Consider a queue implemented in an array. The queue in Figure 1 is used to represent a line of waiting customers. Alice arrived first. Fantine arrived last. The customer that will be served next is called the *oldest* because that customer has been in the line the longest. The customer that arrived most recently is called *newest*.

Figure 1. Queue implemented as an array

Array indexes are printed above customer names. For example, Alice is stored in element zero of the array. The index of the oldest customer trails that of the newest customer when two or more customers are in line. A customer is inserted onto the queue by incrementing the value of *newest* and storing the new customer in the element of the array indexed by *newest*. Index variable *newest* always contains the index of the customer that arrived most recently. A customer is deleted from the queue by removing the oldest customer from the queue and incrementing the value of *oldest*. Index variable *oldest* always contains the index of the customer that has been waiting in line the longest.

The problem with implementing a queue as an array is that arrays have finite length.

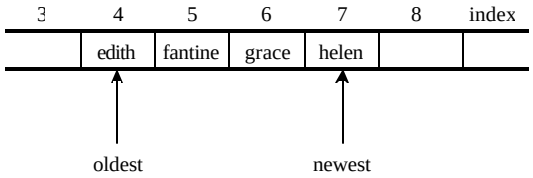

Figure 2. Queue in progress

As customers are served their elements in the array become available for arriving customers. By changing our perspective, so that the array is viewed as a circle, the elements of the array containing departed customers can now be reused.

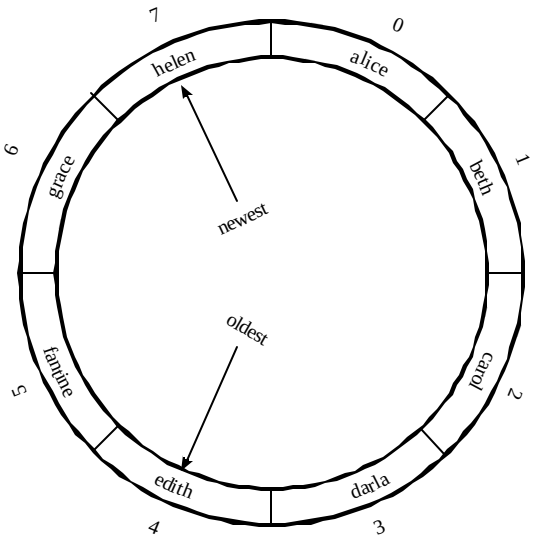

Figure 3. Circular array implementation of a queue

In the queue of Figure 3, alice, beth, carol, and darla have been served. We wish to reuse the space occupied by those customers who have departed.

It appears that element zero (0) follows element seven (7).

By using modulo arithmetic, we can make indexes *oldest* and *newest* follow each other around the circular queue of Figure 3.

newest=(*newest*+1)% *size*;
oldest=(*oldest*+1)% *size*;

Define *size* to be the number of elements in the array used to implement the queue.

The discussion thus far has included only a single instance of a queue. A fully functional abstract data type needs to the ability to create and destroy queues.

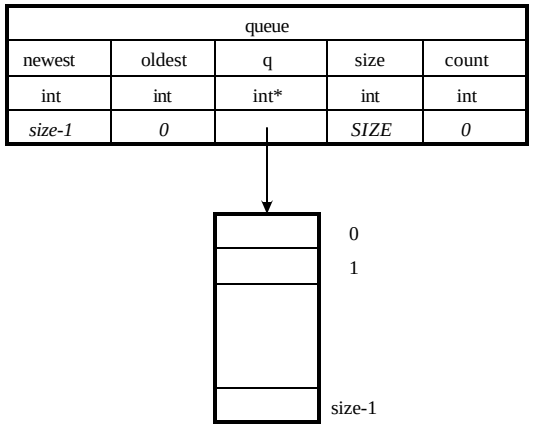

Figure 4. Queue implemented as a dynamically allocated array.

Functions:

1. **Constructor** - Function *Queue* assigns initial values to private members of class Queue. Function Queue also allocates storage for elements of the queue.

   Member *newest* contains the index of the most recently added value.

   Member *oldest* contains the index of the value that has been on the queue the longest.

   Member *q* is the array used to implement the queue. Technically, member *q* is a pointer to a value on the queue.

   Member *size* contains the number of elements in array *q*.

   Member *count* contains the number of elements on the queue.
2. **Destructor –** Function ~ frees the dynamically allocated array used to implement the queue.
3. **Member function** *Enq* - Function *Enq* inserts a value on the newest end of the queue.
4. **Member function** *Deq* - Function *Deq* removes a value from the oldest end of the queue and returns it to the caller.
5. **Member function** *IsEmpty* - Function *IsEmpty* determines if the queue is empty.
6. **Member function** *IsFull* - Function *IsIsFull* determines if the queue is IsFull.
7. **Member function** *Length* - Function *Length* returns the number of elements in the queue.

Queue::*Queue*(**int** *sz*)
1. Assign one less than the size of the array used to implement the queue to member *newest*. The first value on the queue is inserted in element zero.
2. Assign zero to member *oldest*. The first value removed from the queue is in element zero.
3. Allocate an array to implement the queue. Assign the pointer to the array to member *q*. There are *sz* elements in the array. Each element in the array is an integer.
4. Assign input parameter *sz* to member *size*.
5. Assign zero to member *count*. The queue is empty.

Queue::~*Queue*()
1. Free the array whose address is in member *q* if storage has been assigned to member *q*.

void *Queue*::*Enq*(**int** *v*)
1. Throw exception *QueueException* if the queue is *IsFull*. Note that *QueueException* does not return. Statements following this check depend on the queue having at least one free element.
2. Increment the value of member *newest* using modulo arithmetic.
3. Increment the value of member *count*.
4. Assign the value of input parameter *v* to that element of member *q* indexed by member *newest*.

int *Queue*::*Deq*(**void**)
1. Call function QueueException if the queue is empty. Note that *QueueException* does not return. Statements following this check depend on the queue having at least one element.
2. Increment the value of member *oldest* using modulo arithmetic.
3. Decrement the value of member *count*.
4. Return that element of member *q* indexed by member *oldest*.

int *Queue*::*IsEmpty*(**void**)
1. The queue is empty if the value of member *count* is zero.

int *Queue*::*IsFull*(**void**)
1. The queue is IsFull if the value of member *count* is greater than or equal to member *size*.

int *Queue*::*Length*(**void**)
1. Return the value of member *count*.

```
struct QueueException {
    QueueException(char* m)
    {   cout << endl;
        cout << "The queue is " << m << ".";
        cout << endl;
    }

};
```
1. Write the value of the string reference by input parameter *m* to stream *cout*.