

```
1 // Shaun Chemplavil U08713628
2 // shaun.chemplavil@gmail.com
3 // C/C++ Programming IV : Advanced Programming with Objects
4 // 152488 Raymond L. Mitchell III
5 // hw2.cpp
6 // Win10
7 // Visual C++ 19.0
8 //
9
10 #include <iostream>
11 #include <cassert>
12 #include <exception>
13
14 using namespace std;
15
16 template <typename T>
17 class Queue
18 {
19 public:
20     Queue(); // Construct empty queue
21     ~Queue(); // Destructor
22     Queue(const Queue &); // Copy constructor
23     Queue &operator=(const Queue &); // Copy assignment operator
24     void push(const T &); // Add elem to back of queue
25     void pop(); // Remove front elem from queue
26     T &front(); // Return ref to front elem in queue
27     const T &front() const; // Return ref to front elem in queue
28     bool empty() const; // Return whether queue is empty
29     size_t size() const; // Return # of elems in queue
30 private:
31     T *v_; // Elems in queue
32     size_t vsize_;
33     size_t vused_;
34 };
35 // Default Constructor
36 template <typename T>
37 Queue<T>::Queue() : v_(0), vsize_(10), vused_(0)
38 {
39     v_ = new T[vsize_]; // Initial Allocation
40 }
41
42 // Destructor
43 template <typename T>
44 Queue<T>::~~Queue()
45 {
46     delete[] v_;
47 }
48
49 // Helper Function for copying array
50 template <typename T>
51 T*
52 newCopy(const T *src, size_t srcsize, size_t destsize)
```

```
53 {
54     assert(destsize >= srcsize);
55     T *dest = new T[destsize];
56     try
57     {
58         copy(src, src + srcsize, dest);
59     }
60     catch (...)
61     {
62         delete[] dest;
63         throw;
64     }
65     return dest;
66 }
67
68 // Copy Constructor
69 template <typename T>
70 Queue<T>::Queue(const Queue<T> &other)
71     :v_(newCopy(other.v_, other.vsize_, other.vsize_)),
72     vsize_(other.vsize_), vused_(other.vused_)
73 {
74 }
75 // Copy Assignment
76 template <typename T>
77 Queue<T> &
78 Queue<T>::operator=(const Queue<T> &other)
79 {
80     if (this != &other)
81     {
82         T *v_new = newCopy(other.v_, other.vsize_, other.vsize_);
83         delete[] v_;
84         v_ = v_new;
85         vsize_ = other.vsize_;
86         vused_ = other.vused_;
87     }
88
89     return *this;
90 }
91
92 // Push to the back of Queue
93 template <typename T>
94 void
95 Queue<T>::push(const T &t)
96 {
97     // if necessary grow size
98     if (vused_ == vsize_)
99     {
100         size_t vsize_new = vsize_ * 2 + 1;
101         T *v_new = newCopy(v_, vsize_, vsize_new);
102         delete[] v_;
103         v_ = v_new;
104         vsize_ = vsize_new;
```

```
105     }
106     // add element to back of queue
107     v_[vused_] = t;
108
109     // increase only after element copy
110     ++vused_;
111 }
112
113 // Pop from the front of Queue
114 template <typename T>
115 void
116 Queue<T>::pop()
117 {
118     if (vused_ == 0)
119     {
120         throw logic_error("pop from empty Queue");
121     }
122     else
123     {
124         // only copy (vused_-1) elements during a "pop"
125         size_t vsize_new = vused_ - 1;
126         // copy all but the first element into v_new
127         T *v_new = newCopy(v_ + 1, vsize_new, vsize_);
128
129         delete[] v_;
130         // replace private member variable with "popped" array
131         v_ = v_new;
132         // decrease only after element removal
133         --vused_;
134     }
135 }
136
137 // Return reference to Front element
138 template <typename T>
139 T &
140 Queue<T>::front()
141 {
142     if (vused_ == 0)
143         throw logic_error("front from empty Queue");
144     else
145         return v_[0];
146 }
147
148 // Return reference to Front element
149 template <typename T>
150 const T &
151 Queue<T>::front() const
152 {
153     if (vused_ == 0)
154         throw logic_error("front from empty Queue");
155     else
156         return v_[0];
```

```
157 }
158
159 // Check if Queue is empty
160 template <typename T>
161 bool
162 Queue<T>::empty() const
163 {
164     return (vused_ == 0);
165 }
166
167 // Return number of elements in Queue
168 template <typename T>
169 size_t
170 Queue<T>::size() const
171 {
172     return (vused_);
173 }
174
175 // Unit Tests:
176 void testQueueConstructor()
177 {
178     try
179     {
180         Queue<int> tempQueue;
181         clog << "testQueueConstructor PASSED\n";
182     }
183     catch (...)
184     {
185         clog << "testQueueConstructor FAILED\n";
186     }
187 }
188 void testQueueDestructor()
189 {
190     Queue<int> *testQueue = new Queue<int>;
191     try
192     {
193         delete testQueue;
194         clog << "testQueueDestructor PASSED\n";
195     }
196     catch (...)
