Queues — Overview

- A **queue** is a data structure for storing…
  - ...an ***ordered*** collection of items…
    - (there is ***some positional order*** that is significant with respect to how items can be ***accessed***)
  - ...such that items can only be added through one end…
    - (called the **rear**)
  - ...and removed through another end
    - (called the **front**)
- The items at the *front* and *rear* are sometimes…
  - …called the **first item** and the **last item**, respectively

Queues — Overview

- A queue is a ***first-in-first-out*** (***FIFO***) data structure
  - The first item added to the queue will be the first item removed from the queue

Queues

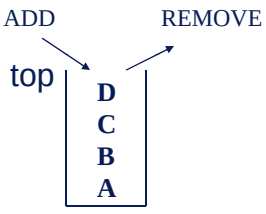
Overview

- Queues and stacks are very similar data structures...
  - ...differing only in the rule that determines which item is removed from the list first
    - Items removed from stacks are removed in an order that is the reverse of that in which they were added
    - Items removed from queues are removed in an order that is the same as that in which they were added

Queues

Stacks vs Queues

- With a stack, items are…
  - …added and removed through the same end
- With a queue, items are…
  - … added through one end and removed through another end



Queues — Queue Operations

- ***Two fundamental operations*** for a queue:
  - Adds a new item to the queue
    - Through the ***rear***
    - Traditionally called ***enqueue*** but implemented as **`push`** in the C++ Standard Library ➔ adopted and noted by textbook authors
  - Removes an existing item from the queue
    - Through the ***front***
    - Traditionally called ***dequeue*** but implemented as **`front`** and **`pop`** in the C++ Standard Library ➔ adopted and noted by textbook authors
- Other useful supporting operations for a queue:
  - *E.g.*: **`empty`** ➔ checks if a queue is empty
  - *E.g.*: **`size`** ➔ returns the number of items in the queue

Queues — Queue Errors

- ***Two queue error conditions*** are possible:
  - ***Queue Underflow***
    - The condition resulting from an attempt to ***remove*** an item from an ***empty*** queue
    - Can be avoided by testing if the queue is empty (using **`empty`**, say) before attempting to remove
  - ***Queue Overflow***
    - The condition resulting from an attempt to ***add*** an item to a ***full*** queue ➔ relevant if the queue has a fixed-size capacity
    - Can be avoided by testing if the queue is empty (comparing the # of items in the queue – obtained using **`size`**, say – and the queue's capacity) before attempting to add

Queues
Example Real-life Applications

- Queues are a frequently occurring real-life structure...
  - Waiting in line at an ATM machine
  - Waiting for clear or green light at a traffic stop
  - Waiting on hold when making a telephone call
  - ...

  ...and are thus frequently used in simulation programs
- In a computer system in which resources are shared, a queue is typically used to allocate a shared resource so that requests for the shared resource are handled on a "first-come, first-served" basis
  - *E.g.*: The "Ready Room" print queue

Queues
Putting a Queue to Use
Echoing a Word

- A trivial example of a problem that can be solved using a queue is ***echoing*** a word
  - We've seen how ***reversing*** a word was solved using a stack
- To do this we would read each character in the input one at a time and add it (using **`push`**) to an initially empty queue to store it
- After the entire word is read and stored we would remove (using **`front`** and **`pop`**) each stored character from the queue and output the results until the queue is empty

Algorithm

Declare a queue of characters
While (there are still characters to read)
  read a character
  add the character to the queue (push)
While (the queue is not empty)
  remove a character from the queue (front & pop)
  output the result

- A ***palindrome*** is a sequence of characters that reads the same forward and backward
  - The characters are the same when read from left to right as they are when read from right to left
- Some simple one-word examples are…

  `radar rotor madam racecar level`
- A more complicated multi-word example is...

