CMSC 341
Lecture 6 – Templates, Stacks & Queues

Based on slides by Shawn Lupoli & Katherine Gibson at UMBC

Today's Topics

- Data types in C++
- Overloading functions
- Templates
  - How to implement them
  - Possible problems (and solutions)
  - Compiling with templates
- Stacks
- Queues

Data Types

Data Types (Review)

- Values of variables are stored somewhere in an unspecified location in the computer memory as zeros and ones
- Our program does not need to know the exact location where a variable is stored
  - It can simply refer to it by its name
- What the program needs to be aware of is the **kind** of data stored in the variable

Fundamental Data Types in C++

Group	Type names*	Notes on size / precision
Character types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than `char`. At least 16 bits.
	char32_t	Not smaller than `char16_t`. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as `char`. At least 8 bits.
	signed **short** *int*	Not smaller than `char`. At least 16 bits.
	signed **int**	Not smaller than `short`. At least 16 bits.
	signed **long** *int*	Not smaller than `int`. At least 32 bits.
	signed **long long** *int*	Not smaller than `long`. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	**float**	
	double	Precision not less than `float`
	long double	Precision not less than `double`
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

Source: http://www.cplusplus.com/doc/tutorial/variables/

Overloading Functions

What is Overloading?

- Used to create multiple definitions for functions in various settings:
  - Class constructors
  - Class operators
  - Functions

- Let’s look at a simple swap function

Example: Swap Function

- Here is a function to swap two integers:

```
void SwapVals(int &v1, int &v2) {
  int temp;

   temp = v1;
   v1 = v2;
   v2 = temp;
}
```

what if we want to swap two floats?

what do we need to change?

Example: Swap Function

- Here is a function to swap two floats

```
void SwapVals(float &v1, float &v2) {
  float temp;

    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

what if we want to swap two chars?

what do we need to change?

Example: Swap Function

- Here is a function to swap two chars

```
void SwapVals(char &v1, char &v2) {
  char temp;

   temp = v1;
   v1 = v2;
   v2 = temp;
}
```

what if we want to swap two strings?

what do we need to change?

Example: Swap Function

- This is getting ridiculous!
- We should be able to write just <u>one</u> function that can handle all of these things
  - The only difference is the data type, after all
- This is possible by using templates

Templates

Common Uses for Templates

- Some common algorithms that easily lend themselves to templates:
  - Swap
  - Sort
  - Search
  - FindMax
  - FindMin

`maxx()` Overloaded Example

```
float    maxx ( const float a, const float b );
int      maxx ( const int a, const int b );
Rational maxx ( const Rational& a, const Rational& b);
myType   maxx ( const myType& a, const myType& b);
```

- Code for each looks the same…

```
if ( a < b )
  return b;
else
  return a;
```

we want to reuse this code for **all** types

What are Templates?

- Templates let us create functions and classes that can use “generic” input and types

- This means that functions like **`SwapVals()`** only need to be written once
  - And can then be used for almost anything

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

this keyword tells the compiler that what follows this will be a template

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

this **does not** mean "class" in the same sense as C++ classes with members!

in fact, another keyword we can use is actually "`typename`", because we are defining a new type

but "`class`" is more common by far, and so we will use class to avoid confusion

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

"**`T`**" is the name of our new type

we can call it anything we want, but using "**`T`**" is the style convention

(of course, we can't use "**`int`**" or "**`for`**" or any other types or keywords as a name for our type)

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

- What this line means overall is that we plan to use "**`T`**" in place of a data type
  - *e.g.*, **`int`**, **`char`**, **`myClass`**, etc.
- This template prefix needs to be used before function declarations and function definitions

Template Example

Function Template

```
template <class T>
T maxx ( const T& a, const T& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

Compiler generates code based on the argument type

```
cout << maxx(4, 7) << endl;
```

Generates the following:

```
int maxx ( const int& a, const int& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

Notice how '**`T`**' is mapped to '**`int`**' everywhere in the function…

Using Templates

- When we call these templated functions, nothing looks different:

```
SwapVals(intOne,     intTwo);
SwapVals(charOne,    charTwo);
SwapVals(strOne,     strTwo);
SwapVals(myClassA,  myClassB);
```

(In)valid Use of Templates

- Which of the following will work?

