Chapter 4
ADTs Stack and Queue
Stacks of Coins and Bills
What is a Stack?

• **Logical (or ADT) level:** A stack is an ordered group of homogeneous items (elements), in which the removal and addition of stack items can take place only at the top of the stack.

• A stack is a **LIFO** “last in, first out” structure.
Stacks of Boxes and Books
Stack ADT Operations

- **MakeEmpty** -- Sets stack to an empty state.
- **IsEmpty** -- Determines whether the stack is currently empty.
- **IsFull** -- Determines whether the stack is currently full.
- **Push (ItemType newItem)** -- Throws exception if stack is full; otherwise adds newItem to the top of the stack.
- **Pop** -- Throws exception if stack is empty; otherwise removes the item at the top of the stack.
- **ItemType Top** -- Throws exception if stack is empty; otherwise returns a copy of the top item
ADT Stack Operations

Transformers
- Push
- Pop

Observers
- IsEmpty
- IsFull
- IsFull

change state

observe state
// Class specification for Stack ADT in file StackType.h

class FullStack // Exception class thrown by
{ // Push when stack is full
};

class EmptyStack // Exception class thrown by
{ // Pop and Top when stack is empty
};

#include "ItemType.h"

class StackType
{
public:

    StackType( ); // Class constructor.
    bool IsFull () const;
    // Function: Determines whether the stack is full.
    // Pre: Stack has been initialized
    // Post: Function value = (stack is full)
bool IsEmpty() const;
// Function: Determines whether the stack is empty.
// Pre:    Stack has been initialized.
// Post:   Function value = (stack is empty)
void Push( ItemType item );
// Function: Adds newItem to the top of the stack.
// Pre: Stack has been initialized.
// Post: If (stack is full), FullStack exception is thrown;
//       otherwise, newItem is at the top of the stack.
void Pop();
// Function: Removes top item from the stack.
// Pre: Stack has been initialized.
// Post: If (stack is empty), EmptyStack exception is thrown;
//       otherwise, top element has been removed from stack.
ItemType Top();
// Function: Returns a copy of top item on the stack.
// Pre: Stack has been initialized.
// Post: If (stack is empty), EmptyStack exception is thrown;
//       otherwise, top element has been removed from stack.
private:
  int top;
  ItemType items[MAX_ITEMS];
};
// File: StackType.cpp

#include "StackType.h"
#include <iostream>
StackType::StackType( )
{
    top = -1;
}
bool StackType::IsEmpty() const
{
    return (top == -1);
}

bool StackType::IsFull() const
{
    return (top == MAX_ITEMS-1);
}
void StackType::Push(ItemType newItem)
{
    if( IsFull() )
        throw FullStack();
    top++;
    items[top] = newItem;
}

void StackType::Pop()
{
    if( IsEmpty() )
        throw EmptyStack();
    top--;
}

ItemType StackType::Top()
{
    if (IsEmpty())
        throw EmptyStack();
    return items[top];
}
StackType class

Private data:

- top
- [MAX_ITEMS-1]
  - ..
  - [2]
  - [1]
- items
  - [0]

Methods:
- Push
- Pop
- IsFull
- isEmpty
- StackType
Tracing Client Code

Private data:

- `top`:

  - `[MAX_ITEMS-1]`
  - `:`
  - `:`
  - `[2]`
  - `[1]`

- `items`:

  - `[0]`

```cpp
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ) )
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
    { letter = charStack.Top();
      charStack.Pop(0) }
```
Tracing Client Code

```java
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ) )
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
    { letter = charStack.Top();
      charStack.Pop(0)
    }
```

Private data:

```
[ 0 ]  [ 1 ]  [ 2 ]
items    :         :
[MAX_ITEMS-1]
top     -1
```
```java
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty() )
    charStack.Pop();
charStack.Push('K');
while ( !charStack.IsEmpty() )
    { letter = charStack.Top();
      charStack.Pop(0);
    }
```
Tracing Client Code

```c
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ) )
  charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
  { letter = charStack.Top();
    charStack.Pop(0);}
```

Private data:

<table>
<thead>
<tr>
<th>top</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[ 0 ]</td>
</tr>
<tr>
<td></td>
<td>‘V’</td>
</tr>
<tr>
<td></td>
<td>[ 1 ]</td>
</tr>
<tr>
<td></td>
<td>‘C’</td>
</tr>
<tr>
<td></td>
<td>[ 2 ]</td>
</tr>
<tr>
<td>[MAX_ITEMS-1]</td>
<td></td>
</tr>
</tbody>
</table>
Tracing Client Code

```
char  letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ) )
    charStack.Pop( );
charStack.Push('K');
while ( !charStack.IsEmpty( ) )
    { letter = charStack.Top();
    charStack.Pop(0) }
```

