



Murdoch
UNIVERSITY

Data Structures and Abstractions

Stacks and Queues

Lecture 8



Temporary Storage

- When processing it is often necessary to put data into temporary storage.
- This can happen, for example, when:
 - processing events in an event-driven OS;
 - processing email in and out of a server;
 - scheduling jobs on a main-frame;
 - doing calculations;
 - sorting or merging;
- The most common data structures for temporary storage are **stacks**, **queues**, **heaps** and **priority queues**.

Stacks

- Stacks are ADS that emulate, for example, a stack of books: you can only put things on or take them off at the top.
- There are only two operations allowed on a stack: **[1]**
 - **Push (something on to it)**
 - **Pop (something off it)**
- Plus two query methods:
 - **Empty ()**
 - **Full () // optional**
- Since the last thing on is the first thing off, they are known as LIFO (Last In, First Out) data structures, or sometimes FILO (First In, Last Out).
- In essence, a stack reverses the order of the data.

Stack Implementation

- Stacks can be implemented any way you want, the encapsulation of the container used ensures that it does not matter.
- As long as it only has **Push**, **Pop**, **Empty** and (optionally) **Full**, then it is a stack.
- Most commonly they are implemented using arrays, lists or an STL structure.
- If none of these exactly fit the required abstraction that we are after, they should be encapsulated inside our own Stack. [1]

Error Conditions for Stacks

- If you try to **Push ()** onto a stack that has no free memory, then you get overflow.
- If you try to **Pop ()** from an empty stack then you have underflow.
- So **Push ()** and **Pop ()** return a boolean to indicate if one of these errors has occurred.

Stack Example (Animation)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (10)

Array Implementation



m_top

Linked List Implementation

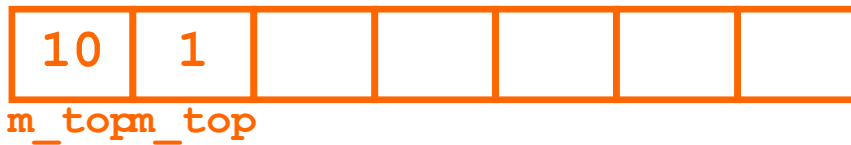
m_top



Stack Example (Animation)

Push (1)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (23)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Pop (num)

num 23

Array Implementation



Linked List Implementation

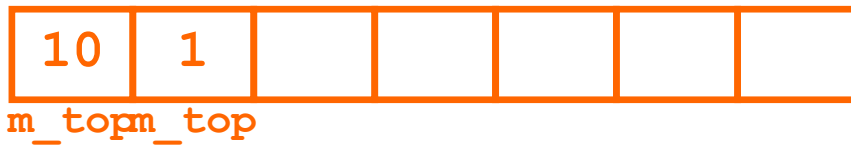


Stack Example (Animation)

Pop (num)

num 1

Array Implementation



Linked List Implementation



Stack Example (Animation)

Pop (num)

num 10

Array Implementation



m_top

Linked List Implementation

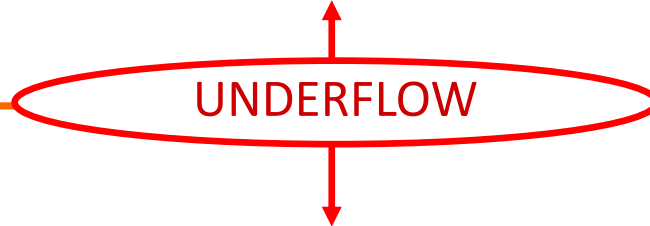


Stack Example (Animation)

Pop (num)

num

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (12)

Array Implementation



m_top

Linked List Implementation

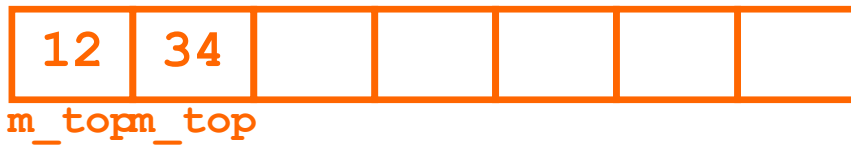
m_top



Stack Example (Animation)

Push (34)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (23)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (36)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (98)

Array Implementation



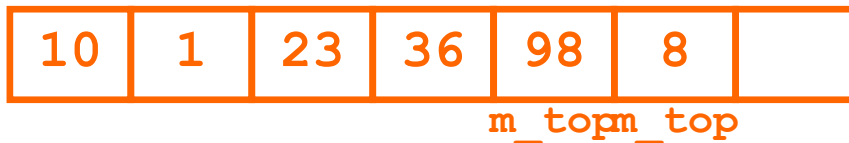
Linked List Implementation



Stack Example (Animation)

Push (8)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (76)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (66)

Array Implementation



Linked List Implementation



Array Push Algorithm

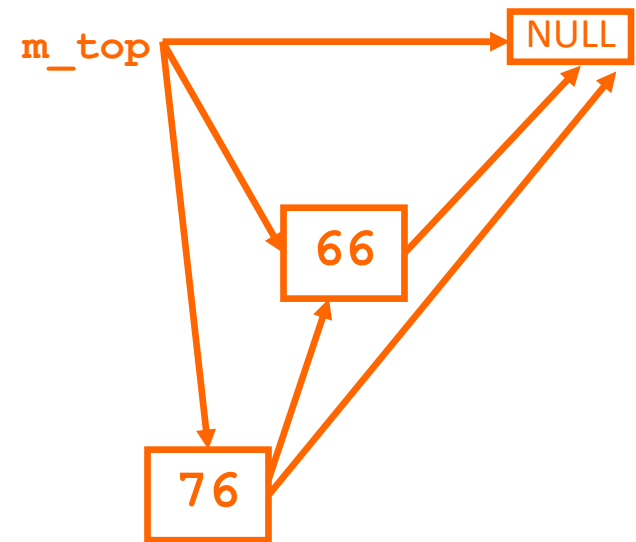
- PUSH (DataType data): boolean
- IF $m_top \geq ARRAY_SIZE - 1$
- return FALSE
- ELSE
- Increment m_top
- Place data at position m_top
- return TRUE
- ENDIF
- END Push