```
SwapVals(int, int);
SwapVals(char, string);
SwapVals("hello", "world");
SwapVals(double, float);
SwapVals(Shape, Shape);
```

These use two different types, and the SwapVals() function doesn't allow this.

These are two string literals – we can't swap those!

Template Requirements

- Templated functions can handle any input types that "makes sense"
  - *i.e.*, any data type where the behavior that occurs in the function is defined
- Even user-defined types!
  - **As long as the behavior is defined**
  - What happens if the behavior isn't defined?
    - Compiler will give you an error

Overloading Templates

Why Overload Templates?

- Sometimes, even though the behavior is defined, the function performs incorrectly

- Assume the code:

```
char* s1 = "hello";
char* s2 = "goodbye";
cout << maxx( s1, s2 );
```

- What is the call to **`maxx()`** actually going to do?

Incorrect Template Performance

- The compiler generates:

```
char* maxx (const char*& a, const char*& b)
{
if ( a < b )
  return b;
else
  return a;
}
```

- Is this what we want?

Overloading a Template

- Fix this by creating a version of **`maxx()`** specifically to handle **`char*`** variables
  - Compiler will use this instead of the template

```
char* maxx(char *a, char *b)
{
  if (strcmp(a,b) < 0)
    return b;
  else
    return a;
}
```

Compiling Templates

Compiler Handling of Templates

- Exactly what versions of **`SwapVals()`** are created is determined at compile time
- If we call **`SwapVals()`** with integers and strings, the compiler will create versions of the function that take in integers and strings

Separate Compilation

- Which versions of templated function to create are determined at compile time
- How does this affect our use of separate compilation?
  - Function declaration in **`.h`** file
  - Function definition in **`.cpp`** file
  - Function call in separate **`.cpp`** file

Separate Compilation: Example Code

- Here's an illustrative example:

```
#include "swap.h"

int main()
{
   int a = 3, b = 8;
   SwapVals(a, b);
}
```

main.cpp

```
template <class T>
void SwapVals(T &v1, T &v2);
```

swap.h

```
#include "swap.h"

template <class T>
void SwapVals(T &v1, T &v2)
{
   T temp;
   temp = v1;
   v1   = v2;
   v2   = temp;
}
```

swap.cpp

Separate Compilation

- Most compilers (including GL's) cannot handle separate compilation with templates

- When **`swap.cpp`** is compiled…
  - There are no calls to **`SwapVals()`**
  - **`swap.o`** has no **`SwapVals()`** definitions

Separate Compilation

- When **`main.cpp`** is compiled…
  - It assumes everything is fine
  - Since **`swap.h`** has the appropriate declaration

- When **`main.o`** and **`swap.o`** are linked…
  - Everything goes wrong
  - **`error: undefined reference to ‘void SwapVals<int>(int&, int&)’`**

Separate Compilation Solutions

- The template function definition code must be in the same file as the function call code

- Two ways to do this:
  - place function definition in **`main.c`**
  - place function definition in **`swap.h`**, which is #included in **`main.c`**

Template Compilation Solution

- Second option keeps some sense of separate compilation, and better allows code reuse

```
#include "swap.h"

int main()
{
  int a = 3, b = 8;
  SwapVals(a, b);
}
```

main.cpp

```
// declaration
template <class T>
void SwapVals(T &v1, T &v2);

