Queues

- A *queue* is an object container for which objects are inserted and removed according to the *first-in-first-out* (FIFO) principle
  - objects are added to the *rear* of the queue and are removed from the *front* of the queue

Queues

- An instance of the queue class supports two fundamental methods:

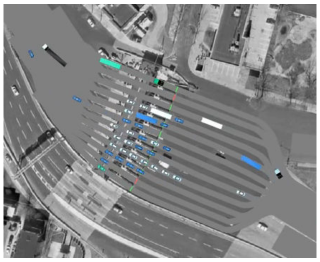
```
enqueue(obj):  Add obj to the rear of the queue
                 • Input: Object; Output: none
obj dequeue():Remove and return the object at
               the head of the queue, error if
               the queue is empty
                 • Input: None; Output: Object
```

The queue ADT provides the following supporting methods:

```
size():        Returns the number of objects in
               the queue
                 • Input: None; Output: Integer
bool isEmpty(): Returns a boolean indicating
               whether the queue is empty
                 • Input: None; Output: Boolean
obj& front():  Returns but does not remove the
               object at the front of the
               queue, error if empty queue
                 • Input: None; Output: Reference to
                 Object
```

Queues

- An example of traffic control software using simulation and queues



Queues

- A .h for the Queue class would be:

```
class Queue
{
   public:
   typedef LinkedList::value_type value_type;

   // Mutator member functions
   Queue();
   ~Queue();
   void enqueue(const value_type& entry);
   value_type dequeue();
   bool isEmpty() const;

   // Query member functions
   int size() const;
   value_type& front();

   private:
   LinkedList data;
   int used;
};
```

Implementation of Functions

```
void Queue::enqueue(const value_type& entry) {
   ++used;
   data.addToTail(entry);
}

value_type Queue::dequeue() {
    --used;
   return data.removeFromHead();
}

value_type& Queue::front(){
   data.moveToHead();
   return data.getCurrent();
}
```

Linked List-based Queue

- Given a class `LinkedList` that provides the following (presuming `template <typename Item>`):

```
void addToHead(const Item&);
void addToTail(const Item&);
void addToCurrent(const Item&);
Item removeFromHead();
Item removeFromTail();
void removeCurrent();
bool removeOne(const Item&);
bool removeAll(const Item&);
void moveToHead();
void moveToTail();
void forward();
void back();
Item& getCurrent() const;
int size() const;
```

- we can implement a `Queue` based on a linked list
- Ok... I guess... wait! What is that "template" thing?

Templates

- The *template* is the C++ mechanism for producing generic functions and classes
  - E.g. it could be used to avoid the

    `typedef <class> value_type;`

    statement in definition of our `Node` class
- A function template is not a function
  - Rather it is a blueprint for what can become a function at compile time
- For example we could define a generic function `findMax()` that took two instances of any class and returned the instance that is 'bigger' by some class-defined criterion

Template Function Example

- Templates are defined in a `.h` file in the same way as are class profiles

```
// This is the file templates.h
// Generic function findMax(const Item& arg1,
//                          const Item& arg2)
// Returns arg1 if it is bigger than arg2,
// arg2 otherwise.
// Item objects must implement the > operator
// A copy constructor is also required

template <typename Item>
const Item& findMax(const Item& a,
                        const Item& b){
   if (a > b)
      return a;
   else
      return b;
}
```

- This function can be used by any class the "includes" `templates.h`

Comments

- Note the requirement that it must be possible for the `>` operator to be applied to the arguments
  - Thus, in the case of non-primitive types, the `>` operator must be overloaded before the `findMax()` function can be used for that type
- The effect of the template is that the compiler creates overloaded implementations of `findMax()` for each type to which it is applied in the code

What the Compiler Does

- Given that the statements:

```
#include "templates.h"
…
Account acc1;
Account acc2;
int i1 = 3;
int i2 = 5;
```

exist in the code, later followed by:

```
Account acc3 = findMax(acc1, acc2);
int i3 = findMax(i1, i2);
```

the compiler would produce the following code:

```
// Compiler-generated implementation to deal with account
   instances
const Account& findMax(const Account& a,const Account& b) {
   if (a > b) return a;
   else return b;
}

// Compiler-generated implementation to deal with int
const int& findMax(const int& a,const int& b) {
   if (a > b) return a;
   else return b;
}
```

Comments

- The fact that `Account` must have a defined `>` operator requires the following:

```
// In account.h
bool operator >(const account& a1,
                const account& a2);

// In account.cpp
bool operator >(const account& a1,
                const account& a2) {
    return (a1.balance() > a2.balance());
};
```

- The operator `>` is already defined for `int` in the C++ language so it can be relied upon for the overloaded `findMax()` function as applied to `int`
- Note that we can now apply `findMax()` to any type for which `>` is defined

Generic Classes

- This is achieved in a similar way to that used for functions, e.g. a generic Node class would be defined as follows

```
template <typename Item>
class Node {
   public:
      Node();
…
   private:
      Item data;
      Node* next;
}
```

- In simple terms, wherever you had `value_type` from:

```
typedef Node::value_type value_type;
```

- Now you will have `Item`

Example

- Change `LinkedList` and `Node` from last week to template classes.