Array Pop Algorithm

- POP (DataType data): boolean
- IF m_top < 0
- return FALSE
- ELSE
- data = data at position m_top
- Decrement m_top
- return TRUE
- ENDIF
- END Pop

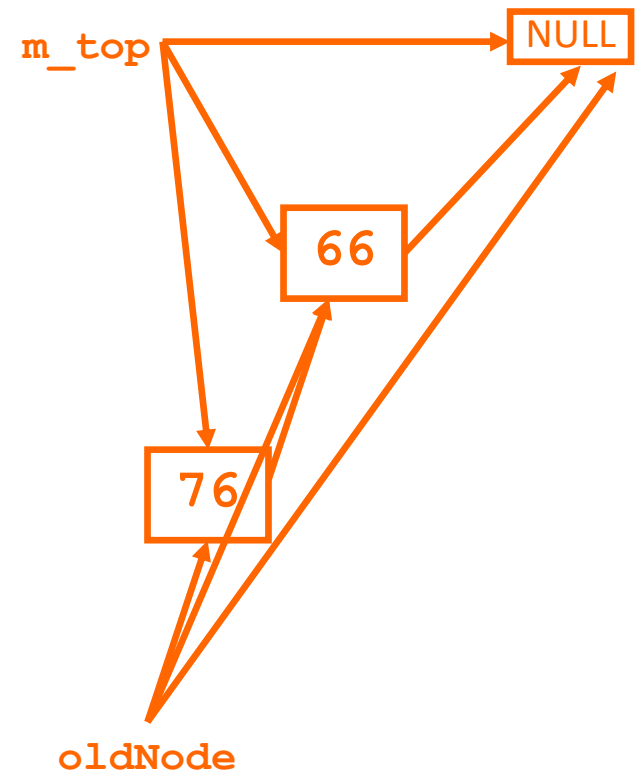
Linked List Push Algorithm

- PUSH (DataType data): boolean
- IF there is memory on the heap
- Get newNode from the heap
- Put data into the newNode
- IF m_top is NULL
- m_top = newNode
- ELSE
- newNode.next = m_top
- m_top = newNode
- ENDIF
- return TRUE
- ELSE
- return FALSE
- ENDIF
- END Push



Linked List Pop Algorithm

- POP (DataType data): boolean
- IF m_top == NULL
- return FALSE
- ELSE
- data = m_top.data
- oldNode = m_top
- m_top = oldNode.next
- release oldNode memory
- oldNode = NULL
- return TRUE
- ENDIF
- END Pop



Using the STL

- The other possibility is to use one of the STL structures.
- If using a vector or list, then the algorithms above barely change.
- However, remember that the structure *must* still be encapsulated in a class, otherwise it will not have just the **Pop ()** and **Push ()** that it is supposed to have.
- Finally, there is the STL stack class, (requiring `<stack>`), which is obviously the **best** STL class to use your Stack class.
- STL stack is an adapted STL container (container adapter) for special use as a stack. No iterators are provided.
- However, even this must be encapsulated if it does not conform to our abstraction of what a stack should be (pointed out earlier and see slide notes from earlier). [1]

Features of the STL stack which don't fit in with our Abstraction

1. Its **pop ()** method, only removes the data, it does *not* pass it back to the calling method.
 2. In fact there is a **top ()** method which returns the data (by reference) at the top of the stack.
 3. Neither **pop ()** , **top ()** nor **push ()** return a boolean: overflow and underflow must be checked separately.
- Given the abstraction we are after, even the STL stack must be encapsulated.

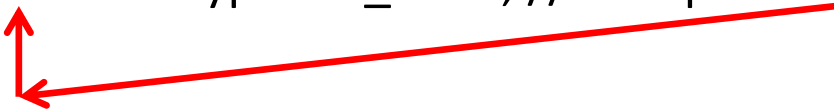
Stack Header File using STL stack

- `// Stack.h`
- `//`
- `// Stack class`
- `// Version`
- `// Nicola Ritter`
- `// modified smr`
- `//-----`
- `// NO I/O HERE. LET THE CLIENT DEAL WITH I/O`
- `#ifndef MY_STACK`
- `#define MY_STACK`

- `//-----`

- `#include <stack>`
- `#include <iostream>`
- `using namespace std;`

- `template <class DataType>`
- `class Stack`
- `{`
- `public:`
- `Stack () {};`
- `~Stack () {};`
- `bool Push(const DataType &data);`
- `bool Pop (DataType &data);`
- `bool Empty () const {return m_stack.empty();}`
- `private:`
- `stack<DataType> m_stack; // encapsulated STL stack`
- `};`



- `//-----`
- `// It is a template, so we have to put all the code`
- `// in the header file`
- `//-----`
- `template<class DataType>`
- `bool Stack<DataType>::Push(const DataType &data)`
- `{`
- `bool okay = true;`
- `try`
- `{`
- `m_stack.push(data);`
- `}`
- `catch (...)`
- `{`
- `okay = false;`
- `}`
- `return okay;`
- `}`

```
• //-----  
• template<class DataType>  
• bool Stack<DataType>::Pop(DataType &data)  
• {  
•     if (m_stack.size() > 0)  
•     {  
•         data = m_stack.top();  
•         m_stack.pop();  
•         return true;  
•     }  
•     else  
•     {  
•         return false;  
•     }  
• }  
• //-----  
• #endif
```

Simple Example of Stack Use

- `// StackTest.cpp`
- `//`
- `// Tests Stack classes`
- `//`
- `// Nicola Ritter`
- `// Version 01`
- `// modified smr`
- `// Reverse a string`
- `//`
- `//-----`

- `#include <iostream>`
- `#include <string>`
- `#include "Stack.h" ← //Our stack`

- `using namespace std;`

- //-----
- typedef Stack<char> CharStack;
- void Input (string &str);
- void Reverse (const string &str, CharStack &temp);
- void Output (CharStack &temp); // const – check what it
- //does first??