// definition
template <class T>
void SwapVals(T &v1, T &v2)
{
  T temp;
  temp = v1;
  v1   = v2;
  v2   = temp;
}
```

swap.h

CMSC 341
Stacks and Queues

Prof. Jeremy Dixon

Topics for Today

- Introduction to Standard Template Library (STL)
- Stacks
  - Types of Stacks
  - Examples
- Queues
  - Types of Queues
  - Examples

Standard Template Library (STL)

Standard Template Library (STL)

- The Standard Template Library (*STL*) is a C++ library of container classes, algorithms, and iterators
- Provides many of the basic algorithms and data structures of computer science

From: https://www.sgi.com/tech/stl/stl_introduction.html

Considerations of the STL

- Containers replicate structures very commonly used in programming.
- Many containers have several member functions in common, and share functionalities.

From: http://www.cplusplus.com/reference/stl/

Considerations of the STL

- The decision of which type of container to use for a specific need depends on:
  - the functionality offered by the container
  - the efficiency of some of its members (complexity)

From: http://www.cplusplus.com/reference/stl/

Types of Containers

Focus of Today

- Sequence containers
  - Array, vector, deque, forward_list, list
- Container adapters
  - Stacks, queues, priority_queues
- Associative containers (and the unordered)
  - Set, multiset, map, multimap

Standard Containers

- Sequences:
  - **vector**: Dynamic array of variables, struct or objects. Insert data at the end.
  - **list**: Linked list of variables, struct or objects. Insert/remove anywhere.
  - Sequence **means order does matter**

Container Adapters

- Container adapters:
  - **stack** LIFO
  - **queue** FIFO
  - adapter means ***VERY LIMITED*** functionality

Will we use STL?

- Today we are going to talk about the ways that we can implement stacks and queues

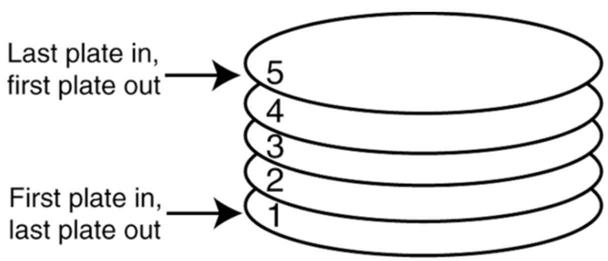
- 3 Ways to Create a Stack or Queue
  - Create a static stack or queue using an array
