4  * CLASS           : CS1D
5  * SECTION         : MW - 2:30p
6  * DUE DATE        : 02 / 10 / 20
7  *****/
8  #ifndef _MAIN_H_
9  #define _MAIN_H_
10
11 //Standard includes
12 #include <iostream>
13 #include <iomanip>
14 #include <string>
15 #include "PrintHeader.h"
16
17 //Program Specific
18 #include "LinkedDeque.h"
19
20
21
22 #endif // _HEADER_H_
```

```
1  /*****
2  * AUTHOR          : Nick Reardon
3  * Assignment #4   : Deque To Queue
4  * CLASS           : CS1D
5  * SECTION         : MW - 2:30p
6  * DUE DATE        : 02 / 10 / 20
7  *****/
8  #include "main.h"
9
10 using std::cout; using std::endl;
11
12
13 int main()
14 {
15     /*
16     * HEADER OUTPUT
17     */
18     PrintHeader(cout, "Prompt.txt");
19
20     /*****/
21
22     LinkedDeque<std::string> deque;
23     LinkedQueue<std::string> queue;
24
25     std::ifstream iFile;
26
27     cout << endl << " --- Using Deque implementation ---" << endl << endl;
28
29
30
31     cout << "deque currently empty - empty() method called in printAll() to show" << endl;
32     PrintWithLabel("Printing deque:", deque, cout);
33
34     //-----
35
36     cout << "Reading into deque from file using insertBack() method" << endl;
37
38     std::string temp;
39     iFile.open("Input.txt");
40
41     while ( getline(iFile, temp) )
42     {
43         deque.insertBack(temp);
44     }
45     iFile.close();
46
47     PrintWithLabel("Printing deque:", deque, cout);
48
49     //-----
50
51     cout << "deque - calling size method" << endl;
```

```
52     cout << "Size is currently: " << deque.size() << endl << endl;
53
54     //-----
55
56     cout << "deque - Calling front() method" << endl;
57     cout << "front value is currently: " << deque.front() << endl << endl;
58
59     //-----
60
61     cout << "deque - Calling eraseFront() method" << endl;
62     deque.eraseFront();
63     cout << "front value is now currently: " << deque.front() << endl << endl;
64
65     cout << "deque - Calling eraseFront() method" << endl;
66     deque.eraseFront();
67     PrintWithLabel("Printing deque:", deque, cout);
68
69     //-----
70     //-----
71
72     cout << endl << " --- Using Queue wrapper of deque implementation ---" <<  ↗
       endl << endl;
73
74
75     cout << "Queue currently empty - empty() method called in printAll() to show"  ↗
       << endl;
76     PrintWithLabel("Printing Queue:", queue, cout);
77
78     //-----
79
80     cout << "Reading into queue from file using enqueue() method" << endl;
81
82     temp.clear();
83     iFile.open("Input.txt");
84
85     while (getline(iFile, temp))
86     {
87         queue.enqueue(temp);
88     }
89     iFile.close();
90
91     PrintWithLabel("Printing Queue:", queue, cout);
92
93     //-----
94
95     cout << "queue - calling size method" << endl;
96     cout << "Size is currently: " << queue.size() << endl << endl;
97
98     //-----
99
100    cout << "queue - Calling front() method" << endl;
101    cout << "front value is currently: " << queue.front() << endl << endl;
```

```
102
103     //-----
104
105     cout << "queue - Calling dequeue() method" << endl;
106     queue.dequeue();
107     cout << "front value is now currently: " << queue.front() << endl << endl;
108
109     cout << "queue - Calling dequeue() method" << endl;
110     queue.dequeue();
111     PrintWithLabel("Printing Queue:", queue, cout);
112
113
114     system("pause");
115     return 0;
116 }
117
118
```