- //-----
- int main()
- {
- string str;
- CharStack temp;
- Input (str);
- Reverse (str, temp); [1]
- Output (temp);
- cout << endl;
- return 0;
- }

- `//-----`
- `void Input (string &str)`
- `{`
- `cout << "Enter a string, then press <Enter>: ";`
- `getline(cin,str);`
- `}`

- `//-----`
- `void Reverse (const string &str, CharStack &temp)`
- `{`
- `bool okay = true;`
- `for (int index = 0; index < str.length() && okay; index++)`
- `{`
- `okay = temp.Push(str[index]);`
- `}`
- `}`

- //-----
- void Output (CharStack &temp) // would const work?
- {
- bool okay;
- char ch;
- cout << "Your string reversed is: ";
- okay = temp.Pop(ch);
- while (okay)
- {
- cout << ch;
- okay = temp.Pop(ch);
- }
- cout << endl;
- }
- //-----

Screen Output

- Enter a string, then press <Enter>: This is a string
- Your string reversed is: gnirts a si sihT
- Press any key to continue . . .

Advantages of Implementations

- It is assumed for each of the containers below, that **our Stack** encapsulates it.

Array	Linked List	list/vector/deque	STL stack
Easy to code	Full memory control	Easy to code	Easier to code compared to all the others.
	Memory 'never' runs out.	Memory 'never' runs out.	Memory 'never' runs out.

Disadvantages of Implementations

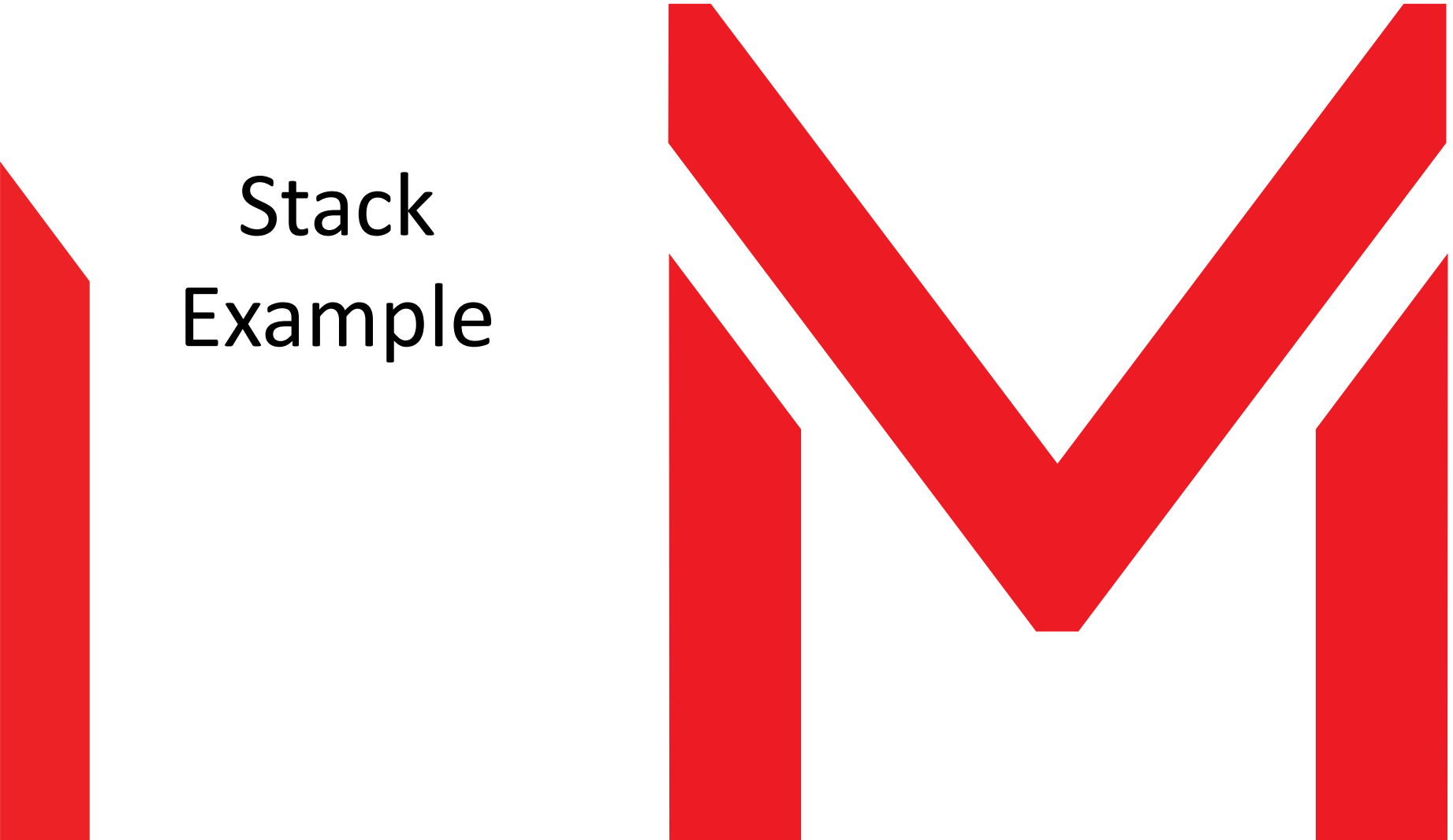
- It is assumed for each of the containers below, that **our Stack** encapsulates it.

Array	Linked List	list/vector/deque	STL stack
Can run out of space easily.	More difficult to code as it uses pointers.	Excess code sitting 'behind' the implementation.	Excess code sitting 'behind' the implementation.
		Only available in with some languages. e.g C++ has STL, Java has Java collections framework	Only available with some languages. e.g. C++ has STL, Java has Java collections framework

Readings

- Textbook: Stacks and Queues, entire section on Stacks.
- For amore details of Stacks with some level of language independence, see the reference book, Introduction to Algorithms section on “Stacks and Queues” in the chapter on “Elementary Data Structures”. You will see how removed the STL stack is from the abstract stack. We want the abstract level – see earlier lecture notes on level of abstractions.
- Textbook: Standard Template Library, section on Container Adapters
- Library Ereserve: Deitel & Deitel, [C++ how to program \[ECMS\]. Chapter 15](#) part A. [1]

Stack Example



A Calculator

It is possible to use two stacks to do simple one line calculations.