  - Create a dynamic stack or queue using a linked list
  - Create a stack or queue using the STL

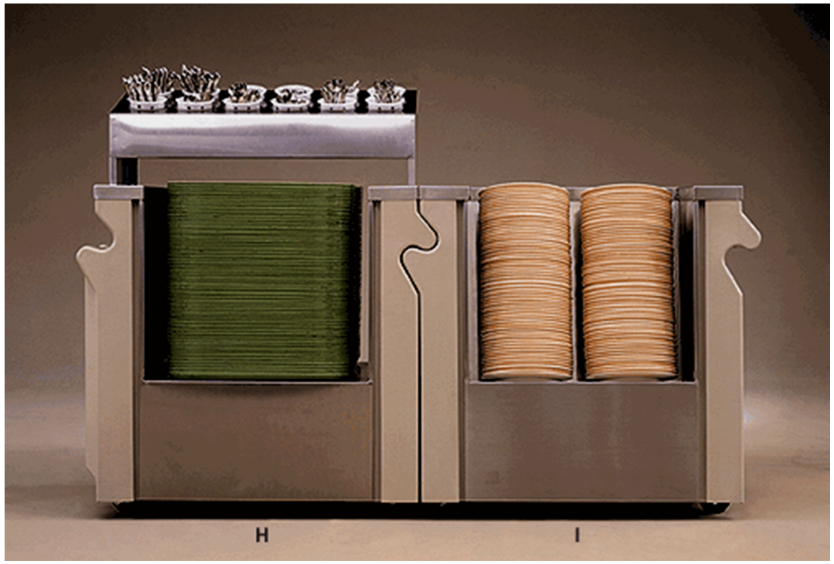
Stacks

Stacks

Introduction to Stacks

- A *stack* is a data structure that stores and retrieves items in a last-in-first-out (LIFO) manner.





Applications of Stacks

- Computer systems use stacks during a program's execution to store function return addresses, local variables, etc.
- Some calculators use stacks for performing mathematical operations.

Implementations of Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list
- Using STL (dynamic)

Stack Operations

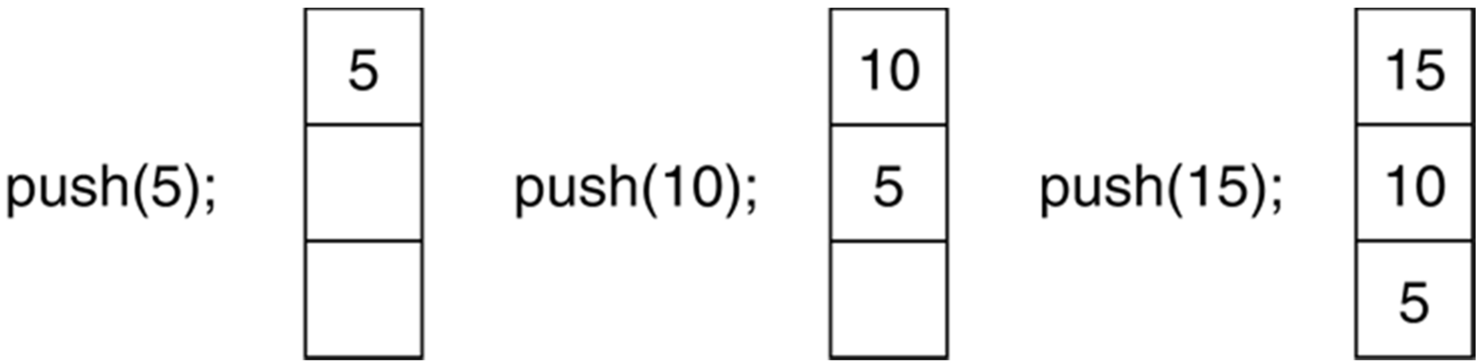
- Push
  - causes a value to be stored in (pushed onto) the stack
- Pop
  - retrieves and removes a value from the stack

The Push Operation

- Suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations.

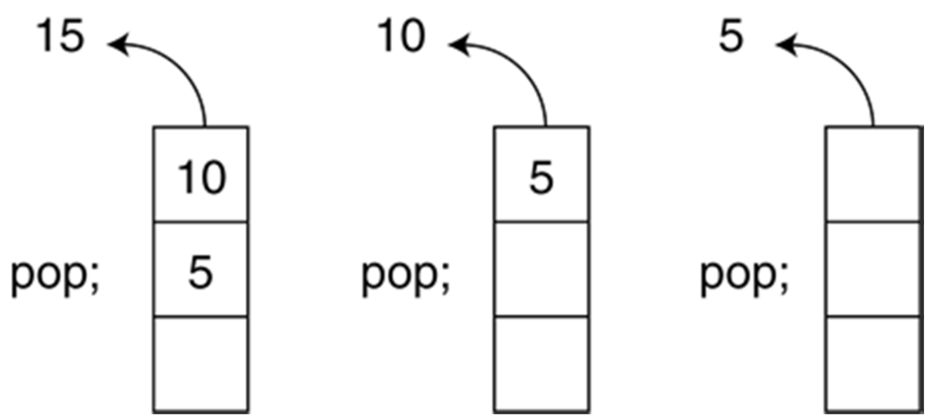
```
push(5);
push(10);
push(15);
```

The Push Operation



The Pop Operation

- Now, suppose we execute three consecutive pop operations on the same stack:



Other Stack Operations

- **`isFull()`**: A Boolean operation needed for static stacks. Returns true if the stack is full. Otherwise, returns false.
- **`isEmpty()`**: A Boolean operation needed for all stacks. Returns true if the stack is empty. Otherwise, returns false.

Static Stacks

Static Stacks

- A *static stack* is built on an array
  - As we are using an array, we must specify the starting size of the stack
  - The stack may become full if the array becomes full

Member Variables for Stacks

- Three major variables:
  - **`Pointer`** Creates a pointer to stack
  - **`size`** Tracks elements in stack
  - **`top`** Tracks top element in stack

Member Functions for Stacks

- **`CONSTRUCTOR`** Creates a stack
- **`DESTRUCTOR`** Deletes a stack
- **`push()`** Pushes element to stack
- **`pop()`** Pops element from stack
- **`isEmpty()`** Is the stack empty?
- **`isFull()`** Is the stack full?

Static Stack Definition

