```cpp
// Shaun Chemplavil U08713628
// shaun.chemplavil@gmail.com
// C/C++ Programming IV : Advanced Programming with Objects
// 152488 Raymond L. Mitchell III
// hw2.cpp
// Win10
// Visual C++ 19.0
//

#include <iostream>
#include <cassert>
#include <exception>

using namespace std;

template <typename T>
class Queue
{
public:
   Queue(); // Construct empty queue
   ~Queue(); // Destructor
   Queue(const Queue &); // Copy constructor
   Queue &operator=(const Queue &); // Copy assignment operator
   void push(const T &); // Add elem to back of queue
   void pop(); // Remove front elem from queue
   T &front(); // Return ref to front elem in queue
   const T &front() const; // Return ref to front elem in queue
   bool empty() const; // Return whether queue is empty
   size_t size() const; // Return # of elems in queue
private:
   T *v_; // Elems in queue
   size_t vsize_;
   size_t vused_;
};
// Default Constructor
template <typename T>
Queue<T>::Queue() : v_(0), vsize_(10), vused_(0)
{
   v_ = new T[vsize_]; // Initial Allocation
}

// Destructor
template <typename T>
Queue<T>::~Queue()
{
   delete[] v_;
}

// Helper Function for copying array
template <typename T>
T*
newCopy(const T *src, size_t srcsize, size_t destsize)
```

```
{
   assert(destsize >= srcsize);
   T *dest = new T[destsize];
   try
   {
      copy(src, src + srcsize, dest);
   }
   catch (...)
   {
      delete[]dest;
      throw;
   }
   return dest;
}

// Copy Constructor
template <typename T>
Queue<T>::Queue(const Queue<T> &other)
   :v_(newCopy(other.v_, other.vsize_, other.vsize_)),
   vsize_(other.vsize_), vused_(other.vused_)
{
}
// Copy Assignment
template <typename T>
Queue<T>  &
Queue<T>::operator=(const Queue<T> &other)
{
   if (this != &other)
   {
      T *v_new = newCopy(other.v_, other.vsize_, other.vsize_);
      delete[] v_;
      v_ = v_new;
      vsize_ = other.vsize_;
      vused_ = other.vused_;
   }

   return *this;
}

// Push to the back of Queue
template <typename T>
void
Queue<T>::push(const T &t)
{
   // if necessary grow size
   if (vused_ == vsize_)
   {
      size_t vsize_new = vsize_ * 2 + 1;
      T *v_new = newCopy(v_, vsize_, vsize_new);
      delete[]v_;
      v_ = v_new;
      vsize_ = vsize_new;
```

```
   }
   // add element to back of queue
   v_[vused_] = t;

   // increase only after element copy
   ++vused_;
}

// Pop from the front of Queue
template <typename T>
void
Queue<T>::pop()
{
   if (vused_ == 0)
   {
      throw logic_error("pop from empty Queue");
   }
   else
   {
      // only copy (vused_-1) elements during a "pop"
      size_t vsize_new = vused_ - 1;
      // copy all but the first element into v_new
      T *v_new = newCopy(v_ + 1, vsize_new, vsize_);

      delete[] v_;
      // replace private member variable with "popped" array
      v_ = v_new;
      // decrease only after element removal
      --vused_;
   }
}

// Return reference to Front element
template <typename T>
T &
Queue<T>::front()
{
   if (vused_ == 0)
      throw logic_error("front from empty Queue");
   else
      return v_[0];
}

// Return reference to Front element
template <typename T>
const T &
Queue<T>::front() const
{
   if (vused_ == 0)
      throw logic_error("front from empty Queue");
   else
      return v_[0];
```

```
}

// Check if Queue is empty
template <typename T>
bool
Queue<T>::empty() const
{
   return (vused_ == 0);
}

// Return number of elements in Queue
template <typename T>
size_t
Queue<T>::size() const
{
   return (vused_);
}

// Unit Tests:
void testQueueConstructor()
{
   try
   {
      Queue<int> tempQueue;
      clog << "testQueueConstructor PASSED\n";
   }
   catch (...)
   {
      clog << "testQueueConstructor FAILED\n";
   }
}
void testQueueDestructor()
{
   Queue<int> *testQueue = new Queue<int>;
   try
   {
      delete testQueue;
      clog << "testQueueDestructor PASSED\n";
   }
   catch (...)
   {
      Queue<int> tempQueue;
      clog << "testQueueDestructor FAILED\n";
   }
}

void testQueueCopyConstructor()
{
   Queue<int> sourceQueue;
   const int VALID_INPUT = 42;
   const int NUM_ELEMENTS = 4;
   for (int idx = 0; idx < NUM_ELEMENTS; idx++)
```

```
      sourceQueue.push(VALID_INPUT * idx + 1);
   try
   {
      Queue<int> copyQueue(sourceQueue);

      for (int idx = 0; idx < NUM_ELEMENTS; idx++)
      {
         if (sourceQueue.front() == copyQueue.front())
         {
            sourceQueue.pop(); copyQueue.pop();
         }
         else
         {
            clog << "testQueueCopyConstructor FAILED : Expected  "
               << sourceQueue.front() << " instead saw "
               << copyQueue.front() << "\n";
         }
      }
      clog << "testQueueCopyConstructor PASSED\n";
   }
   catch (...)