- The first stack stores operators (characters) that have not yet been performed.
- We will start with just + - * /.
- The second stack stores the numbers being operated upon.
- Therefore we are trying to find the answer to something like
$$10 + 8 / 2 - 6 * 5$$
- What *is* the answer to this?
- For simplicity's sake, we will assume integer input, but floating point output.

Using Diagrams

- Try this yourself or in a group.
- Draw up an array (set of boxes) to represent the string:

“10 + 8 / 2 – 6 * 5”

With one number (not digit but the whole integer) or operation in each box

- Draw an array for the number stack and one for the operation stack
- Figure how to do it with diagrams first.

Test Data

- The next thing, of course, is to design the test data: build the test plan.
- Construction of the test plan occurs **before any code is written**.
 - The test plan is written once you have analysed the problem to be solved
 - Gets added to as software development progresses.
- I came up with over 50 possibilities that should be tested in the test plan!
 - See the spreadsheet with testdata related to this lecture note.
 - More extended examples of testing in the “*testing 4 later units*” folder.
 - You must perform regression testing. This can be “painful” so think of ways to automate the testing process. You don’t have to use it in this unit but should in later units. Various approaches are used in industry.
- **Ignore advice about test plans and testing at your own peril.**

Top Level Algorithm

- We are used to the **infix** notation: $2 + 3$ and using this notation means that when you want to override operator precedence, you need to use () as in $(2 + 3) * 5$.
- In Polish (discovered by a Polish logician Jan Lukasiewicz) notation (prefix notation), there is no need to use (). **Prefix** notation: $+ 2 3$
- Someone else came along later with something called **Reverse Polish Notation** (postfix form). **Postfix** notation: $2 3 +$
- With RPN, there is an additional advantage in that operators are in the correct order for digital computers.
- RPN examples: $2 3 + 5 *$
- So think this way: push the numbers on the stack until you get an operator. Then pop the last two numbers of the stack and apply the operator. Put the result back on the stack and repeat the whole process. This is easy. The question is how to convert from infix to RPN (postfix).

[1]

Top Level Algorithm

- Assuming that we have the equation in a string, try to design an algorithm that will do the top level of process control of the string....
- Use what you understood when you tried to figure it out using a diagram. If you have forgotten go through using diagrams again.
- In other words, most of it will be enclosed in a loop
 - WHILE more characters
 - ENDWHILE
- Remember to keep it a *control* function: put off until later working out how things are actually done.
- In other words, concentrate on *what* not *how*.
- Normally (of course), we would be designing, coding and testing in parallel.

Next...

- Next work on each part of the algorithm that you have got, as a *what* not a *how*.
- Ideally, you should do this with a group of two or three other people. But you can always give it a go on your own.

Readings

- Textbook: Stacks and Queues, section on Application of Stacks: Postfix Expressions Calculator.
 - The RPN calculator is described in the above section.
 - There are a number of calculators which accept RPN entry and therefore make calculations of long expressions easy – less keys to press to get the same result.

Queues



Queues

- Queues are ADS that emulate, for example, a queue at the movies: you can only get on the back of the queue, and off at the front of the queue.
- There are only two operations shown for a queue: [1]
 - **Enqueue** (something on to it)
 - **Dequeue** (something off it)
- Plus two query methods:
 - **Empty** ()
 - **Full** ()
- Since the last thing on is the last thing off, they are known as FIFO (First In, First Out) data structures, or sometimes LIFO (Last In, Last Out).

Queue Implementation

- Queues can be implemented any way you want, the encapsulation of the container used ensures that it does not matter.
- As long as it only has **Enqueue**, **Dequeue**, **Empty** and **Full**, then it is a minimal queue.
- Most commonly they are implemented using arrays, lists or an STL structure, with the STL Queue being more relevant.
- However since none of these exactly fit the required minimal abstraction we are after, they should always be encapsulated.

Error Conditions for Queues

- If you try to **Enqueue** () onto a queue that has no free memory, then you get *overflow*.
- If you try to **Dequeue** () from an empty queue then you have *underflow*.
- So **Enqueue** () and **Dequeue** () return a boolean to indicate if one of these errors has occurred.
- In the animation that follows, two approaches are used.
 - The internal container is an array
 - The internal container is a linked list

Queue Example (Animation)

Array Implementation



m_size

0

m_front

m_back

Linked List Implementation

m_front

m_back

NULL

Queue Example (Animation)

Enqueue (10)

Array Implementation



m_front
m_back

Linked List Implementation



Queue Example (Animation)

Enqueue (1)

Array Implementation



m_size

2

m_front

m_back

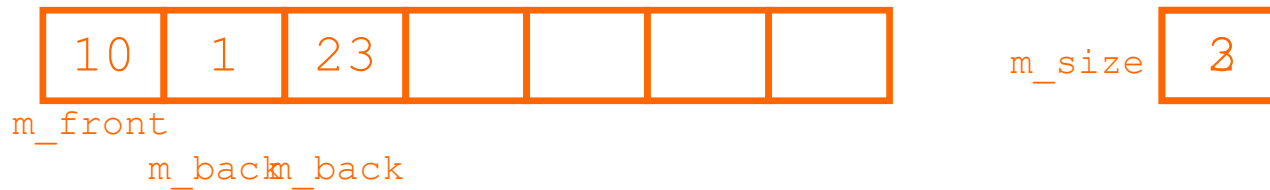
Linked List Implementation



Queue Example (Animation)

Enqueue (23)

Array Implementation



Linked List Implementation

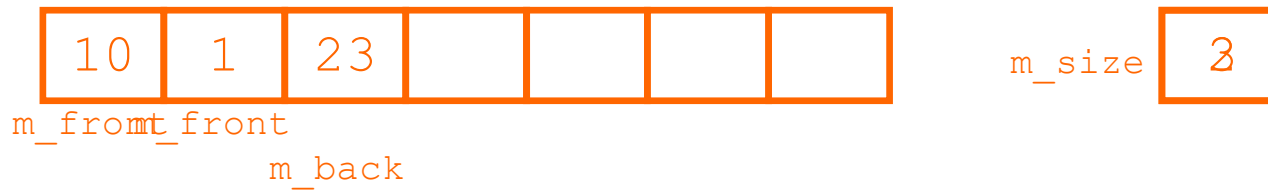


Queue Example (Animation)