```
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
private:
        int *stackArray;
        int stackSize;
        int top;

public:
        IntStack(int);
        ~IntStack()
           {delete[] stackArray;}
        void push(int);
        void pop(int &);
        bool isFull();
        bool isEmpty();
};

#endif
```

Member Variables:

- **pointer**
- **size()**
- **top()**

Member Functions:

- **Constructor**
- **Destructor**
- **push()**
- **pop()**
- **isFull()**
- **isEmpty()**

Dynamic Stacks

Dynamic Stacks

- A *dynamic stack* is built on a linked list instead of an array.
- A linked list-based stack offers two advantages over an array-based stack.
  - No need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, and then expands by one node each time a value is pushed.
  - A dynamic stack will never be full, as long as the system has enough free memory.

Member Variables for Dynamic Stacks

- Parts:
  - **`Linked list`** Linked list for stack (nodes)
  - **`size`** Tracks elements in stack

Member Functions for Dynamic Stacks

- **`CONSTRUCTOR`** Creates a stack
- **`DESTRUCTOR`** Deletes a stack
- **`push()`** Pushes element to stack
- **`pop()`** Pops element from stack
- **`isEmpty()`** Is the stack empty?
- **`top()`** What is the top element?

What happened to `isFull()`?

Dynamic Stack

```
class DynIntStack
{
private:
     struct StackNode
     {
          int value;
          StackNode *next;
     };

     StackNode *top;

