```
#ifndef CSCI180_ARRAY_QUEUE_H
#define CSCI180_ARRAY_QUEUE_H

#include <stdexcept>                //   defines std::runtime_error

namespace csci180 {

  /** A queue implementation based upon use of a fixed-sized array.
   *  Elements are inserted and removed according to the first-in
   *  first-out principle.
   *
   *  This implementation is based loosely on the discussion given in
   *  Chapter 4.3 of our text, but it has been adjusted to suit my tastes.
   */
  template <typename Object>
  class ArrayQueue {

  private:
    int       capacity;                    // actual length of underlying array
    int       sz;                          // current size of the queue
    int       f;                           // index of the front of the queue
    Object*   Q;                           // pointer to the underlying array

  public:

    /** Standard constructor creates an empty queue with given capacity. */
    ArrayQueue(int cap = 1000) :
      capacity(cap), sz(0), f(0), Q(new Object[capacity]) { }

    /** Returns the number of objects in the queue.
     *  @return number of elements
     */
    int size() const {
      return sz;
    }

    /** Determines if the queue is currently empty.
     *  @return true if empty, false otherwise.
     */
    bool empty() const {
      return sz == 0;
    }

    /** Returns a const reference to the front object in the queue.
     * @return reference to front element
     */
    const Object& front() const {
      if (empty())
        throw std::runtime_error("Access to empty queue");
      return Q[f];
    }

    /** Returns a live reference to the front object in the queue.
     * @return reference to front element
     */
    Object& front() {
      if (empty())
        throw std::runtime_error("Access to empty queue");
      return Q[f];
    }
```

```
    /** Inserts an object at the back of the queue.
     *  @param the new element
     */
    void push(const Object& elem) {
      if (size() == capacity)
        throw std::runtime_error("Queue overflow");
      int back = (f + sz) % capacity;        // circular arithmetic
      Q[back] = elem;
      sz++;
    }

    /** Removes the front object from the queue. */
    void pop()  {
      if (empty())
        throw std::runtime_error("Access to empty queue");
      sz--;
      f = (f + 1) % capacity;               // advance circularly
    }

  private:
    // utility for copying data from other queue to this one,
    // intentionally aligning front with zero.
    void copyData(const ArrayQueue& other) {
      f = 0;
      int walk = other.f;                 // walk through other array
      for (int i=0; i < sz; i++) {
        Q[i] = other.Q[walk];
        walk = (walk + 1) % capacity;
      }
    }

  public:
    // Housekeeping functions

    /** Copy constructor */
    ArrayQueue(const ArrayQueue& other) :
      capacity(other.capacity), sz(other.sz), f(0), Q(new Object[capacity])
    {
      copyData(other);
    }

    /** Destructor */
    ~ArrayQueue() {
      delete[] Q;
    }

    /** Assignment operator */
    ArrayQueue& operator=(const ArrayQueue& other) {
      if (this != &other) {             // avoid self copy (x = x)
        capacity = other.capacity;
        sz = other.sz;
        delete [] Q;                    // delete old contents
        Q = new Object[capacity];
        copyData(other);
      }
      return *this;
    }

  };  // end of ArrayQueue class

} // end of csci180 namespace
#endif
```