Dequeue (num)

num 10

Array Implementation



Linked List Implementation

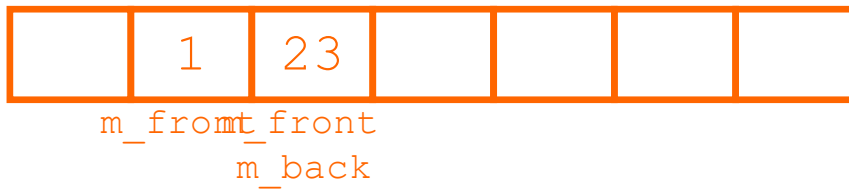


Queue Example (Animation)

Dequeue (num)

num 1

Array Implementation



`m_size` 2

Linked List Implementation



Queue Example (Animation)

Dequeue (num)

num 23

Array Implementation



m_size

0

m_front
m_back

m_front
m_back

Linked List Implementation

m_front

m_back

23

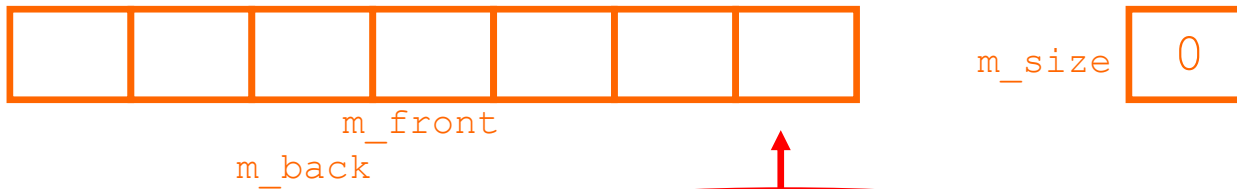
NULL

Queue Example (Animation)

Dequeue (num)

num

Array Implementation



UNDERFLOW

Linked List Implementation

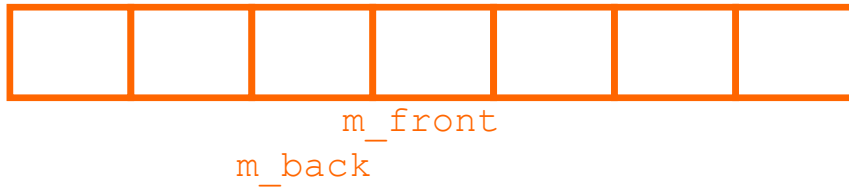


Queue Example (Animation)

Dequeue (num)

num

Array Implementation



m_size

Linked List Implementation



Queue Example (Animation)

Enqueue (12)

Array Implementation



m_size

0

m_front
m_back

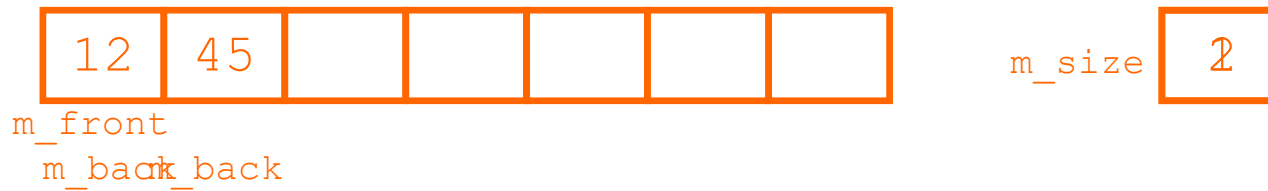
Linked List Implementation



Queue Example (Animation)

Enqueue (45)

Array Implementation



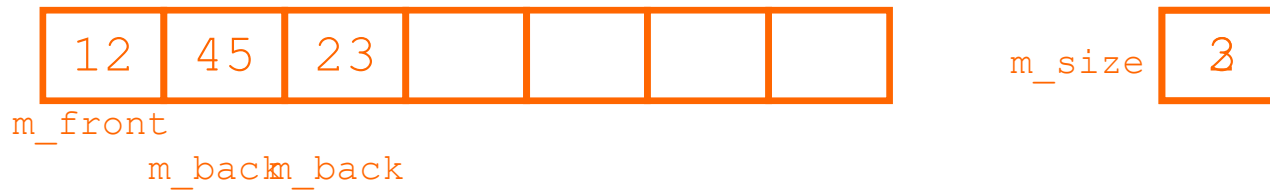
Linked List Implementation



Queue Example (Animation)

Enqueue (23)

Array Implementation



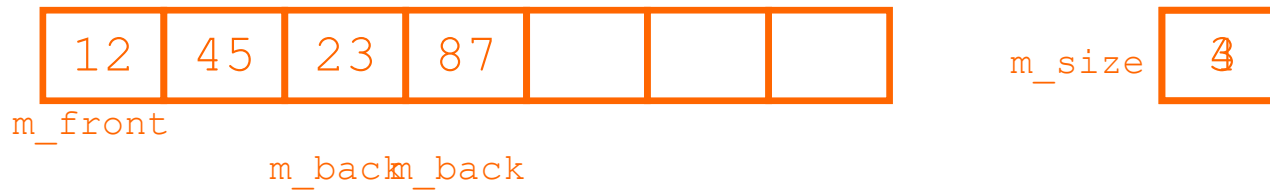
Linked List Implementation



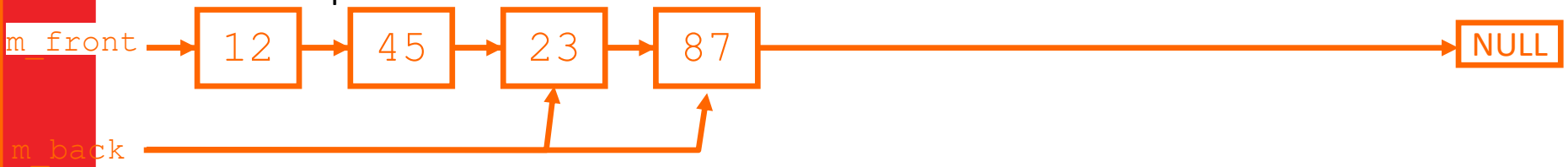
Queue Example (Animation)