public:
     DynIntStack(void)
          {    top = NULL; }
     void push(int);
     void pop(int &);
     const Elem& top() const throw(StackEmpty);
     bool isEmpty(void);
};
```

Member Variables:

- **Linked list of elements** → `struct StackNode`
- **value** → `int value;`
- **pointer** → `StackNode *next;`
- **top** → `StackNode *top;`

Member Functions:

- **Constructor** → `DynIntStack(void)`
- **push()** → `void push(int);`
- **pop()** → `void pop(int &);`
- **top()** → `const Elem& top() const throw(StackEmpty);`
- **isEmpty()** → `bool isEmpty(void);`

Common Problems with Stacks

- Stack underflow
  - no elements in the stack, and you tried to pop
- Stack overflow
  - maximum elements in stack, and tried to add another
  - not an issue using STL or a dynamic implementation
- Practice question: *Stack Min* - How would you design a stack which, in addition to push and pop, has a function min which returns the minimum element? Push, pop and min should all operate in 0(1) time.

Queues

Introduction to the Queue

- Like a stack, a queue is a data structure that holds a sequence of elements.
- A queue, however, provides access to its elements in *first-in, first-out (FIFO)* order.

- The elements in a queue are processed like customers standing in a line: the first customer to get in line is the first one to be served (and leave the line).

Example Applications of Queues

- In a multi-user system, a queue is used to hold print jobs submitted by users, while the printer services those jobs one at a time.
- Communications software also uses queues to hold information received over networks. Sometimes information is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.

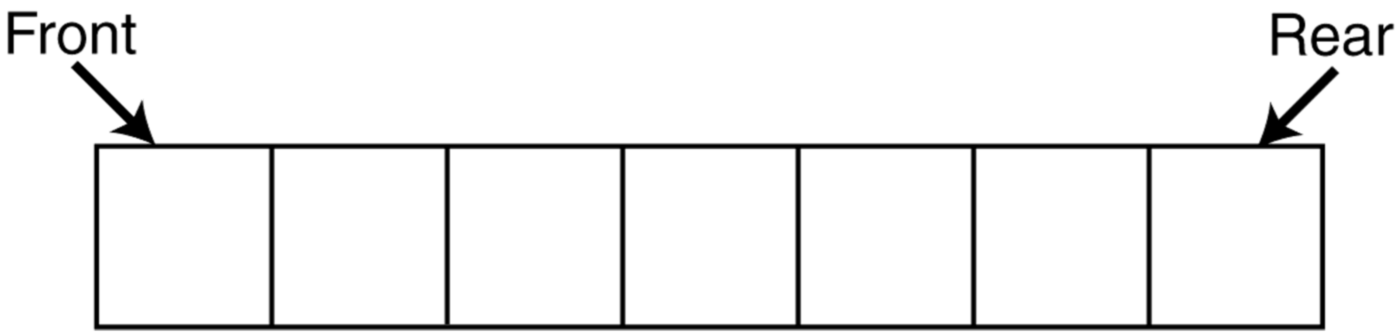
Implementations of Queues

Just like stacks!

- Static Queues
  - Fixed size
  - Can be implemented with an array
- Dynamic Queues
  - Grow in size as needed
  - Can be implemented with a linked list
- Using STL (dynamic)

Queue Operations

- Think of queues as having a front and a rear.
  - rear: position where elements are added
  - front: position from which elements are removed



Queue Operations

- The two primary queue operations are *enqueuing* and *dequeuing*.
- To *enqueue* means to insert an element at the rear of a queue.
- To *dequeue* means to remove an element from the front of a queue.

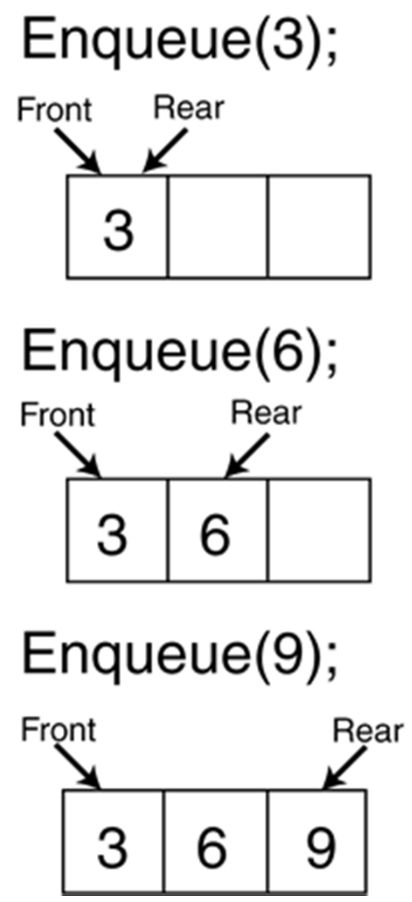
Queue Operations

- Suppose we have an empty static integer queue that is capable of holding a maximum of three values. With that queue we execute the following enqueue operations.