  `Able was I ere I saw Elba`

- Suppose we need a program to help use recognize if an input sequence of characters is a palindrome
- We could solve the problem by using a stack and a queue
  - The ***stack*** will hold the characters in ***reverse*** order
  - The ***queue*** will hold the characters in ***regular*** order

- The first step would be to read the input sequence, pushing each character onto the stack and inserting the same character into the queue
- When the input is finished, a pair of characters is repeatedly removed from the stack and queue (one from each) and compared
  - A mismatch is tallied if the characters in a pair are not equal
  - The process continues until the stack and queue are empty
- The input sequence is a palindrome if no mismatch is found when the process is complete, otherwise it is not a palindrome

- Page 383 of the textbook presents a C++ program that implements the above algorithm
- To increase the chances of finding palindromes...
  - It ignores case-sensitivity by converting each character read to upper case before adding the character to the stack or queue
    - It uses the `toupper` function from the `cctype` library (`ctype.h` for the older style)
  - It ignores spaces, punctuation and digits by simply skipping any character read that is not an alphabet, adding the character to the stack or queue only if the character is an alphabet
    - It uses the `isalpha` function from the `cctype` library (`ctype.h` for the older style)

- Queues may be conceptually simpler than stacks...
  - …because manifestations of queues are common in our everyday life
- Implementing a queue is more complicated than implementing a stack, however...
  - …because a stack has only ***one*** end for handling items
  - …whereas a queue has ***two*** ends for handling items

- Since we are likely to need queues of different kinds of things…
  - ...it makes sense to implement the queue with a template class
- Two implementations of a queue template class…
  - ...are shown in Section 8.3 of the textbook
  - ...one using a static array, the other using a linked list
- The queue template class implements some general operations required of a queue

- The queue template class implemented has the following member functions:
  - constructor ➔ creates an empty queue
  - `void push(const Item& entry)`
  - `void pop()`
  - `Item front() const`
  - `size_type size() const`
  - `bool empty() const`

- The queue implementation using an array shows a technique that is interesting/useful for us to learn
  - We are likely to run into the same idea again later
  - We will therefore study in more detail the queue implementation using an array
- The next group of slides collate and summarize the key aspects involved

- Summary notes on implementing a queue using a static array:
  - The queue can hold up to a fixed number of items
    - **`CAPACITY`** – size of the array that dictates the fixed number
  - Variables are needed to keep track of how much of the array is used
    - **`first`** – index of the first item in the queue (***front index***)
    - **`last`** – index of the last item in the queue (***rear index***)
    - **`count`** – number of items currently in the queue

Queue Implementation
Straightforward but Inefficient Array Implementation

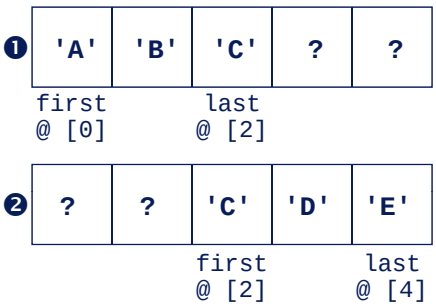
- Summary notes on implementing a queue using a static array (*continued*):
  - The most straightforward approach is to maintain the queue so that the ***front of the queue is always the first element*** (*i.e.*, the ***front index is fixed and is always 0***)
    - Every time an item is removed, all the remaining items are shifted forward one spot
    - Items are always added after the rear index
  - The preceding approach is not very efficient
    - Shifting all items that are still in the queue after each removal of an item from the queue is expensive

Queue Implementation
More Efficient (Circular) Array Implementation

- Summary notes on implementing a queue using a static array (*continued*):
  - It is therefore desirable not to have to shift all items remaining in the queue after each removal of an item
  - The desired effect can be realized if the ***front index is allowed to vary*** (instead of being fixed at 0)
  - But this introduces a new problem...
    - The rear index is incremented when an item is added
    - The front index is incremented when an item is removed
    - Both indexes are never decremented
    - What's to become when an index reaches the array end-bound?

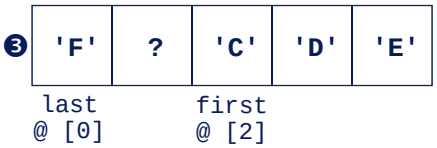
- Summary notes on implementing a queue using a static array (*continued*):
  - The key idea is to simply ***restart an index at 0 when the index reaches the array end-bound***
    - (Notice that unless the queue is full or empty, there will be applicable location(s) or item(s) at the beginning of the array when an index reaches the array end-bound)
    - Conceptually, this is treating the first element of the array as if it is immediately after the last element
    - This is the same as looking at an array bent into a circle ➔ thus the term ***circular array***