Enqueue (87)

Array Implementation



Linked List Implementation



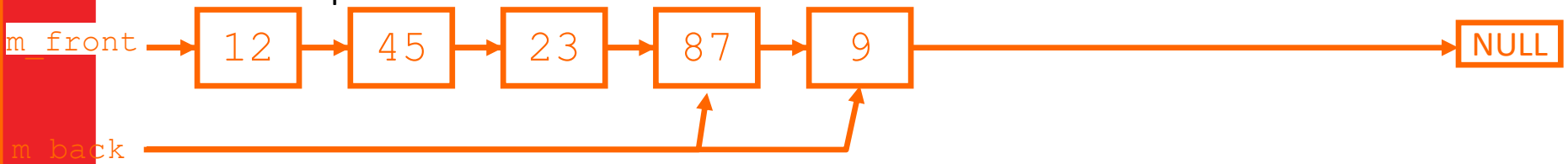
Queue Example (Animation)

Enqueue (9)

Array Implementation



Linked List Implementation



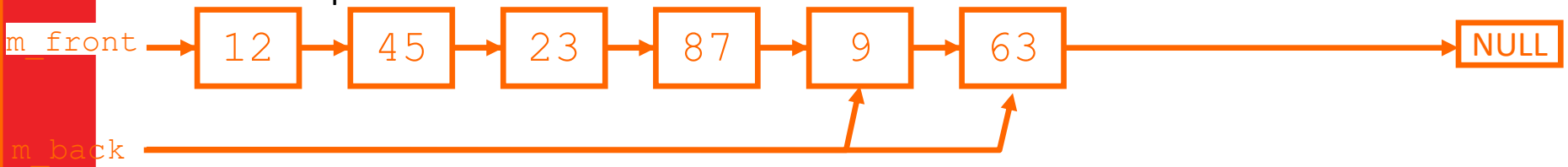
Queue Example (Animation)

Enqueue (63)

Array Implementation



Linked List Implementation



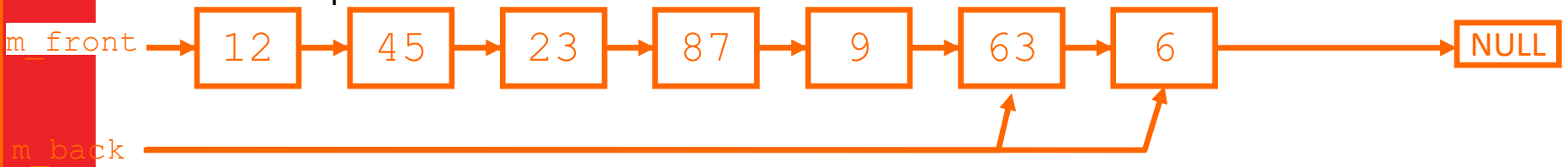
Queue Example (Animation)

Enqueue (6)

Array Implementation



Linked List Implementation



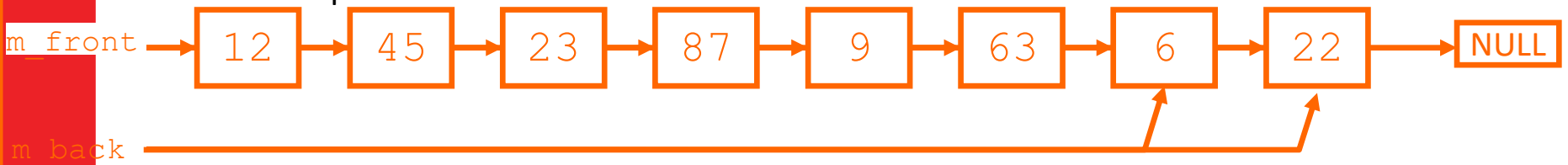
Queue Example (Animation)

Enqueue (22)

Array Implementation



Linked List Implementation



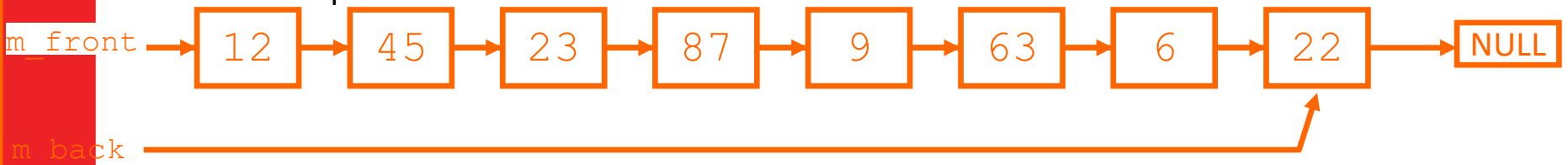
Queue Example (Animation)

Dequeue (num)

Array Implementation



Linked List Implementation



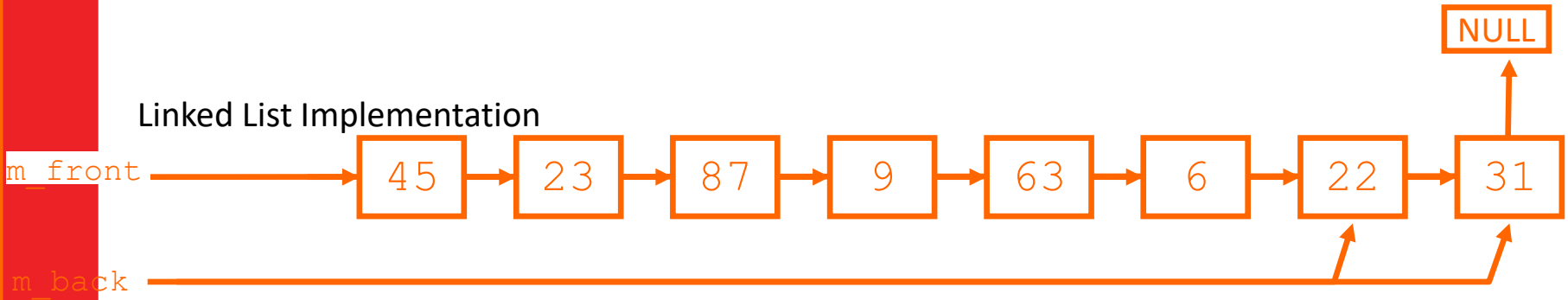
Queue Example (Animation)