```

1  /*****
2  * AUTHOR          : Nick Reardon
3  * Assignment #4   : Deque To Queue
4  * CLASS           : CS1D
5  * SECTION        : MW - 2:30p
6  * DUE DATE       : 02 / 10 / 20
7  *****/
8  #ifndef _LINKEDDEQUE_H_
9  #define _LINKEDDEQUE_H_
10 #include <exception>
11 #include <sstream>
12
13
14
15 class Except : virtual public std::runtime_error {
16
17 protected:
18
19     int error_number;          ///< Error number
20     int error_offset;          ///< Error offset
21
22 public:
23
24     /** Constructor (C++ STL string, int, int).
25     * @param msg      The error message
26     * @param err_num  Error number
27     * @param err_off  Error offset
28     */
29     explicit
30     Except(const std::string& msg, int err_num, int err_off) :
31         std::runtime_error(msg)
32     {
33         error_number = err_num;
34         error_offset = err_off;
35     }
36
37
38     /** Destructor.
39     * Virtual to allow for subclassing.
40     */
41     virtual ~Except() throw () {}
42
43     /** Returns error number.
44     * @return #error_number
45     */
46     virtual int getErrorNumber() const throw() {
47         return error_number;
48     }
49
50     /**Returns error offset.
51     * @return #error_offset
52     */

```



```
53     virtual int getErrorOffset() const throw() {
54         return error_offset;
55     }
56
57 };
58
59 enum ERROR_TYPE
60 {
61     DEFAULT,
62     FULL,
63     EMPTY,
64     OUT_OF_RANGE
65 };
66
67 template <class Type>
68 struct Node
69 {
70     Type value;
71
72     Node<Type>* prev;
73     Node<Type>* next;
74
75     Node<Type>(const Type& newValue, Node<Type>* prevNode, Node<Type>* nextNode)
76     {
77         value = newValue;
78
79         prev = prevNode;
80         next = nextNode;
81     }
82
83 };
84
85 template <class Type>
86 class LinkedDeque
87 {
88 private:
89     Node<Type>* head;
90     Node<Type>* tail;
91
92     int capacity;
93     int currentSize;
94
95
96 protected:
97
98 public:
99
100     LinkedDeque<Type>(const int newCapacity = 32);
101
102     LinkedDeque<Type>(const LinkedDeque<Type>& otherDeque);
103
104     ~LinkedDeque();
```

```
105
106     void destroy();
107
108     bool empty() const;
109     bool full() const;
110     int size() const;
111
112     void insertBefore(const Type& newItem, const int index);
113     void insertAfter(const Type& newItem, const int index);
114
115     void insertFront(const Type& newItem);
116     void insertBack(const Type& newItem);
117
118     void eraseFront();
119     void eraseBack();
120
121
122     Type front() const;
123     Type back() const;
124
125     void printAll(std::ostream& output) const;
126
127 };
128
129
130 template<class Type>
131 inline LinkedDeque<Type>::LinkedDeque(const int newCapacity)
132 {
133     head = nullptr;
134     tail = nullptr;
135
136     capacity = newCapacity;
137     currentSize = 0;
138 }
139
140 template<class Type>
141 inline LinkedDeque<Type>::LinkedDeque(const LinkedDeque<Type>& otherDeque)
142 {
143     capacity = otherDeque.capacity;
144     currentSize = 0;
145
146     head = nullptr;
147     tail = nullptr;
148
149     for (Node<Type>* temp = otherDeque.head; temp != nullptr; temp = temp->next)
150     {
151         insertBack(temp->value);
152     }
153 }
154
155
156 template<class Type>
```

```
157 inline LinkedDeque<Type>::~~LinkedDeque()
158 {
159     destroy();
160 }
161
162 template<class Type>
163 inline bool LinkedDeque<Type>::empty() const
164 {
165     bool value = (currentSize == 0 && head == nullptr && tail == nullptr);
166
167     return value;
168 }
169
170 template<class Type>
171 inline bool LinkedDeque<Type>::full() const
172 {
173     bool value = (currentSize == capacity);
174
175     return value;
176 }
177
178 template<class Type>
179 inline void LinkedDeque<Type>::destroy()
180 {
181     for (Node<Type>* temp = head; temp != nullptr; )
182     {
183         Node<Type>* hold = temp;
184         temp = temp->next;
185         delete hold;
186     }
187
188     head = nullptr;
189     tail = nullptr;
190 }
191
192
193 template<class Type>
194 inline void LinkedDeque<Type>::insertBefore(const Type& newItem, const int index)
195 {
196     if (empty())
197     {
198         throw Except("container is empty", EMPTY, 5);
199     }
200     else if (full())
201     {
202         throw Except("container is full", FULL, 5);
203     }
204     else
205     {
206         if (index <= (size() - 1 / 2))
207         {
208             Node<Type>* ptr = head;
```