- Summary notes on implementing a queue using a static array (*continued*):



Remove first two items and add two new items

- Summary notes on implementing a queue using a static array (*continued*):



If we now want to add another item, we cannot place it after **`last`** because **`last`** has reached the array end-bound

We should instead go to the beginning of the array and add it at location 0

- Summary notes on implementing a queue using a static array (*continued*):
  - It is convenient to encapsulate the task of incrementing an index (in a way that incorporates the circular array concept) in a **`next_index`** "helper" function
    - The function is meant for "internal" use ➔ to facilitate the implementation of the queue ADT
    - It is not meant to be made available to programmers who wish to make use of the queue ADT
    - It is therefore appropriate to include the function as a **`private`** (not **`public`**) member function

Queue Implementation
More Efficient (Circular) Array Implementation

- Summary notes on implementing a queue using a static array (*continued*):
  - The **`next_index`** function is to help us step through the array one element after another with a wraparound at the end
    - **`next_index(i)`** usually returns **`i+1`**...
    - …except if **`i`** is the ***index of the last array element***…
    - …for which **`next_index(i)`** should return **`0`** (the index of the first array element) ➔ the wraparound
  - Noting that the expression

```
i = (i + 1) % CAPACITY
```

    properly updates **`i`** to capture the preceding desired effect, the body of the **`next_index`** function can be written simply as

```
return (i + 1) % CAPACITY;
```


Queue Implementation
Template Circular Array Implementation

```
#include <cstdlib>  // provides size_t

template <class Item>
class queue
{
public:
  typedef size_t size_type;
  static const size_type CAPACITY = 30;
  queue();
  void push(const Item& entry);
  void pop();
  Item front() const;
  size_type size() const;
  bool empty() const;
```

See Section 8.3 of the textbook for similar implementation in new style. Beware! There are some errors.

continued

Queue Implementation
Template Circular Array Implementation

```
private:
  Item data[CAPACITY];     // circular array
  size_type first;         // front index
  size_type last;          // rear index
  size_type count;         // number of items
  size_type next_index(size_type i) const;
};

#include "queue1.template"
```


Queue Implementation
Template Circular Array Implementation

```
#include <cassert>
#include <cstdlib>

template <class Item>
queue<Item>::queue()
: count(0), first(0), last(CAPACITY - 1) { }

template <class Item>
void queue<Item>::push(const Item& entry)
{
  std::assert( size() < CAPACITY );
  last = next_index(last);
  data[last] = entry;
  ++count;
}
```

?

continued

Queue Implementation
Template Circular Array Implementation

```
template <class Item>
void queue<Item>::pop()
{
  std::assert( !empty() );
  first = next_index(first);
  --count;
}

template <class Item>
Item queue<Item>::front() const
{
  std::assert( !empty() );
  return data[first];
}
```

continued

Queue Implementation
Template Circular Array Implementation

```
template <class Item>
typename queue<Item>::size type queue<Item>::size() const
{ return count; }

template <class Item>
bool queue<Item>::empty() const
{ return (count == 0); }

template <class Item>
typename queue<Item>::size_type
queue<Item>::next_index(size_type i) const
{ return (i+1) % CAPACITY; }
```

Priority Queues

- A ***priority queue*** is a data structure that stores entries along with a ***priority*** for each entry
  - Entries are removed in order of priorities
    - Highest priority items are removed first
    - If there are multiple items of equal priority level, they are removed in a FIFO manner ➔ how it relates to queue
  - Essentially, it allows items to "cut in line"

Priority Queues

One Implementation Approach (Code Outline)

```
#include "queue1.h"  // provides ordinary queue
#include <cstdlib>   // provides size_t

template <class Item>
class priority_queue
{
public:
  type size_t size_type;
  static const size_type HIGHEST = 2;
  ...
private:
  // creates an array of ordinary queues
  queue<Item> queues[HIGHEST + 1]; // 0, 1, 2
  ...
};
```

Priority Queues

Other Implementation Considerations

- If the number of possible priorities is large, the array-of-queues approach may not be practical
- Another approach is to use an ordinary linked list implementation of a queue but add a priority variable to each node
  - Each item would be inserted at the appropriate position on the list

Textbook Readings

- Chapter 8
  - All sections