Enqueue (31)

Array Implementation



Linked List Implementation



Array Enqueue Algorithm

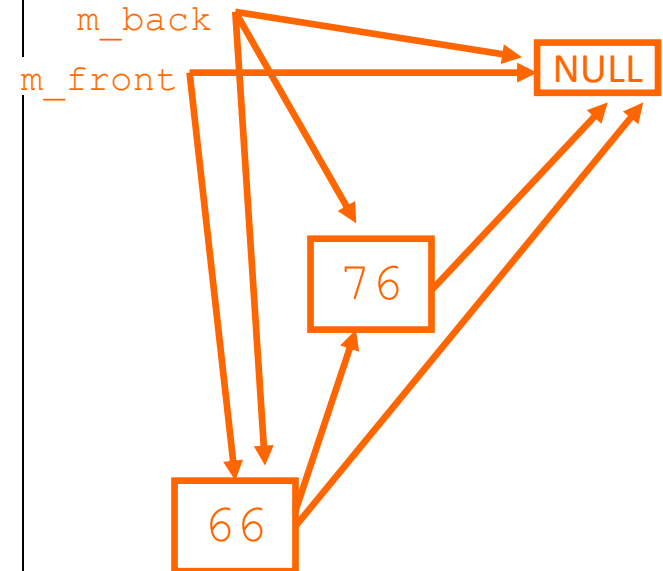
- ENQUEUE (DataType data): boolean
- IF $m_size \geq ARRAY_SIZE - 1$
- return FALSE
- ELSE
- Increment m_size
- Increment $m_back \text{ MOD } ARRAY_SIZE$ [1]
- Place data at position m_back
- return TRUE
- ENDIF
- END Enqueue

Array Dequeue Algorithm

- DEQUEUE (DataType data): boolean
- IF m_size == 0
- return FALSE
- ELSE
- data = data at position m_front
- Increment m_front MOD ARRAY_SIZE
- Decrement m_size
- IF m_size == 0
- m_front = -1
- m_back = -1
- ENDIF
- return TRUE
- ENDIF
- END Dequeue

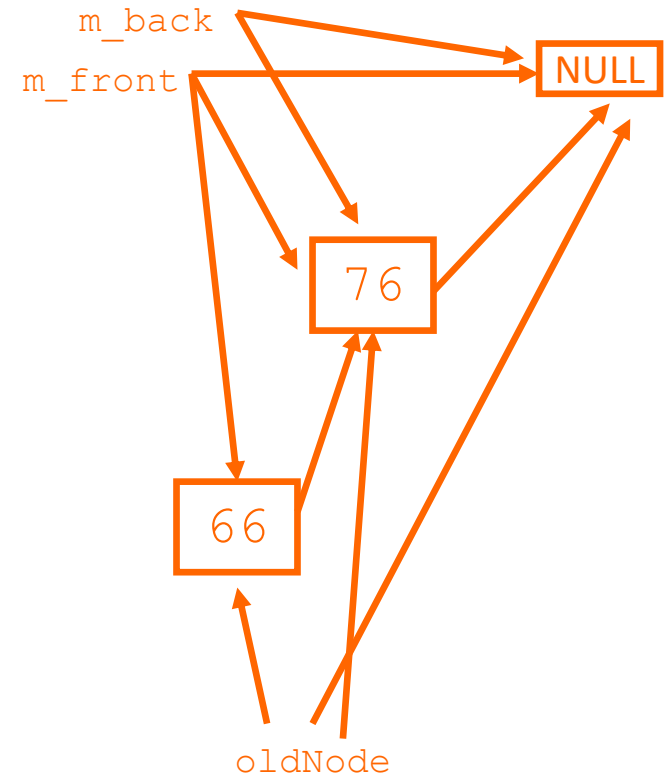
Linked List Enqueue Algorithm

- ENQUEUE (DataType data): boolean
- IF there is memory on the heap
- Get newNode from the heap
- IF m_front is NULL
- m_front = newNode
- m_back = newNode
- ELSE
- m_back.next = newNode
- m_back = newNode
- ENDIF
- return TRUE
- ELSE
- return FALSE
- ENDIF
- END Enqueue



Linked List Dequeue Algorithm

- DEQUEUE (DataType data): boolean
- IF m_front == NULL
- return FALSE
- ELSE
- data = m_front.data
- oldNode = m_front
- IF m_back == m_front
- m_front = NULL
- m_back = NULL
- ENDIF
- m_front = oldNode.next
- release oldNode memory
- oldNode = NULL
- return TRUE
- ENDIF
- END Dequeue



Using the STL

- The other possibility is to use one of the STL structures.
- If using a vector or list, then the algorithms above barely change.
- However, remember that the structure *must* still be encapsulated in a class, otherwise it will not have just the **Enqueue ()** and **Dequeue ()** that it is supposed to have.
- Finally, there is the STL queue class, (requiring `<queue>`), **which is obviously the best STL class to use.**
 - However, even this should be encapsulated, because it does not conform to the standard queue description! **[1]**

Non Standard Features of the STL

queue

1. Its Enqueue method is called `push ()` [1]
 2. Its Dequeue method is called `pop ()` .
 3. Its `pop ()` method, only removes the data, it does *not* pass it back to the calling method.
 4. In fact there is a `front ()` method which returns a *reference* to the front of the queue.
 5. Neither `dequeue ()` , `front ()` nor `enqueue ()` return a boolean: overflow and underflow must be checked separately.
- Therefore it is best that the STL queue must be encapsulated. [2]

Queue Header File using STL queue

- `// Queue.h`
- `//`
- `// Queue class`
- `//`
- `// See actual code provided in the zip file`
- `//-----`
- `#ifndef MY_QUEUE`
- `#define MY_QUEUE`
- `//-----`
- `#include <queue> // for the STL queue`
- `#include <iostream>`
- `using namespace std;`
- `//-----`

