Queue Implementations

1. Circular Queues
   - buffer of fixed capacity
   - improvements and cost estimates

2. Deques
   - the double ended queue
   - queue as double linked circular list
Queue Implementations

1. Circular Queues
   - buffer of fixed capacity
   - improvements and cost estimates

2. Deques
   - the double ended queue
   - queue as double linked circular list
A queue can be linear or circular.

Applications for circular queues:
- waiting room with fixed # seats,
- buffer to process data.

We first use an array to implement a queue
1. two indices, to front and back element,
2. update: (index + 1) modulo array size.
   → modulo capacity, indices only increase
a circular buffer of fixed size

Imagine a waiting room with a circular seating arrangement.

Three important numbers define the state of the queue:
- The index \textit{current} points to the front of the queue.
- The \textit{back} points to the end of the queue.
- The index calculation happens modulo the capacity of the queue.

The queue may be empty or not.
the UML class diagram

<table>
<thead>
<tr>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>− data</td>
</tr>
<tr>
<td>− capacity</td>
</tr>
<tr>
<td>− current</td>
</tr>
<tr>
<td>− back</td>
</tr>
<tr>
<td>− number</td>
</tr>
</tbody>
</table>

| + Queue() |
| + push() |
| + empty() |
| + front() |
| + pop() |
private data members

```cpp
#ifndef __MCS360_CIRCULAR_FIXED_BUFFER_H__
#define __MCS360_CIRCULAR_FIXED_BUFFER_H__

namespace mcs360_circular_fixed_buffer
{
    template <typename T>
    class Queue
    {
        private:

            T *data; // an array of items
            size_t capacity; // capacity of the buffer
            int current; // index to the front of queue
            int back; // index to the end of queue
            size_t number; // the number of elements
    ```
the public methods

public:

Queue ( int c );
// creates an empty queue of capacity c
void push ( T item );
// pushes the item at the end
bool empty();
// returns true if queue is empty
T front();
// returns the front element
void pop();
// removes the front element

};

#include "mcs360_circular_fixed_buffer.tpp"
#endif
constructor and push

namespace mcs360_circular_fixed_buffer
{
    template <typename T>
    Queue<T>::Queue ( int c )
    {
        capacity = c; data = new T[c];
        current = -1; back = -1; number = 0;
    }
    template <typename T>
    void Queue<T>::push ( T item )
    {
        this->back = (this->back + 1) % this->capacity;
        this->data[this->back] = item;
        this->number = this->number + 1;
        if(this->current < 0) this->current = 0;
    }
}
rest of methods

template <typename T>
bool Queue<T>::empty()
{
    return (this->number == 0);
}
template <typename T>
T Queue<T>::front()
{
    return data[this->current];
}
template <typename T>
void Queue<T>::pop()
{
    this->current = (this->current + 1) % this->capacity;
    this->number = this->number - 1;
}
testing the buffer

#include <iostream>
#include "mcs360_circular_fixed_buffer.h"
using namespace mcs360_circular_fixed_buffer;
using namespace std;

int main()
{
    Queue<int> q(10);

    for(int i=1; i<6; i++) q.push(i);
    for(; !q.empty(); q.pop())
        cout << q.front() << endl;
    for(int i=6; i<12; i++) q.push(i);
    for(; !q.empty(); q.pop())
        cout << q.front() << endl;
    for(int i=12; i<18; i++) q.push(i);
    for(; !q.empty(); q.pop())
        cout << q.front() << endl;
Queue Implementations

**1 Circular Queues**
- buffer of fixed capacity
- improvements and cost estimates

**2 Deques**
- the double ended queue
- queue as double linked circular list
The implementation is simple and efficient:

- management of indices straightforward,
- efficient if queue size \( \approx \) capacity.

All operations have cost \( O(1) \).

Suggestions for improvement:

- throw exceptions for when pop empty or push to full queue;
- enlarge capacity when full.
Queue Implementations

1. Circular Queues
   - buffer of fixed capacity
   - improvements and cost estimates

2. Deques
   - the double ended queue
   - queue as double linked circular list
the Deque

A deque is a double ended queue:
- we can pop from the front or from the end; and
- we can push to the front or to the end.

Methods of STL deque, on `deque<T> q`:
- `q.push_back(t)`: append to queue at end
- `q.push_front(t)`: insert to queue at front
- `t = q.back()`: return last element of queue
- `t = q.front()`: return first element of queue
- `q.pop_back()`: remove last element of queue
- `q.pop_front()`: remove first element of queue
Queue Implementations

1 Circular Queues
   - buffer of fixed capacity
   - improvements and cost estimates

2 Deques
   - the double ended queue
   - queue as double linked circular list
implementing a deque

We adapt our `mcs360_double_list::List`, using the `Node` definition of `mcs360_double_node.h`.

The goal is to implement a deque:
- circular: we can circulate
- doubly linked: move forward & backward.

If we allow to push and pop to the front and end of the queue, then the queue is double ended, or a deque.
a double ended queue or a deque

Consider a dequeue which stores 'a', 'b', 'c':

Observe: last = current->prev.
Queue

- current Node
- number

+ Queue()
+ empty()
+ size()
+ front()
+ move_front_forward()
+ move_front_back()
+ push_front()
+ push_back()
+ pop_front()
+ pop_back()
the data members

