Stacks

- Restricted list structure
  - Dynamic LIFO Storage Structure
    - Size and Contents can change during execution of program
    - Last In First Out (lifo)
- Elements are added to the top and removed from the top
- How do you implement one?
  - What about a dynamic array?
  - What about a linked list?
  - What about a string?

Stack Implementation

- Has two main operations
  - Push
    - adds element to top of stack
  - Pop
    - removes elements from top of stack
  - Both should return a bool to indicate success or failure

More Ideas

- Also nice to include some maintenance functions:
  - Stack () ;
    - set Stack to be empty
  - bool Empty ( ) const;
    - check if stack is empty
  - bool Full ( ) const;
    - check if stack is full
  - bool Push ( const ItemType& item ) ;
    - insert item onto the stack
  - Item Pop ( ) ;
    - remove & return the item at the top of the stack

More Ideas

- Some implementations define:
  - Item Top( ) ;
    - Returns top item in the stack, but does not remove it.
  - Pop() ;
    - In this case removes the top item in the stack, but does not return it.

Implementations

- String Representation
  - Empty Stack == Empty String
  - Top of Stack == End of String
  - String operations are used to implement stack operations
    - Enforces stack behavior on strings of type stack
    - Maps one data structure, (stack), onto another, (string)

- Linked-List Representation
  - top is fixed at the head (tail) of the list
  - Push & Pop operate only on the head (tail) of the list
String Implementation

```cpp
#include <string>
typedef char Item;

class Stack {
    private:
        string stk;
    public:
        bool Empty() const;
        bool Full() const;
        bool Push(const Item& Item);
        Item Pop();
};
```

String Implementation

```cpp
#include "Stack.h"
using namespace std;

bool Stack::Empty() const {
    return stk.empty();
}

bool Stack::Full() const {
    return stk.length() == stk.max_size();
}

bool Stack::Push(const Item& Item) {
    stk = stk + Item;
    return Full();
}

Item Stack::Pop() {
    Item temp;
    int i;
    i = stk.length();
    temp = stk.at(i-1);
    stk.erase(i-1, 1);
    return temp;
}
```

String Implementation

```cpp
//if top() was to be implemented:

Item Stack::Top() {
    Item temp;
    int i;
    i = stk.length();
    temp = stk.at(i-1);
    return temp;
}
```

Linked List Implementation

```cpp
#include "LinkList.h"
//typedef arbitrary Itemtype;
#include "Item.h"
class Stack {
    private:
        LinkList stk;
    public:
        Stack();
        bool Empty() const;
        bool Full() const;
        bool Push(const Itemtype& Item);
        Item Pop();
};
```

Queues

- Restricted (two-tailed) list structure
- Dynamic FIFO Storage Structure
  - Size and Contents can change during execution of program
  - First in First Out
  - Elements are inserted (enqueue) into the rear and retrieved (dequeue) from front.
- Think of waiting in line to check-out of a store.
Queue Implementation

- **Queue**();
- † set queue to be empty
- **bool Empty**();
- † check if queue is empty
- **bool Full**();
- † check if queue is full
- **Enqueue** (const Item& item);
- † Insert item into the queue
- **Item Dequeue**();
- † Remove & return the item at the front of the queue

What about a Front()?

- Some implementations define:
  - **Item Front**();
  - Returns first item in the queue, but does not remove it.
  - **bool Dequeue()**;
  - In this case removes the first item in the queue, but does not return it.
  - What about a Clear()?

Implementation Details

- **Linear Array**: not as easy to implement as it seems.
  - Front or Rear must be fixed at one end of the array
    - Enqueuing or Dequeuing requires inefficient array shifting.
  - OR if not fixed
    - The head and tail move causing problems.

Linear Array Solution

- Make the queue circular.
  - The problem now becomes when is the queue empty and full?
- Solution
  - Leave one cell empty.
  - The trade-off is one empty cell for processing time.

Whaaaaaat?

- Code operations to force array indices to ‘wrap-around’
  - † front = front % MAXQUE;
  - † rear = rear % MAXQUE;

States of the Queue

- front and rear indicies delimit the bounds of the queue contents
- **Enqueue**
  - † Move the que.rear pointer 1 position clockwise & write the element in that position.
- **Dequeue**
  - † Return element at que.front and move que.front one position clockwise
- Count (queue size) is stored and maintained or boolean full status flag maintained.
Array Interface

```cpp
const int MAXQUE = 100;
#include "Item.h"

class Queue {
private:
    int    Front;
    int    Rear;
    Item   Items[MAXQUE];
public:
    Queue();
    bool Empty();
    bool Full();
    void Enqueue (const Item& item);
    Item Dequeue ();
};
```

