4.1.3 An Array-Based Implementation

If applications in which a fixed-sized queue does not present a problem, you can use an array to represent a queue. A naive array-based implementation of a queue might include the following definition, as Figure 14-7a illustrates:

```c
#define MAX_QUEUE 100 // maximum size of queue

ItemType items[MAX_QUEUE]; // Array of queue items
int front; // Index to front of queue
int back; // Index to back of queue
```

![Figure 14-7](a) A naive array-based implementation of a queue; (b) rightward drift can cause the queue to appear full

Here `front` and `back` are the indices of the front and back entries, respectively, in the queue. Initially, `front` is 0 and `back` is 1. To add a new item to the queue, you increment `back` and place the item in `items[back]`. To remove an item, you simply increment `front`. The queue is empty whenever `front` is less than `back`. The queue is full when `back` equals `MAX_QUEUE - 1`.

The problem with this strategy is rightward drift—that is, after a sequence of additions and removals, the items in the queue will drift toward the end of the array, making it appear full. In other words, `back` could equal `MAX_QUEUE - 1` even when the queue contains only a few items. Figure 14-7b illustrates this situation.

One possible solution to this problem is to shift array entries to the left, either after each removal from the queue or whenever `back` equals `MAX_QUEUE - 1`. This solution guarantees that the queue can never contain up to `MAX_QUEUE` items. Shifting is not really satisfactory, however, as it would dominate the cost of the implementation.

**Question 4** Suppose that we change the naive array-based implementation of a queue pictured in Figure 14-7 so that the back of the queue is in `items[0]`. Although repeated removals from the front would no longer cause rightward drift, what other problem would this implementation cause?

A much more elegant solution is possible by viewing the array as circular, as Figure 14-8 illustrates. To remove an item, you increment the queue index `front`, and to insert an item, you increment...
Figure 14-9 illustrates the effect of a sequence of three queue operations on front, back, and the array. Notice that front and back "advance" clockwise around the array.

When either front or back advances past MAX_QUEUE - 1, it should wrap around to 0. This wraparound eliminates the problem of rightward drift, which occurred in the previous naïve implementation, because here the circular array has no end. You obtain the wraparound effect of a circular queue by using modulo arithmetic (that is, the C++ % operator) when incrementing front and back. For example, you can add newEntry to the queue by using the statements:

```
back = (back + 1) % MAX_QUEUE;
items[back] = newEntry;
```

Notice that if back equaled MAX_QUEUE - 1 before the addition of newEntry, the first statement, back = (back + 1) % MAX_QUEUE, would have the effect of wrapping back around to index 0. Similarly, you can remove the entry at the front of the queue by using the statement:

```
front = (front + 1) % MAX_QUEUE;
```

To initialize the queue, you set front to 0 and back to MAX_QUEUE - 1.

Figure 14-9 demonstrates the effect of three consecutive operations on the queue in Figure 14-8.
The only difficulty with this scheme is detecting when the queue is empty or full. It seems reasonable to select as the queue-empty condition

front is one slot ahead of back

This appears to indicate that front "passes" back when the queue becomes empty, as Figure 14-10(a) shows. However, it is also possible that this condition signals a full queue: Because the queue is circular, back might in fact "catch up" with front as the queue becomes full. Figure 14-10(b) illustrates this situation.

Obviously, you need a way to distinguish between the two situations. One way is to keep a count of the number of items in the queue. Before adding an item to the queue, you check whether the count equals MAX_QUEUE; if it is, the queue is full. Before removing an item from the queue, you check whether the count is equal to zero; if it is, the queue is empty.

The header file. Listing 14-4 contains the header file for an array-based implementation of the ADT that uses a circular array as just described. Because the data is stored in statically allocated memory, the compiler-generated destructor and copy constructor are sufficient.\(^1\)

\(^1\) If a dynamically allocated array, you must provide a destructor and copy constructor.
LISTING 14-4 The header file for the class ArrayQueue

```cpp
/** ADT queue: Circular array-based implementation. 
 * @file ArrayQueue.h */
#include "QueueInterface.h"
#define _ARRAY_QUEUE

const int MAX_QUEUE = 50;

template<class ItemType>
class ArrayQueue : public QueueInterface<ItemType>
{
private:
    ItemType items[MAX_QUEUE]; // Array of queue items
    int front; // Index to front of queue
    int back; // Index to back of queue
    int count; // Number of items currently in the queue

public:
    ArrayQueue(); // Copy constructor and destructor supplied by compiler
    bool isEmpty() const;
    bool enqueue(const ItemType& newEntry);
    bool dequeue();
    // @throw PrecondViolatedExcep if queue is empty. */
    ItemType peekFront() const throw(PrecondViolatedExcep);
}; // end ArrayQueue
#endif
#include "ArrayQueue.cpp"
#endif
```

The implementation file. Listing 14-5 contains the definitions of ArrayQueue's methods as they appear in the implementation file.