```
209         for (int i = 0; i < index; i++)
210         {
211             ptr = ptr->next;
212         }
213         Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
214
215         temp->prev = ptr->prev;
216         temp->next = ptr;
217
218         ptr->prev = temp;
219         temp->prev->next = temp;
220
221         temp = nullptr;
222         ptr = nullptr;
223
224         currentSize++;
225     }
226     else
227     {
228         Node<Type>* ptr = head;
229         for (int i = size() - 1; i >= size() - 1 - index; i--)
230         {
231             ptr = ptr->next;
232         }
233         Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
234
235         temp->prev = ptr->prev;
236         temp->next = ptr;
237
238         ptr->prev = temp;
239         temp->prev->next = temp;
240
241         temp = nullptr;
242         ptr = nullptr;
243
244         currentSize++;
245     }
246 }
247
248 }
249
250 template<class Type>
251 inline void LinkedDeque<Type>::insertAfter(const Type& newItem, const int index)
252 {
253     if (empty())
254     {
255         throw Except("container is empty", EMPTY, 5);
256     }
257     else if (full())
258     {
259         throw Except("container is full", FULL, 5);
260     }
```

```
261     else
262     {
263         if (index <= (size() - 1 / 2))
264         {
265             Node<Type>* ptr = head;
266             for (int i = 1; i <= index; i++)
267             {
268                 ptr = ptr->next;
269             }
270             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
271
272             temp->next = ptr->next;
273             ptr->next = temp;
274
275             temp->prev = ptr;
276             temp->next->prev = temp;
277
278             temp = nullptr;
279             ptr = nullptr;
280
281             currentSize++;
282         }
283     else
284     {
285         Node<Type>* ptr = head;
286         for (int i = size() - 1; i >= size() - 1 - index; i--)
287         {
288             ptr = ptr->next;
289         }
290         Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
291
292         temp->next = ptr->next;
293         ptr->next = temp;
294
295         temp->prev = ptr;
296         temp->next->prev = temp;
297
298         temp = nullptr;
299         ptr = nullptr;
300
301         currentSize++;
302     }
303 }
304
305 }
306
307 template<class Type>
308 inline void LinkedDeque<Type>::insertFront(const Type& newItem)
309 {
310
311     if (!full())
312     {
```

```
313     if (head == 0)
314     {
315         head = new Node<Type>(newItem, nullptr, nullptr);
316         tail = head;
317     }
318     else
319     {
320         Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
321
322         head->prev = temp;
323
324         temp->next = head;
325         temp->prev = nullptr;
326
327         head = temp;
328
329         temp = nullptr;
330     }
331     currentSize++;
332 }
333 }
334 else
335 {
336     throw Except("container is full", FULL, 5);
337 }
338 }
339
340 template<class Type>
341 inline void LinkedDeque<Type>::insertBack(const Type& newItem)
342 {
343     if (!full())
344     {
345         if (head == 0)
346         {
347             head = new Node<Type>(newItem, nullptr, nullptr);
348             tail = head;
349         }
350         else
351         {
352             Node<Type>* temp = new Node<Type>(newItem, nullptr, nullptr);
353
354             tail->next = temp;
355
356             temp->prev = tail;
357             temp->next = nullptr;
358
359             tail = temp;
360
361             temp = nullptr;
362         }
363     }
364     currentSize++;
```