```
Enqueue(3);
Enqueue(6);
Enqueue(9);
```

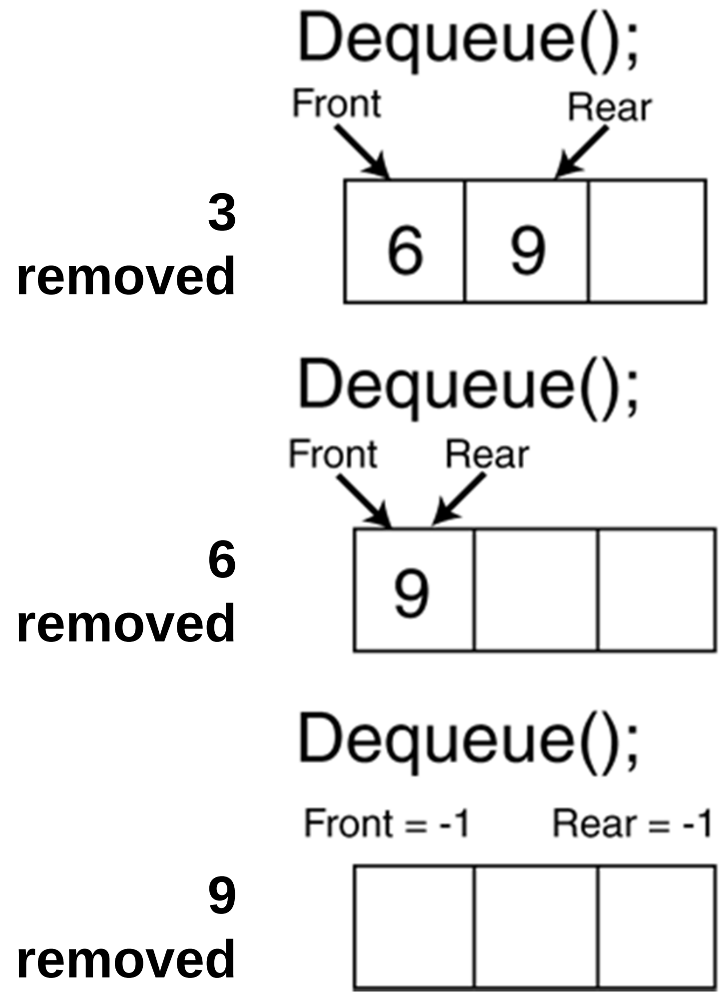
Queue Operations - Enqueue

- The state of the queue after each of the enqueue operations.



Queue Operations - Dequeue

- Now let's see how dequeue operations are performed. The figure on the right shows the state of the queue after each of three consecutive dequeue operations
- An important remark
  - After each dequeue, remaining items shift toward the front of the queue.



Efficiency Problem of Dequeue & Solution

- Shifting after each dequeue operation causes inefficiency.
- Solution
  - Let front index move as elements are removed
  - let rear index "wrap around" to the beginning of array, treating array as circular
    - Similarly, the front index as well
  - Yields more complex enqueue, dequeue code, but more efficient
  - Let's see the trace of this method on the board for the enqueue and dequeue operations given on the right (queue size is 3)