LISTING 14-5 The implementation file for the class ArrayQueue

```cpp
/** ADT queue: Circular array-based implementation. 
 * @file ArrayQueue.cpp */
#include "ArrayQueue.h" // Header file

template<class ItemType>
ArrayQueue<ItemType>::ArrayQueue() : front(0), back(MAX_QUEUE - 1), count(0)
{
} // end default constructor

template<class ItemType>
bool ArrayQueue<ItemType>::isEmpty() const
{
    return count == 0;
} // end isEmpty
```
```cpp
template<class ItemType>
bool ArrayQueue<ItemType>::enqueue(const ItemType &newEntry)
{
    bool result = false;
    if (count < MAX_QUEUE)
    {
        // Queue has room for another item
        back = (back + 1) % MAX_QUEUE;
        items[back] = newEntry;
        count++;
        result = true;
    } // end if
    return result;
} // end enqueue

template<class ItemType>
bool ArrayQueue<ItemType>::dequeue()
{
    bool result = false;
    if (!isEmpty())
    {
        front = (front + 1) % MAX_QUEUE;
        count--;
        result = true;
    } // end if
    return result;
} // end dequeue

template<class ItemType>
ItemType ArrayQueue<ItemType>::peekFront() const
    throw(PrecondViolatedExcep)
{
    // Enforce precondition
    if (isEmpty())
        throw PrecondViolatedExcep("peekFront() called with empty queue");
    // Queue is not empty; return front
    return items[front];
} // end peekFront
```

**Question 5** If the ADT queue had a method `clear` that removed all entries from a queue, what would its definition be in the previous array-based implementation?

**Variations.** Several commonly used variations of the previous circular-array approach do not require a count of the number of entries in the queue. One approach uses a boolean variable `isFull` to distinguish between the full and empty conditions. The expense of maintaining this variable is about the same as that of maintaining a counter, however. A faster implementation declares `MAX_QUEUE + 1` locations for the array `items`, but uses only `MAX_QUEUE` of them for queue items. You sacrifice one
Using an extra array location is more time efficient.

array location and let \( \text{front} \) be the index of the location before the front of the queue. As Figure 14-11 illustrates, the queue is full if

\[
\text{front} \text{ equals } (\text{back} + 1) \mod (\text{MAX\_QUEUE} + 1)
\]

but the queue is empty if

\[
\text{front} \text{ equals } \text{back}
\]

This approach does not have the overhead of maintaining a counter or boolean variable, and so is more efficient of time. Programming Problems 3 and 4 discuss these two alternate implementations further.

### 14.1.4 Comparing Implementations

We have suggested implementations of the ADT queue that store the queue's entries in either an instance of the ADT list, a chain of linked nodes that has both a head pointer and a tail pointer, a circular chain that has only one external pointer, an array, or a circular array. You have seen the details of three of these implementations. All of our implementations of the ADT queue are ultimately either array based or link based.

The reasons for making the choice between array-based and link-based implementations are the same as those discussed in earlier chapters. The discussion here is similar to the one in Section 4.5 of Chapter 4. We repeat the highlights here in the context of queues.

An implementation based on a statically allocated array prevents the enqueue operation from adding an item to the queue if the array is full. Such a queue is appropriate for many data structures, such as buffers, within an operating system. If this restriction is not acceptable, you must use either a dynamically allocated array or a link-based implementation.

Suppose that you decide to use a link-based implementation. Should you choose the implementation that uses a linked chain or the one that uses a link-based implementation of the ADT list? Because a linked chain actually represents the items in the ADT list, using the ADT list to represent a queue is not as efficient as using a linked chain directly. However, the ADT list approach is much simpler to write.

If you decide to use a linked chain instead of the ADT list to represent the queue, should you use a linear chain or a circular chain? We leave this question for you to answer in Programming Problem 1.