   {
      clog << "testQueueCopyConstructor FAILED\n";
   }
}

void testQueueCopyAssignment()
{
   Queue<int> sourceQueue;
   const int VALID_INPUT = 42;
   const int NUM_ELEMENTS = 4;
   for (int idx = 0; idx < NUM_ELEMENTS; idx++)
      sourceQueue.push(VALID_INPUT * idx + 1);
   try
   {
      Queue<int> copyQueue = sourceQueue;

      for (int idx = 0; idx < NUM_ELEMENTS; idx++)
      {
         if (sourceQueue.front() == copyQueue.front())
         {
            sourceQueue.pop(); copyQueue.pop();
         }
         else
         {
            clog << "testQueueCopyAssignment FAILED : Expected "
               << sourceQueue.front() << " instead saw " <<
               copyQueue.front() << "\n";
         }
      }
      clog << "testQueueCopyAssignment PASSED\n";
   }
```

```
   catch (...)
   {
      clog << "testQueueCopyAssignment FAILED\n";
   }
}

void testQueuePush()
{
   Queue<int> tempQueue;
   try
   {
      const int INPUT = 1;
      const size_t ORG_SIZE = 10; // one more than
      const size_t NEW_SIZE = 11; // one more than

      for (int idx = 0; idx < ORG_SIZE; ++idx)
         tempQueue.push(idx * INPUT);

      clog << "testQueuePush PASSED\n";

      // Add one more than initial size allocation
      tempQueue.push(INPUT);

      // Check if size increase to expected value
      if (tempQueue.size() == NEW_SIZE)
         clog << "testQueuePush size increase PASSED\n";
      else
         clog << "testQueuePush size increase FAILED : Expected size "
         << NEW_SIZE << " instead saw " << tempQueue.size() << "\n";
   }
   catch (...)
   {
      clog << "testQueuePush FAILED\n";
   }
}

void testQueuePop()
{
   Queue<int> tempQueue;

   // test an empty pop
   try
   {
      tempQueue.pop();

      clog << "testQueuePop empty FAILED\n";
   }
   catch (exception &e)
   {
      if (strcmp(e.what(), "pop from empty Queue") == 0)
         clog << "testQueuePop empty  PASSED\n";
      else
```

```
         clog << "testQueuePop empty FAILED expected different error message\n";
   }

   //test successful pop
   try
   {
      const int FIRST_INPUT = 1;
      const int SECOND_INPUT = 2;
      tempQueue.push(FIRST_INPUT);
      tempQueue.push(SECOND_INPUT);

      if (tempQueue.front() == FIRST_INPUT)
         tempQueue.pop();

      // Check to see if FIRST_INPUT was successfully popped off
      if (tempQueue.front() == SECOND_INPUT)
         clog << "testQueuePop PASSED\n";
   }
   catch (...)
   {
      clog << "testQueuePop FAILED\n";
   }
}

void testQueueFront()
{
   Queue<int> tempQueue;
   const int VALID_VALUE = 1;
   tempQueue.push(VALID_VALUE);
   try
   {
      const int *TEST_VALUE = &(tempQueue.front());

      if (*TEST_VALUE == VALID_VALUE)
         clog << "testQueueFront PASSED\n";
      else
         clog << "testQueueFront FAILED : Expected value  "
         << VALID_VALUE << " instead saw " << TEST_VALUE << "\n";
   }
   catch (...)
   {
      clog << "testQueueFront FAILED\n";
   }
}

void testQueueFrontConst()
{
   Queue<int> tempQueue;
   const int VALID_VALUE = 1;
   tempQueue.push(VALID_VALUE);
   try
   {
```

```
      const int *TEST_VALUE = &(tempQueue.front());

      if (*TEST_VALUE == VALID_VALUE)
         clog << "testQueueFrontConst PASSED\n";
      else
         clog << "testQueueFrontConst FAILED : Expected value  "
         << VALID_VALUE << " instead saw " << TEST_VALUE << "\n";
   }
   catch (...)
   {
      clog << "testQueueFrontConst FAILED\n";
   }
}

void testQueueEmpty()
{
   Queue<int> tempQueue;
   const int VALID_VALUE = 1;
   tempQueue.push(VALID_VALUE);
   tempQueue.push(VALID_VALUE);
   const int VALID_SIZE = 2;
   try
   {
      Queue<int> tempQueue2;

      if ((tempQueue2.empty()) && (!tempQueue.empty()))
         clog << "testQueueEmpty PASSED\n";
      else
         clog << "testQueueEmpty FAILED\n";
   }
   catch (...)
   {
      clog << "testQueueEmpty FAILED\n";
   }
}

void testQueueSize()
{
   Queue<int> tempQueue;
   const int VALID_VALUE = 1;
   tempQueue.push(VALID_VALUE);
   tempQueue.push(VALID_VALUE);
   const int VALID_SIZE = 2;
   try
   {
      const int TEST_VALUE = tempQueue.size();

      if (TEST_VALUE == VALID_SIZE)
         clog << "testQueueSize PASSED\n";
      else
         clog << "testQueueSize FAILED : Expected value  "
         << VALID_VALUE << " instead saw " << VALID_SIZE << "\n";
```

```
   }
   catch (...)
   {
      clog << "testQueueSize FAILED\n";
   }
}

int main(void)
{
   testQueueConstructor();       //1a)
   testQueueDestructor();        //1b)
   testQueueCopyConstructor();   //1c)
   testQueueCopyAssignment();    //1d)

   testQueuePush();              //2a)
   testQueuePop();               //2b)

   testQueueFront();             //3a)
   testQueueFrontConst();        //3b)
   testQueueEmpty();             //3c)
   testQueueSize();              //3d)
}

```