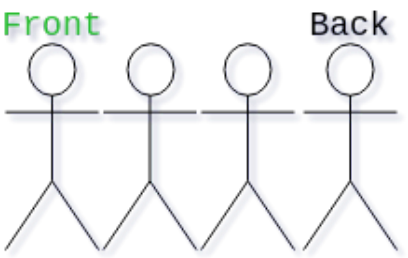
Queue



A queue is a data structure where data is inserted at one end and removed at the other end. The insertion end is known as the **back**, and the removal end is known at the **front**.

A queue can be thought of like a line at an amusement park. The first person to enter the line for a rollercoaster is the first person to ride the rollercoaster. The last person to enter the line for the rollercoaster is the last person to ride the rollercoaster. This process is known as **FIFO** (**F**irst **I**n **F**irst **O**ut). The people are always entering the line from the back and make their way to the front where they are able to ride the rollercoaster.

There are two terms in regards to insertion and deletion of data when it comes to a queue. **Push** is when you insert data into the queue and **pop** is when you delete data from the queue.

In order to create your own queue data structure, you need to have a few mandatory functions. These functions mimic the functions provided for you in the Standard Template Library queue class. A **push(DATA)** function is needed to insert data into the queue where **DATA** is a value of the type of the queue. A **pop()** function is used to delete data from the queue. A **size()** function returns how many pieces of data are in your queue. An **empty()** function returns a **bool** true if empty and false if data exists in the queue. A **front()** function that returns the data that is in the front of the queue. Lastly, a **back()** function that returns the data that is back in the queue.

It is wise to make your queue out of a template so you can have the flexibility to insert any simple base type into the queue. When you make a template queue, only one type is allowed per queue. Remember, the type is denoted in angle brackets, such as **Queue<char> myTemplateQueue;** One way to start with the creation of your queue is to use a **struct** for your data and implement the queue using a **linked list**. If you further decide to go the Object-Oriented route with a **class**, continue to use a **struct** for the data and implement a pointer to the **struct** as a **private** attribute of the class. For an example without a class implementation, see **Queue.cpp**. For an example with a class implementation, see **TemplateQueueClass.cpp**. Both programs are explained via comments.

When using the Standard Template Library version of a queue, you must include the preprocessor directive **#include <queue>**. If you are not **using namespace std;**, you need to either put **using std::queue;** at the top of your program or prepend queue with **std::** as in **std::queue<int> myQueue;**

Since a queue's data is inserted into the back and deleted from the front, there are no iterators to be used.

A queue in the Standard Template Library is implemented using either a **deque** or **list**. The default data structure for a queue is a **deque**. This means you are able to put a deque into a queue. An example of this is as follows:

```
std::deque<int> myDeque(5, 50); //deque with 5 elems with value of 50
std::queue<int> myQueue(myDeque); //queue 5 elements with value of 50
```

Now, if you want to put a **list** into a queue, you must explicitly override the default deque implementation as such (keep in mind you must have declared and initialized the list beforehand:

```
std::queue<double, std::list<double>> myQueue(myList);
```

To initialize an empty queue, you do the following:

```
std::queue<int> myQueue;
```

The following are member functions for the **queue** class. There aren't many due to the simplistic properties of a queue.

To check whether the queue is empty, use the **empty()** member function. This will return true if empty and false if there are some contents in the queue. An example of this is as follows: `if(myQueue.empty())`

To see the size of the queue, use the member function **size()**. This will return a **size_t** (like an **int** but cannot be a negative number) that is the number of elements currently in the queue. An example of this is as follows: `cout << myQueue.size();`

To access the next element in the queue, use the member function **front()**. This returns a reference to the front element of the queue (the first element that was pushed into the queue). An example of this is: `cout << myQueue.front();` Since this returns a reference, you are also able to change the value of the front element such as: `myQueue.front() = 4;`

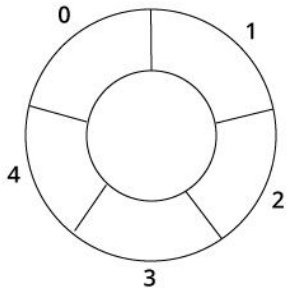
To access the last element in the queue, use the member function **back()**. This returns a reference to the back element of the queue (the last element that was pushed into the queue). An example of this is: `cout << myQueue.back();` Since this returns a reference, you are also able to change the value of the last element such as: `myQueue.back() = 5*3;`

To push data into the queue, use the member function **push(DATA)**. This will place data into the back of the queue. An example of this is: `myQueue.push('a');` This will insert the **char a** into a queue of type **char**.