```
365     }
366     else
367     {
368         throw Except("container is full", FULL, 5);
369     }
370 }
371 }
372 }
373
374 template<class Type>
375 inline void LinkedDeque<Type>::eraseFront()
376 {
377     if (!empty())
378     {
379         if (head == tail)
380         {
381             delete head;
382             head = nullptr;
383             tail = nullptr;
384         }
385         else
386         {
387             Node<Type>* temp = head;
388             head = head->next;
389             head->prev = nullptr;
390             delete temp;
391         }
392         currentSize--;
393     }
394     else
395     {
396         throw Except("container is empty", EMPTY, 5);
397     }
398 }
399
400 }
401 }
402
403 template<class Type>
404 inline void LinkedDeque<Type>::eraseBack()
405 {
406     if (!empty())
407     {
408         if (head == tail)
409         {
410             delete head;
411             head = nullptr;
412             tail = nullptr;
413         }
414         else
415         {
416             Node<Type>* temp = tail;
```

```
417         tail = tail->prev;
418         tail->next = nullptr;
419         delete temp;
420     }
421     currentSize--;
422
423 }
424 else
425 {
426     throw Except("container is empty", EMPTY, 5);
427 }
428
429 }
430
431 template<class Type>
432 inline int LinkedDeque<Type>::size() const
433 {
434     return currentSize;
435 }
436
437
438 template<class Type>
439 inline Type LinkedDeque<Type>::front() const
440 {
441     return head->value;
442 }
443
444 template<class Type>
445 inline Type LinkedDeque<Type>::back() const
446 {
447     return tail->value;
448 }
449
450 template<class Type>
451 inline void LinkedDeque<Type>::printAll(std::ostream& output) const
452 {
453     for (Node<Type>* temp = head; temp != nullptr; temp = temp->next)
454     {
455         output << temp->value << '\n';
456     }
457 }
458
459
460
461 template<class Type>
462 class LinkedQueue : protected LinkedDeque<Type>
463 {
464 private:
465     LinkedDeque<Type> deque;
466
467 protected:
468
```



```
469 public:
470
471     LinkedQueue<Type>(const int newCapacity = 32);
472
473     LinkedQueue<Type>(LinkedQueue<Type>& otherQueue);
474     LinkedQueue<Type>(LinkedDeque<Type>& otherDeque);
475
476     ~LinkedQueue();
477
478     void destroy();
479
480     bool empty() const;
481     bool full() const;
482     int size() const;
483
484     void enqueue(const Type& newItem);
485     void dequeue();
486
487     Type front() const;
488     Type back() const;
489
490     void printAll(std::ostream& output) const;
491
492 };
493
494
495
496 #endif // !_LINKEDDEQUE_H_
497
498 template<class Type>
499 inline LinkedQueue<Type>::LinkedQueue(const int newCapacity) :
500     LinkedDeque<Type>(newCapacity)
501 {
502
503 }
504
505 template<class Type>
506 inline LinkedQueue<Type>::LinkedQueue(LinkedQueue<Type>& otherQueue) :
507     LinkedDeque<Type>(otherQueue)
508 {
509
510 }
511
512 template<class Type>
513 inline LinkedQueue<Type>::LinkedQueue(LinkedDeque<Type>& otherDeque) :
514     LinkedDeque<Type>(otherDeque)
515 {
516
517 }
518 template<class Type>
519 inline void LinkedQueue<Type>::destroy()
520 {
```

```
521     deque.destroy();
522 }
523
524 template<class Type>
525 inline LinkedQueue<Type>::~~LinkedQueue()
526 {
527     deque.destroy();
528 }
529
530 template<class Type>
531 inline bool LinkedQueue<Type>::empty() const
532 {
533     return deque.empty();
534 }
535
536 template<class Type>
537 inline bool LinkedQueue<Type>::full() const
538 {
539     return deque.full();
540 }
541
542 template<class Type>
543 inline int LinkedQueue<Type>::size() const
544 {
545     return deque.size();
546 }
547
548 template<class Type>
549 inline void LinkedQueue<Type>::enqueue(const Type& newItem)
550 {
551     deque.insertBack(newItem);
552 }
553
554 template<class Type>
555 inline void LinkedQueue<Type>::dequeue()
556 {
557     deque.eraseFront();
558 }
559
560 template<class Type>
561 inline Type LinkedQueue<Type>::front() const
562 {
563     return deque.front();
564 }
565
566 template<class Type>
567 inline Type LinkedQueue<Type>::back() const
568 {
569     return deque.back();
570 }
571
572 template<class Type>
```