Stacks

- A *stack* is an object container for which objects are inserted and removed according to the *last-in-first-out* (LIFO) principle
  - only the most recently inserted (or "pushed") object can be removed (or "popped") at any time

Stack Operations

- A stack *S* is a data type supporting the following functionality:

`push(obj)`: Insert *obj* at the top of the stack
- Input: Object; Output: None;

`obj pop()`: Remove and return the top object *obj* from the stack and if the stack is empty return a NULL object
- Input: None; Output: Object;

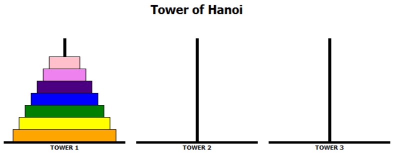
In support of these are the following:

`int size()`: Return the number of objects on the stack
- Input: None; Output: Integer;

`bool empty()`: Return a boolean indicating whether there are objects on the stack
- Input: None; Output: boolean;

`obj& top()`: Return the top object on the stack without removing it.
- Input: None; Output: Reference to Object;

An example



Stack Underflow & Overflow

- Stacks are so important that C++ provides them as a template class of the STL
  - `push(obj), pop(), top(), size() and empty()`
  - Assignment and the copy constructor can be used with stack objects
- An attempt to `pop()` from an empty stack creates a *stack underflow* condition
- An attempt to `push()` onto a full stack causes a *stack overflow* condition

Linked List-based Stack

- Given a class `LinkedList` that provides the following (presuming `template <typename Item>`):

```
void addToHead(const Item&);
void addToTail(const Item&);
void addToCurrent(const Item&);
Item& removeFromHead();
Item& removeFromTail();
void removeCurrent();
bool removeOne(const Item&);
bool removeAll(const Item&);
void moveToHead();
void moveToTail();
void forward();
void back();
Item& getCurrent();
int size() const;
```

- we can implement a `Stack` based on a linked list

List-based Stack

```
// This is Lstack.h
#ifndef LSTACK_H
#define LSTACK_H

#include "LinkedList.h"
#include <cstdlib>
// namespace declaration
template <typename Item>
class Lstack {
public:
    // Constructor
    Lstack();
    // Destructor
    ~Lstack();
    // Mutators
    void push(const Item&);
    Item pop();
    // Query
    Item& top();
    bool empty();
    int size() const;
private:
    LinkedList<Item> data;
};

#include "Lstack.template"
#endif
```

Notes on Class Templates

- In the spirit of separating the profile from the implementation (and noting that neither is 'real', rather they provide a blueprint for the compiler)
  - Provide the profile in `TemplateName.h`
  - Provide the 'implementation' in `TemplateName.template`
- You *must* have `#include “TemplateName.template”` at the end of `TemplateName.h`
  - The profile and its implementation must be in the same file (as far as the compiler is concerned – the template implementation is not a real implementation. Implementation is done during the compilation.)
- Never have any `using` directives in the implementation file
  - So use `std::` in front of any STL functionality, e.g. `std::copy`

More on Class Templates

- Every member function implementation must start with the template header
  - E.g. `template <typename Item>`
- The name of the template class must be provided in a form that provides both the class name and the notional application class
  - E.g. `Item& LinkedList<Item>::getCurrent()`
- The constructor's name does *not* change
  - E.g. `LinkedList<Item>::LinkedList()`
- When the template class is instantiated, the application class is defined. E.g.

```
Lstack<char> charStack;
Lstack<int> intStack;
```

- Your `Makefile` includes the `.h` file for the template class(es), e.g.:

```
SOURCES=test.cpp LinkedList.h Node.h Lstack.h
```

Link List-based Stack Example

```
// This is Lstack.template

template <typename Item>
Lstack<Item>::Lstack(){}
// if data was a pointer to a linked list,
// then the constructor should implement
// "data = new LinkedList<Item>();"


template <typename Item>
Lstack<Item>::~Lstack(){}
// if data was a pointer to a linked list,
// then the destructor should implement
// "delete data;"

template <typename Item>
void Lstack<Item>::push(Item& new_item)
{
    data.addToHead(new_item);
}
// if pointer, use ->

template <typename Item>
Item Lstack<Item>::pop()
{
    return data.removeFromHead();
}
// etc
```