To remove the front element of the queue, use the member function **pop()**. This will delete the element at the front of the queue making the second inserted element the new front element if it exists. If there are no further elements, the queue is empty. An example is simply: **myQueue.pop();**

To find out more information on queues, please visit **http://www.cplusplus.com/reference/queue/queue/**

Circular Queue



A circular queue is a queue that is constructed out of an array. Initially, the **front** of the queue is at **index** 0 and the back of the queue is set to be at position **SIZE-1** where SIZE is the total size of the array. Once the queue is full, no additional insertions may be made.

A circular queue should be made using a template. That way various types are able to utilize the circular queue. It is beneficial to implement this as a class with the array, **front**, **back**, and size as **private** member variables.

The interesting thing about a circular queue is that the **front** and **back** will change position. That is, with each insertion **back** will change to the next index in sequence and with each deletion **front** will change to the next index in sequence. With that being said, **front** and **back** are indexes of the array.

A **helper function** that calculates the next index in the circular queue is also wise to have as a **private** member function. The formula for the next index in sequence is as follows: **(i+1) % SIZE** where **i** is the index of **front** or **back** and **SIZE** is the size of the array. For example, if the array has a **SIZE** of 5 and if **back** has an index of 4, **(4+1) % 5 = 0**. Therefore, 0 will be the index for insertion so long as the circular queue is not full.

To tell if a circular queue is empty, you could check to see if **size** is equal to zero. This may be a beneficial option, because **front** and **back** start one index apart in an empty circular queue and if the circular queue is full, **front** and **back** will also be one index apart. To tell if a circular queue is empty, you could check to see if **size** is equal to the total size of the array. You could also see if **front** and **back** are one index apart and if the circular queue is not empty.

The standard **push(DATA)**, **pop()**, **front()**, **back()**, **size()**, **empty()** functions should be implemented. See **CircularQueue.cpp** for a class implementation and explanation via comments in the code.

Priority Queue

A priority queue is a data structure in which the data is ordered from greatest to least. The highest number has the highest priority and the lowest number has the lowest priority. That is, the highest number is the number that is on top and will be popped first, then the second highest number will be popped second, and so forth. A priority queue is similar to a heap and a heap can be used for a priority queue. For more information on a heap, look at the **Maxheap** page and the program example **MaxHeap.cpp**.

When using the Standard Template Library version of a queue, you must include the preprocessor directive **#include <queue>**. If you are not **using namespace std;** you need to put **using std::priority_queue;** at the top of your program or prepend priority_queue with **std::** as in **std::priority_queue<int> myPq;**

A priority queue in the Standard Template Library is implemented using either a **deque** or **vector**. The default data structure for a priority queue is a **vector**.

To initialize an empty priority queue, you do the following:

```
std::priority_queue<int> myPq;
```

The following are member functions for the **priority_queue** class. There aren't many due to the simplistic properties of a priority queue.

To check whether the priority queue is empty, use the **empty()** member function. This will return true if empty and false if there are some contents in the queue. An example of this is as follows: **myPq.empty()**.

To see the size of the priority queue, use the member function **size()**. This will return a **size_t** (like an **int** but cannot be a negative number) that is the number of elements currently in the queue. An example of this is as follows: **cout << myPq.size();**

To access the top element in the priority queue, use the member function **top()**. An example of this is: **cout << myPq.top();** This is not a manipulative function like the the top function in a stack.

To push data into the priority queue, use the member function **push(DATA)**. This will place data in the appropriate position. An example of this is: **myPq.push(5);** This will insert the **int 50** between the data less than 50 and greater than 50.

visit **http://www.cplusplus.com/reference/queue/priority_queue/**