197     {
198         Queue<int> tempQueue;
199         clog << "testQueueDestructor FAILED\n";
200     }
201 }
202
203 void testQueueCopyConstructor()
204 {
205     Queue<int> sourceQueue;
206     const int VALID_INPUT = 42;
207     const int NUM_ELEMENTS = 4;
208     for (int idx = 0; idx < NUM_ELEMENTS; idx++)
```

```
209     sourceQueue.push(VALID_INPUT * idx + 1);
210     try
211     {
212         Queue<int> copyQueue(sourceQueue);
213
214         for (int idx = 0; idx < NUM_ELEMENTS; idx++)
215         {
216             if (sourceQueue.front() == copyQueue.front())
217             {
218                 sourceQueue.pop(); copyQueue.pop();
219             }
220             else
221             {
222                 clog << "testQueueCopyConstructor FAILED : Expected "
223                     << sourceQueue.front() << " instead saw "
224                     << copyQueue.front() << "\n";
225             }
226         }
227         clog << "testQueueCopyConstructor PASSED\n";
228     }
229     catch (...)
230     {
231         clog << "testQueueCopyConstructor FAILED\n";
232     }
233 }
234
235 void testQueueCopyAssignment()
236 {
237     Queue<int> sourceQueue;
238     const int VALID_INPUT = 42;
239     const int NUM_ELEMENTS = 4;
240     for (int idx = 0; idx < NUM_ELEMENTS; idx++)
241         sourceQueue.push(VALID_INPUT * idx + 1);
242     try
243     {
244         Queue<int> copyQueue = sourceQueue;
245
246         for (int idx = 0; idx < NUM_ELEMENTS; idx++)
247         {
248             if (sourceQueue.front() == copyQueue.front())
249             {
250                 sourceQueue.pop(); copyQueue.pop();
251             }
252             else
253             {
254                 clog << "testQueueCopyAssignment FAILED : Expected "
255                     << sourceQueue.front() << " instead saw " <<
256                     << copyQueue.front() << "\n";
257             }
258         }
259         clog << "testQueueCopyAssignment PASSED\n";
260     }
```

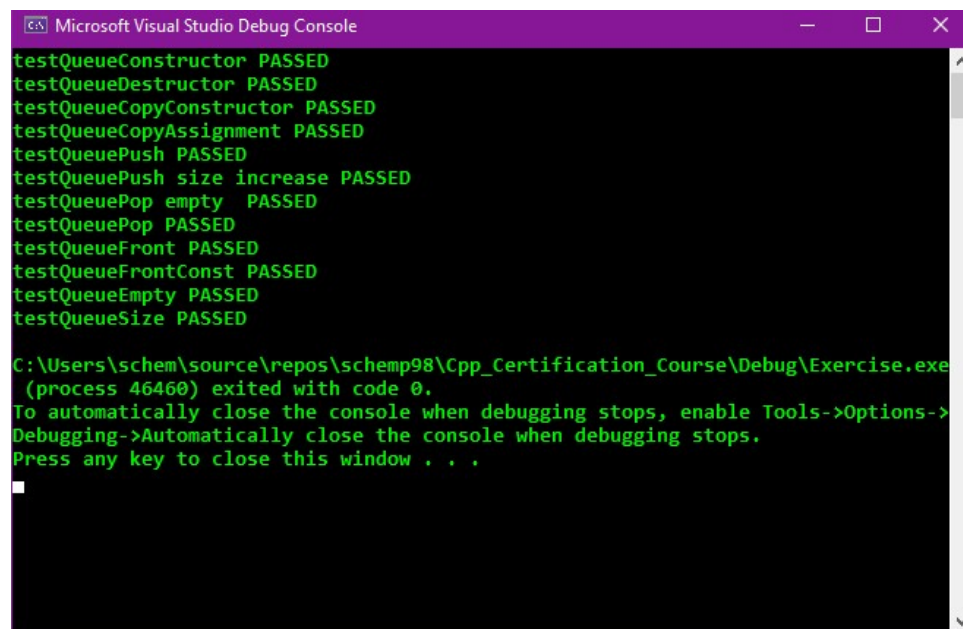
```
261     catch (...)
262     {
263         clog << "testQueueCopyAssignment FAILED\n";
264     }
265 }
266
267 void testQueuePush()
268 {
269     Queue<int> tempQueue;
270     try
271     {
272         const int INPUT = 1;
273         const size_t ORG_SIZE = 10; // one more than
274         const size_t NEW_SIZE = 11; // one more than
275
276         for (int idx = 0; idx < ORG_SIZE; ++idx)
277             tempQueue.push(idx * INPUT);
278
279         clog << "testQueuePush PASSED\n";
280
281         // Add one more than initial size allocation
282         tempQueue.push(INPUT);
283
284         // Check if size increase to expected value
285         if (tempQueue.size() == NEW_SIZE)
286             clog << "testQueuePush size increase PASSED\n";
287         else
288             clog << "testQueuePush size increase FAILED : Expected size "
289                 << NEW_SIZE << " instead saw " << tempQueue.size() << "\n";
290     }
291     catch (...)