- `template <class T>`
- `class Queue // minimal and complete`
- `{`
- `public:`
- `Queue () {};`
- `~Queue () {};`
- `bool Enqueue(const T &data);`
- `bool Dequeue (T &data);`
- `bool Empty () const {return m_queue.empty();}`
- `private:`
- `queue<T> m_queue; // encapsulates STL queue`
- `};`

- `//-----`
- `// It is a template, so we have to put all the code`
- `// in the header file`
- `//-----`

- `template<class DataType>`
- `bool Queue<DataType>::Enqueue(const DataType &data)`
- `{`
- `bool okay = true;`
- `try`
- `{`
- `m_queue.push(data); // calls STL queue method`
- `}`
- `catch (...)`
- `{`
- `okay = false;`
- `}`

- `return okay;`
- `}`

```
• //-----  
• template<class DataType>  
• bool Queue<DataType>::Dequeue(DataType &data)  
• {  
•     if (m_queue.size() > 0)  
•     {  
•         data = m_queue.front();  
•         m_queue.pop();  
•         return true;  
•     }  
•     else  
•     {  
•         return false;  
•     }  
• }  
• //-----  
• #endif
```


Simple (but interesting) Example of Queue Use

```
• // IntQueueTest Program
• //
• // Version
• // original by - Nicola Ritter
• // modified by smr
• //
• //-----

• #include "Queue.h"
• #include <iostream>
• #include <ctime>
• using namespace std;

• //-----

• const int EVENT_COUNT = 20;
• const int MAX_NUM = 100;

• //-----

• typedef Queue<int> IntQueue;
• typedef Queue<float> FloatQueue;

• //-----
```

- `void DoEvents ();`
- `void AddNumber (IntQueue &aqueue);`
- `void DeleteNumber (IntQueue &aqueue);`
- `void TestOverflow();`

- `//-----`

- `int main()`
- `{`
- `DoEvents ();`

- `cout << endl;`
- `system("Pause");`
- `cout << endl;`

- `TestOverflow();`

- `cout << endl;`
- `return 0;`
- `}`

- `//-----`

- `void DoEvents ()`
- `{`
- `IntQueue aqueue;`
- `// Seed random number generator`
- `srand (time(NULL));`
- `for (int count = 0; count < EVENT_COUNT; count++)`
- `{`
- `// Choose a random event`
- `int event = rand() % 5;`
- `// Do something based on that event type, biasing`
- `// it towards Adding`
- `if (event <= 2) // event = 0, 1 or 2`
- `{`
- `AddNumber (aqueue);`
- `}`
- `else // event = 3 or 4`
- `{`
- `DeleteNumber (aqueue);`
- `}`
- `}`
- `// aqueue is local so destructor for aqueue is called when routine finishes.`
- `}`
- `//-----`

- `void AddNumber (IntQueue &aqueue)`
- `{`
- `// Get a random number`
- `int num = rand() % (MAX_NUM+1);`
- `// Try adding it, testing if the aqueue was full`
- `if (aqueue.Enqueue(num))`
- `{`
- `cout.width(3);`
- `cout << num << " added to the queue" << endl;`
- `}`
- `else`
- `{`
- `cout.width(3);`
- `cout << "Overflow: could not add " << num << endl;`
- `}`
- `}`
- `//-----`

- `void DeleteNumber (IntQueue &aqueue)`
- `{`
- `int num;`
- `if (aqueue.Dequeue(num))`
- `{`
- `cout.width(3);`
- `cout << num << " deleted from the queue" << endl;`
- `}`
- `else`
- `{`
- `cout << "IntQueue is empty, cannot delete" << endl;`
- `}`
- `}`
- `//-----`

- `void TestOverflow()`
- `{`
- `Queue<double> mqueue;`
- `int count = 0;`
- `// Keeping adding numbers until we run out of space, will take //time`
- `while (mqueue.Enqueue(count))`
- `{`
- `count++;`
- `cout << "Count is " << count << endl;`
- `}`
- `}`
- `//-----`

Screen Output

- IntQueue is empty, cannot delete
- 79 added to the queue
- 79 deleted from the queue
- IntQueue is empty, cannot delete
- 2 added to the queue
- 2 deleted from the queue
- IntQueue is empty, cannot delete
- 72 added to the queue
- 72 deleted from the queue
- 88 added to the queue
- 88 deleted from the queue
- 22 added to the queue
- 5 added to the queue
- 22 deleted from the queue
- 37 added to the queue
- 46 added to the queue
- 74 added to the queue
- 58 added to the queue
- 5 deleted from the queue
- 37 deleted from the queue

- Press any key to continue . . .

```
Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
Count is 6
Count is 7
Count is 8
Count is 9
Count is 10
Count is 11
Count is 12
Count is 13
Count is 14
Count is 15
Count is 16
Count is 17
...
```

At 300,000 I stopped: it was
just too boring

Advantages of Implementations

- It is assumed for each of the containers below, that the Queue encapsulates it in its own class.

Array	Linked List	list/vector/deque	STL queue
Available in all languages.	Full memory control	Easy to code	Easiest to code
	Memory 'never' runs out. [1]	Memory 'never' runs out. [1]	Memory 'never' runs out. [1]

Disadvantages of Implementations

- It is assumed for each of the containers below, that the Queue encapsulates it in its own class.

Array	Linked List	list/vector/deque	STL queue
Can run out of space easily.	Difficult to code as it uses pointers.	Excess code sitting 'behind' the implementation, increasing the size of the program.	Excess code sitting 'behind' the implementation, increasing the size of the program.
Messy to code, because m_back ends up in front of m_front		Available in some languages like Java, C++. [1]	Available in some languages like Java, C++. [1]

Readings

- Textbook: Stacks and Queues, entire section on Queues
- STL Queue: <http://en.cppreference.com/w/cpp/container/queue>
- For more details of Queues with some level of language independence, see the reference book, *Introduction to Algorithms* section on “Stacks and Queues” in the chapter on “Elementary Data Structures”. You will see that how removed the STL queue is from the abstract queue we are after.
<https://prospero.murdoch.edu.au/record=b2794699~S10>