```
573 inline void LinkedQueue<Type>::printAll(std::ostream& output) const
574 {
575     deque.printAll(output);
576 }
577
578
579
580 template<class Type>
581 void PrintWithLabel(const std::string& label, const LinkedDeque<Type> &container,
582                    std::ostream& output)
583 {
584     output << '\n' << label << '\n';
585     if (container.empty())
586     {
587         output << "Container is empty \n\n";
588     }
589     else
590     {
591         container.printAll(output);
592         output << '\n';
593     }
594 }
595
596
597 template<class Type>
598 void PrintWithLabel(const std::string& label, const LinkedQueue<Type>& container,
599                    std::ostream& output)
600 {
601     output << '\n' << label << '\n';
602     if (container.empty())
603     {
604         output << "Container is empty \n\n";
605     }
606     else
607     {
608         container.printAll(output);
609         output << '\n';
610     }
611 }
612 }
```

```

1  /*****
2  * AUTHOR          : Nick Reardon
3  * Assignment #4   : Deque To Queue
4  * CLASS           : CS1D
5  * SECTION         : MW - 2:30p
6  * DUE DATE        : 02 / 10 / 20
7  *****/
8  #ifndef _PRINTHEADER_H_
9  #define _PRINTHEADER_H_
10
11 #include <iostream>
12 #include <iomanip>
13 #include <ostream>
14 #include <string>
15 #include <fstream>
16
17 /*****
18 * PrintHeader
19 * -----
20 * This function will output a class header through the use of ostream.
21 * It also will output the program description
22 * -----
23 * Call
24 * -----
25 * The function call requires 1 parameters. The following example uses an
26 * output file in the ostream parameter. Ex:
27 *
28 *     PrintHeader (oFile);
29 *
30 * -----
31 * Output
32 * -----
33 * The function will output as follows. Ex:
34 *
35 *     *****/
36 *     * PROGRAMMED BY : Parsa Khazravi and Nick Reardon
37 *     * CLASS          : CS1B
38 *     * SECTION        : MW: 7:30pm
39 *     * Lab #3         : Functions - GCD
40 *     *****/
41 *
42 * -----
43 * CONSTANTS
44 * -----
45 * OUTPUT - USED FOR CLASS HEADING
46 * -----
47 * PROGRAMMER       : Name(s) of programmer(s) - Nick Reardon
48 * SECTION          : Class times - MW - 7:30p
49 * CLASS            : Class label - CS1B
50 * PROGRAM_NUM      : # of the program
51 * PROGRAM_NAME     : Title of the program
52 * PROGRAM_TYPE     : Type of program - Lab, Assignment, etc.

```

```

53 *
54 * -----
55 * MAX_OUTPUT      : Max movies to be output at once
56 *****/
57 const std::string PROGRAMMER = "Nick Reardon";
58 const std::string SECTION = "MW - 2:30p";
59 const std::string CLASS = "CS1D";
60 const int PROGRAM_NUM = 4;
61 const std::string PROGRAM_NAME = "Deque To Queue";
62 const std::string PROGRAM_TYPE = "Assignment";
63
64
65 void PrintHeader(std::ostream &output, std::string inputText)
66 {
67     std::string typeNum = PROGRAM_TYPE + " #" + std::to_string(PROGRAM_NUM);
68
69     output << std::left
70         << std::string(76, '*')
71         << std::endl
72         << "PROGRAMMED BY : " << PROGRAMMER << std::endl
73         << " " << std::setw(14) << "CLASS" << ": " << CLASS << std::endl
74         << " " << std::setw(14) << "SECTION" << ": " << SECTION << std::endl
75         << " " << std::setw(14) << typeNum << ": " << PROGRAM_NAME << std::endl
76         << std::string(76, '*')
77         << std::endl << std::endl
78         << std::string(((76 - typeNum.length() - PROGRAM_NAME.length() ) / 2), ' ')
79         << typeNum + " - " + PROGRAM_NAME
80         << std::endl << std::endl
81         << std::ifstream(inputText).rddbuf()
82         << std::endl
83         << std::string(76, '*')
84         << std::endl << std::endl;
85
86 }
87
88 #endif //_PRINTHEADER_H_

```