292     {
293         clog << "testQueuePush FAILED\n";
294     }
295 }
296
297 void testQueuePop()
298 {
299     Queue<int> tempQueue;
300
301     // test an empty pop
302     try
303     {
304         tempQueue.pop();
305
306         clog << "testQueuePop empty FAILED\n";
307     }
308     catch (exception &e)
309     {
310         if (strcmp(e.what(), "pop from empty Queue") == 0)
311             clog << "testQueuePop empty PASSED\n";
312         else
```

```
313         clog << "testQueuePop empty FAILED expected different error message\n";
314     }
315
316     //test successful pop
317     try
318     {
319         const int FIRST_INPUT = 1;
320         const int SECOND_INPUT = 2;
321         tempQueue.push(FIRST_INPUT);
322         tempQueue.push(SECOND_INPUT);
323
324         if (tempQueue.front() == FIRST_INPUT)
325             tempQueue.pop();
326
327         // Check to see if FIRST_INPUT was successfully popped off
328         if (tempQueue.front() == SECOND_INPUT)
329             clog << "testQueuePop PASSED\n";
330     }
331     catch (...)
332     {
333         clog << "testQueuePop FAILED\n";
334     }
335 }
336
337 void testQueueFront()
338 {
339     Queue<int> tempQueue;
340     const int VALID_VALUE = 1;
341     tempQueue.push(VALID_VALUE);
342     try
343     {
344         const int *TEST_VALUE = &(tempQueue.front());
345
346         if (*TEST_VALUE == VALID_VALUE)
347             clog << "testQueueFront PASSED\n";
348         else
349             clog << "testQueueFront FAILED : Expected value "
350                 << VALID_VALUE << "instead saw " << TEST_VALUE << "\n";
351     }
352     catch (...)
353     {
354         clog << "testQueueFront FAILED\n";
355     }
356 }
357
358 void testQueueFrontConst()
359 {
360     Queue<int> tempQueue;
361     const int VALID_VALUE = 1;
362     tempQueue.push(VALID_VALUE);
363     try
364     {
```

```
365     const int *TEST_VALUE = &(tempQueue.front());
366
367     if (*TEST_VALUE == VALID_VALUE)
368         clog << "testQueueFrontConst PASSED\n";
369     else
370         clog << "testQueueFrontConst FAILED : Expected value "
371             << VALID_VALUE << " instead saw " << TEST_VALUE << "\n";
372 }
373 catch (...)
374 {
375     clog << "testQueueFrontConst FAILED\n";
376 }
377 }
378
379 void testQueueEmpty()
380 {
381     Queue<int> tempQueue;
382     const int VALID_VALUE = 1;
383     tempQueue.push(VALID_VALUE);
384     tempQueue.push(VALID_VALUE);
385     const int VALID_SIZE = 2;
386     try
387     {
388         Queue<int> tempQueue2;
389
390         if ((tempQueue2.empty()) && (!tempQueue.empty()))
391             clog << "testQueueEmpty PASSED\n";
392         else
393             clog << "testQueueEmpty FAILED\n";
394     }
395     catch (...)
396     {
397         clog << "testQueueEmpty FAILED\n";
398     }
399 }
400
401 void testQueueSize()
402 {
403     Queue<int> tempQueue;
404     const int VALID_VALUE = 1;
405     tempQueue.push(VALID_VALUE);
406     tempQueue.push(VALID_VALUE);
407     const int VALID_SIZE = 2;
408     try
409     {
410         const int TEST_VALUE = tempQueue.size();
411
412         if (TEST_VALUE == VALID_SIZE)
413             clog << "testQueueSize PASSED\n";
414         else
415             clog << "testQueueSize FAILED : Expected value "
416                 << VALID_VALUE << " instead saw " << VALID_SIZE << "\n";
```



```
417     }
418     catch (...)
419     {
420         clog << "testQueueSize FAILED\n";
421     }
422 }
423
424 int main(void)
425 {
426     testQueueConstructor();           //1a)
427     testQueueDestructor();            //1b)
428     testQueueCopyConstructor();       //1c)
429     testQueueCopyAssignment();        //1d)
430
431     testQueuePush();                  //2a)
432     testQueuePop();                   //2b)
433
434     testQueueFront();                 //3a)
435     testQueueFrontConst();            //3b)
436     testQueueEmpty();                 //3c)
437     testQueueSize();                  //3d)
438 }
439
```



```
Microsoft Visual Studio Debug Console

testQueueConstructor PASSED
testQueueDestructor PASSED
testQueueCopyConstructor PASSED
testQueueCopyAssignment PASSED
testQueuePush PASSED
testQueuePush size increase PASSED
testQueuePop empty PASSED
testQueuePop PASSED
testQueueFront PASSED
testQueueFrontConst PASSED
testQueueEmpty PASSED
testQueueSize PASSED

C:\Users\schem\source\repos\schemp98\Cpp_Certification_Course\Debug\Exercise.exe
(process 46460) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->
Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```