Deques - 1

- A *deque* is a double-ended queue
  - it supports addition and removal from both the beginning and the end
  - An instance of deque provides the following methods:

```
insertFirst(obj):   Insert e at the beginning
                    • Input: Object, Output:
                    None
insertLast(obj):    Insert e at the end
                    • Input: Object, Output:
                    None
obj removeFirst():  Remove and return first
                    element
                    • Input: None, Output:
                    Object
obj removeLast():   Remove and return last
                    element
                    • Input: None, Output:
                    Object

The following methods may also be supported:
first(), last(), size() and isEmpty()
```

Deques - 2

- Deques can be used to implement *stacks* and *queues*
- The correspondences between stack and deque methods respectively are:
  - `size()` and `size()`
  - `isEmpty()` and `isEmpty()`
  - `top()` and `last()`
  - `push(obj)` and `insertLast(obj)`
  - `pop()` and `removeLast()`

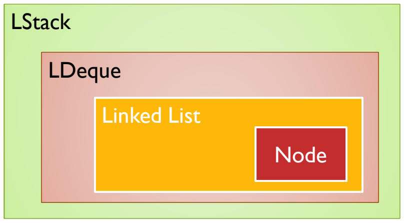
Deques - 3

- the correspondences between queue and deque methods respectively are:
  - `size()` and `size()`
  - `isEmpty()` and `isEmpty()`
  - `front()` and `first()`
  - `enqueue(`*`obj`*`)` and `insertLast(`*`obj`*`)`
  - `dequeue()` and `removeFirst()`

Deques - 4

- Assuming an implementation LDeque of deque, a stack may be implemented as follows:
  - Using `insertLast(`*`obj`*`)` for `push(`*`obj`*`)`
  - Using `removeLast()` for `pop()`
  - Using `last()` for `top()`
  - Etc.

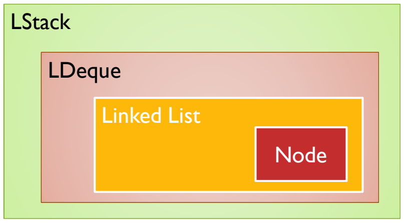
Wrapping around classes



```
class LinkedList
// version with direct access to private member variables

void addToHead(const node::value_type& entry);
void addToTail(const node::value_type& entry);
void add(node* prev_ptr, const node::value_type& entry);
void head_remove();
void tail_remove();
void remove(node* prev_ptr);
...
private:
      node* head_ptr;
      node* tail_ptr;
      node* current;
...
```

Wrapping around classes



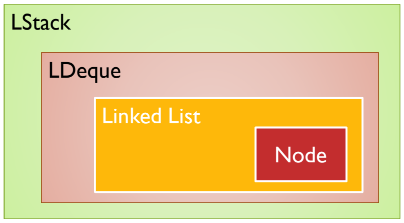
```
class LDeque
// all imports are included

void insertLast(const LinkedList::value_type& entry);
void insertFirst(const LinkedList::value_type& entry);
LinkedList::value_type removeLast();
LinkedList::value_type removeFirst();
...
private:
      LinkedList dequeData;
...
```

Implementation:

```
void insertLast(const LinkedList::value_type& entry){
   dequeData.addToTail(entry);
}
void insertFirst(const LinkedList::value_type& entry){
   dequeData.addToHead(entry);
}
```

Wrapping around classes



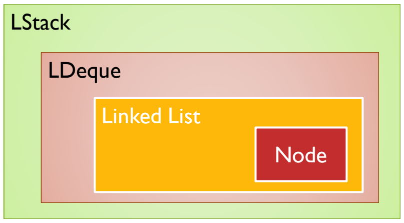
```
class LStack
// all imports are included

void push(const LDeque::value_type& entry);
LDeque::value_type& top();
LDeque::value_type pop();
...
private:
      LDeque stackData;
...
```

Implementation:

```
void push(const LDeque::value_type& entry){
   stackData.insertFirst(entry);
}
LDeque::value_type void pop(){
    return stackData.removeFirst();
}
```

Wrapping around classes



- Of course, this is an extreme example. Normally, you would go straight from `LinkedList` to `LStack`, without passing through `Ldeque`, otherwise there is too much overhead.
- But it serves as an illustration. Use this for the assignments. It will simplify the work a lot.