```cpp
#ifndef __MCS360_CIRCULAR_DOUBLE_RING_H__
#define __MCS360_CIRCULAR_DOUBLE_RING_H__

namespace mcs360_circular_double_ring
{
    template <typename T>
    class Queue
    {
    private:

        #include "mcs360_double_node.h"
        Node *current; // pointer to current node
        int number; // number of elements
    }
}
```

the Node definition

From lecture 12: mcs360_double_node.h:

```c
#ifndef __DNODE_H__
#define __DNODE_H__

struct Node
{
    T data; // T is template parameter
    Node *next; // pointer to next node
    Node *prev; // pointer to previous node

    Node(const T& item,
         Node* next_ptr = NULL,
         Node* prev_ptr = NULL) :
        data(item),next(next_ptr),prev(prev_ptr) {}
};

#endif
```

Introduction to Data Structures (MCS 360)
public methods

public:

Queue(); // returns an empty queue

bool empty();
// true if queue empty, false otherwise
int size();
// returns the size of the queue
T front();
// returns the item at front of queue

void move_front_forward();
// makes next element front of queue
void move_front_backward();
// makes previous item front of queue
void push_front(T item);
// inserts item in front of queue
void push_back(T item);
// appends item to the end

void pop_front();
// removes the element at front
void pop_back();
// removes the element at end

#include "mcs360_circular_double_ring.tpp"
#endif

Notice the grouping of the methods.
namespace mcs360_circular_double_ring
{
    template <typename T>
    Queue<T>::Queue()
    {
        current = NULL;
        number = 0;
    }

    template <typename T>
    bool Queue<T>::empty()()
    {
        return (current == NULL);
    }

    template <typename T>
    int Queue<T>::size()
    {
        return number;
    }
}
template <typename T>
T Queue<T>::front()
{
    return current->data;
}

template <typename T>
void Queue<T>::move_front_forward()
{
    current = current->next;
}

template <typename T>
void Queue<T>::move_front_backward()
{
    current = current->prev;
}
Appending ‘d’ to a dequeue that stores ‘a’, ‘b’, ‘c’:

```
last
  Node
    next:*  
    prev:*  
    data: 'a'

current
  Node
    next:*  
    prev:*  
    data: 'b'

Node
  next:*  
  prev:*  
  data: 'c'
```

Introduction to Data Structures (MCS 360)  Queue Implementations  L-17  21 February 2020  25 / 29
Appending 'd' to a dequeue that stores 'a', 'b', 'c':

```
<table>
<thead>
<tr>
<th>last</th>
<th>current</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="" alt="Node" /></td>
<td><img src="" alt="Node" /></td>
</tr>
</tbody>
</table>
```

append to the dequeue
Appending ‘d’ to a dequeue that stores ‘a’, ‘b’, ‘c’:

last

Node

next: *
prev: *
data: ‘a’

current

Node

next: *
prev: *
data: ‘b’

Node

next: *
prev: *
data: ‘c’
exercise: fix the links

Node
next: *
prev: *
data: 'a'

Node
next: *
prev: *
data: 'b'

Node
next: *
prev: *
data: 'c'

Node
next: *
prev: *
data: 'd'

current

last
template <typename T>
void Queue<T>::push_back(T item)
{
    if (current == NULL)
    {
        current = new Node(item);
        current->next = current;
        current->prev = current;
    }
    else
    {
        Node *last = current->prev;
        current->prev
            = new Node(item, current, current->prev);
        last->next = current->prev;
    }
    number = number + 1;
}
template <typename T>
void Queue<T>::pop_front()
{
    if(current != NULL) {
        if(current->prev == current->next)
        {
            delete current;
            current = NULL; number = 0;
        }
        else {
            Node *last = current->prev;
            last->next = current->next;
            current->next->prev = last;
            delete current;
            current = last->next;
            number = number - 1;
        }
    }
}
Summary + Exercises

More on Chapter 6 on queue implementations.

Exercises:

1. Use the STL list in a templated class to implement a queue. Define an exception class `Queue_Empty` and illustrate how to operate your queue implementation with a test program.

2. Instead of an array use the STL `vector` to implement the circular fixed buffer queue. In redefining all methods, ensure that the `push` automatically doubles the capacity when full.

3. Give code for `push_front` on our circular doubly linked list implementation for a deque. Make a drawing to illustrate the logic of the code.

4. Give code for `pop_back` on our circular doubly linked list implementation for a deque. Make a drawing to illustrate the logic of the code.