```
Enqueue(3);
Enqueue(6);
Enqueue(9);
Dequeue();
Dequeue();
Enqueue(12);
Dequeue();
```

Implementation of a Static Queue

- The previous discussion was about static arrays
  - Container is an array
- Class Implementation for a static integer queue
  - Member functions
    - **`enqueue()`**
    - **`dequeue()`**
    - **`isEmpty()`**
    - **`isFull()`**
    - **`clear()`**

Member Variables for Static Queues

- Five major variables:
  - **`queueArray`** Creates a pointer to queue
  - **`queueSize`** Tracks capacity of queue
  - **`numItems`** Tracks elements in queue
  - **`front`**
  - **`rear`**
    - The variables front and rear are used when our queue "rotates," as discussed earlier

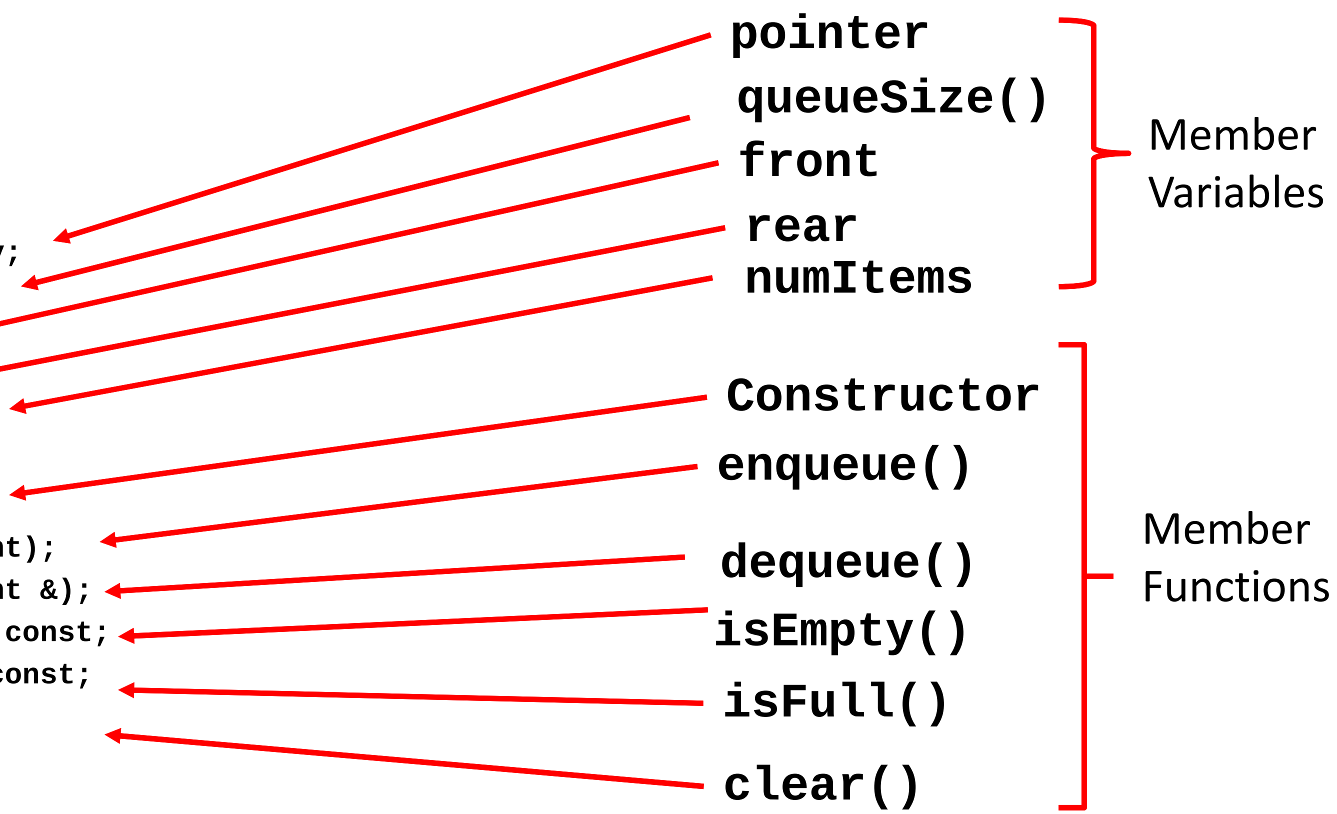
Member Functions for Queues

– **`CONSTRUCTOR`**	Creates a queue
– **`DESTRUCTOR`**	Deletes a queue
– **`enqueue()`**	Adds element to queue
– **`dequeue()`**	Removes element from queue
– **`isEmpty()`**	Is the queue empty?
– **`isFull()`**	Is the queue full?
– **`clear()`**	Empties queue

Static Queue Example

```
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
private:
      int *queueArray;
      int queueSize;
      int front;
      int rear;
      int numItems;
public:
      IntQueue(int);
      void enqueue(int);
      void dequeue(int &);
      bool isEmpty() const;
      bool isFull() const;
      void clear();
};
#endif
```



STL Queues

STL Queues

- Another way to implement a queue is by using the standard library
- An STL queue leverages the pre-existing library to access the data structure
- Much easier to use

STL Queue Example

```
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue
using namespace std;

int main ()
{
  std::queue<int> myqueue;
  int myint;

  std::cout << "Please enter some integers (enter 0 to
end):\n";

  do {
    std::cin >> myint;
    myqueue.push (myint);
  } while (myint);

  std::cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';

  return 0;
}
```