Array Math

- **Distinct States**
  - Full Queue: 
    \( (\text{que}.\text{rear} + 1) \% \text{MAXQUE} \equiv \text{que}.\text{front} \% \text{MAXQUE} \)
  - Empty Queue: 
    \( \text{que}.\text{rear} \equiv \text{que}.\text{front} \)
  - One-element Queue: 
    \( (\text{que}.\text{front} + 1) \% \text{MAXQUE} \equiv \text{que}.\text{rear} \% \text{MAXQUE} \)

Linked-List Representation

- Queue is a structure containing two pointers: 
  - front: points to the head of the list 
  - rear: points to the end of the list (last node) 
- Enque operates upon the rear pointer, inserting after (before) the last (first) node. 
- Deque operates upon the front pointer, always removing the head (tail) of the list. 
- Empty queue is represented by NULL front & rear pointers

Linked List Interface

```cpp
#include "Queue.h"
#include "LinkList.h"

//typedef arbitrary Item

class Queue {
private:
    LinkList que;
public:
    Queue(); //LinkList constructor
    bool Empty();
    bool Full();
    void Enqueue (const Item& item);
    Item Dequeue ();
};
```

Drop-Out Stack (dos)

- **“Bottomless” Stack**
  - Variation of a regular stack. 
    - No fullstack operation (i.e. a dos can never become full). 
  - “Drop-Out” Stack of size N has following behavior: 
    - Let the integers 1, 2 ... be the first elements PUSHed onto the stack respectively. 
    - After the Nth integer element is PUSHed, integer 1 is at the “bottom” of the stack, with 2 immediately above it. 
    - After the N+1 integer is PUSHed, 1 Drops-Out of the bottom and integer 2 is now at the bottom of the stack. 
  - Note: any element that Drops-Out of the stack never reenters the stack automatically from the bottom due to POPs being performed.
“Double-Ended” Queue

- variation of a regular queue.
- elements can be added and removed at either the rear or front of the queue, but nowhere else in the queue.
- operations: Deque(), Empty(), Full(), EnqRear(), EnqFront(), DeqFront(), DeqRear()
- generalization of both a stack and a queue.

Circular Array

New-Style Header Files

- In general, old-style C++ header files are replaced by new-style headers whose names omit the “.h” suffix. Some headers, such as math.h, were inherited from the C language. In those cases, the new-style headers prefix a “c” to the name and omit the “.h”.

Some differences

<table>
<thead>
<tr>
<th>Old style:</th>
<th>New style:</th>
</tr>
</thead>
<tbody>
<tr>
<td>iostream.h</td>
<td>iostream</td>
</tr>
<tr>
<td>fstream.h</td>
<td>fstream</td>
</tr>
<tr>
<td>string.h</td>
<td>string</td>
</tr>
<tr>
<td>math.h</td>
<td>math</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>stdlib</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Old style:</th>
<th>New style:</th>
</tr>
</thead>
<tbody>
<tr>
<td>iostream</td>
<td>iostream</td>
</tr>
<tr>
<td>standard stuff</td>
<td>same type names, but some subtle differences in implementation</td>
</tr>
<tr>
<td>fstream</td>
<td>fstream</td>
</tr>
<tr>
<td>file stuff; does include iostream.h</td>
<td>file stuff; does NOT include iostream.h</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>C-style char arrays</td>
<td>string object library</td>
</tr>
</tbody>
</table>
Observations

- The new-style headers offer enhanced functionality.
- There are some S/E advantages incorporated into the new-style implementation.
- Therefore, use the new-style approach whenever possible.
- Never, ever, mix old- and new-style headers in the same compilation unit. If possible don’t mix them in the same program.

Namespaces

A namespace is a scope with a name attached. That is:

```cpp
namespace FooSpace {
    typedef struct {
        string Message;
        int Target;
    } Foo;
    const int MaxFoo = 1000;
    int numFoo;
    Foo List[MaxFoo];
};
```

Namespaces may be composed; that is, two with the same name are automatically concatenated by the preprocessor.

Benefits

- Modulization
- You could wrap all those tempting globals into a namespace to protect them
- Global scope is itself considered a namespace, with no name

Namespaces

A namespace is a scope with a name attached. That is:

```cpp
namespace FooSpace {
    typedef struct {
        string Message;
        int Target;
    } Foo;
    const int MaxFoo = 1000;
    int numFoo;
    Foo List[MaxFoo];
};
```

Namespaces may be composed; that is, two with the same name are automatically concatenated by the preprocessor.

Benefits

- Modulization
- You could wrap all those tempting globals into a namespace to protect them
- Global scope is itself considered a namespace, with no name