<table>
<thead>
<tr>
<th>items</th>
<th>[0]</th>
<th>'V'</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>'C'</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>'S'</td>
<td></td>
</tr>
</tbody>
</table>

Private data:
- **top**: 2
- **[MAX_ITEMS-1]**:
  - [2]: 'S'
  - [1]: 'C'
  - [0]: 'V'

- **letter**: 'V'
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
  charStack.Pop(0);\}
Tracing Client Code

```plaintext
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ) )
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
    { letter = charStack.Top();
      charStack.Pop(0)
    }
```

Private data:

<table>
<thead>
<tr>
<th>items</th>
<th>[ 0 ]</th>
<th>'V'</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 1 ]</td>
<td>'C'</td>
<td></td>
</tr>
<tr>
<td>[ 2 ]</td>
<td>'S'</td>
<td></td>
</tr>
<tr>
<td>top</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>[MAX_ITEMS-1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty() )
    charStack.Pop();
charStack.Push('K');
while (!charStack.IsEmpty())
    { letter = charStack.Top();
      charStack.Pop(0); }

Private data:

letter: 'V'

<table>
<thead>
<tr>
<th>top</th>
<th>[MAX_ITEMS-1]</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>'K'</td>
<td>'V'</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>[1]</td>
<td>[0]</td>
</tr>
<tr>
<td></td>
<td>'C'</td>
<td>'V'</td>
</tr>
</tbody>
</table>
Tracing Client Code

```c
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ) )
    charStack.Pop();
charStack.Push('K');
while ( !charStack.IsEmpty( ) )
    { letter = charStack.Pop(0); 
    charStack.Pop(0); }
```
### Tracing Client Code

**Private data:**

- **letter**
  - `letter = 'K';`

- **StackType charStack**
  - `StackType charStack;`
  - `charStack.Push(letter);`
  - `charStack.Push('C');`
  - `charStack.Push('S');`
  - `if (!charStack.IsEmpty())`
    - `charStack.Pop();`
  - `charStack.Push('K');`
  - `while (!charStack.IsEmpty())`
    - `{ letter = charStack.Top();`
    - `charStack.Pop(0);`

---

### Example Code

```
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty())
    charStack.Pop();
charStack.Push('K');
while (!charStack.IsEmpty())
    { letter = charStack.Top();
    charStack.Pop(0);}
```
Tracing Client Code

```
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty())
    charStack.Pop();
charStack.Push('K');
while (!charStack.IsEmpty())
{ letter = charStack.Top();
  charStack.Pop();
}
```

Private data:

<table>
<thead>
<tr>
<th>items</th>
<th>[0]</th>
<th>'V'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[1]</td>
<td>'C'</td>
</tr>
<tr>
<td></td>
<td>[2]</td>
<td>'K'</td>
</tr>
<tr>
<td>top</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[MAX_ITEMS-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

22
Tracing Client Code

char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty() )
    charStack.Pop();
charStack.Push('K');
while (!charStack.IsEmpty())
    {
        letter = charStack.Top();
        charStack.Pop(0);
    }
Private data:

<table>
<thead>
<tr>
<th>items</th>
<th>top</th>
<th>[MAX_ITEMS-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[1]</td>
<td>‘C’</td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>‘K’</td>
<td></td>
</tr>
</tbody>
</table>

char letter = ‘V’;
StackType charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if (!charStack.IsEmpty())
    charStack.Pop();
charStack.Push(‘K’);
while (!charStack.IsEmpty())
    { letter = charStack.Top();
    charStack.Pop(0); }

24
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ) )
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
    { letter = charStack.Top();
      charStack.Pop(0); }
Tracing Client Code

Private data:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>[MAX_ITEMS-1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ 2 ]</td>
<td>‘K’</td>
<td></td>
</tr>
<tr>
<td>[ 1 ]</td>
<td>‘C’</td>
<td></td>
</tr>
<tr>
<td>[ 0 ]</td>
<td>‘V’</td>
<td></td>
</tr>
</tbody>
</table>
```

code:

```c
char letter = ‘V’;
StackType charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ) )
    charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ) )
    { letter = charStack.Top();
    charStack.Pop(0); }
```
char letter = 'V';
StackType charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty() )
    charStack.Pop();
charStack.Push('K');
while (!charStack.IsEmpty())
    { letter = charStack.Pop();
      charStack.Pop(0) }
What is a Class Template?

• A class template allows the compiler to generate multiple versions of a class type by using type parameters.

• The formal parameter appears in the class template definition, and the actual parameter appears in the client code. Both are enclosed in pointed brackets, `< >`. 
StackType<int> numStack;

<table>
<thead>
<tr>
<th>top</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[MAX_ITEMS-1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>items</td>
<td>0</td>
</tr>
</tbody>
</table>
StackType<float> numStack;

top

[3]

[2]

[1]

items

[0]

3

3456.8

-90.98

98.6

167.87
StackType<StrType> numStack;

top

[3]
Bradley

[2]
Asad

[1]
Rodrigo

[0]
Max

items


[1] Max

[1] Max

[0] Max

[0] Max

[0] Max
// CLASS TEMPLATE DEFINITION

#include "ItemType.h" // for MAX_ITEMS and ItemTyp

template<class ItemType> // formal parameter list
class StackType
{
    public:
        StackType( );
        bool IsEmpty( ) const;
        bool IsFull( ) const;
        void Push( ItemType item );
        void Pop( ItemType& item );
        ItemType Top( );

    private:
        int top;
        ItemType items[MAX_ITEMS];
};
//--- Sample Class Member Functions ---//

// formal parameter list

template<class ItemType>
StackType<ItemType>::StackType()
{
    top = -1;
}

template<class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
    if (IsFull())
        throw FullStack();
    top++;
    items[top] = newItem; // STATIC ARRAY IMPLEMENTATION
}
Using class templates

• The actual parameter to the template is a data type. Any type can be used, either built-in or user-defined.

• When creating class template
  • Put .h and .cpp in same file or
  • Have .h include .cpp file
char msg [ 8 ];

msg is the base address of the array. We say msg is a pointer because its value is an address. It is a pointer constant because the value of msg itself cannot be changed by assignment. It “points” to the memory location of a char.

6000

<table>
<thead>
<tr>
<th>‘H’</th>
<th>‘e’</th>
<th>‘l’</th>
<th>‘l’</th>
<th>‘o’</th>
<th>‘\0’</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg [0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
</tr>
</tbody>
</table>
Addresses in Memory

- When a variable is declared, enough memory to hold a value of that type is allocated for it at an unused memory location. This is the address of the variable. For example:

  ```
  int x;
  float number;
  char ch;
  ```

  

<table>
<thead>
<tr>
<th>2000</th>
<th>2002</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>number</td>
<td>ch</td>
</tr>
</tbody>
</table>
• The address of a non-array variable can be obtained by using the **address-of operator &.**

```cpp
using namespace std;
int     x;
float   number;
char    ch;

cout << "Address of x is " << &x << endl;
cout << "Address of number is " << &number << endl;
cout << "Address of ch is " << &ch << endl;
```
What is a pointer variable?

• A pointer variable is a variable whose value is the address of a location in memory.

• To declare a pointer variable, you must specify the type of value that the pointer will point to. For example,

```c
int* ptr; // ptr will hold the address of an int
char* q;  // q will hold the address of a char
```
Using a pointer variable

```c
int x;
x = 12;

int* ptr;
ptr = &x;
```

NOTE: Because `ptr` holds the address of `x`, we say that `ptr` “points to” `x`
Unary operator * is the deference (indirection) operator

```cpp
int x;
x = 12;

int* ptr;
ptr = &x;

std::cout << *ptr;
```

NOTE: The value pointed to by *ptr is denoted by *ptr
Using the dereference operator

```c
int x;
x = 12;
int* ptr;
ptr = &x;
*ptr = 5;  // changes the value
           // at adddress ptr to 5
```
Another Example

```c
char ch;
ch = 'A';

char* q;
q = &ch;

*q = 'Z';
char* p;
p = q;  // the right side has value 4000
       // now p and q both point to ch
```
C++ Data Types

Simple
- Integral
  - char
  - short
  - int
  - long
  - enum
- Floating
  - float
  - double
  - long double

Structured
- array
- struct
- union
- class

Address
- pointer
- reference
The **NULL** Pointer

There is a pointer constant 0 called the “null pointer” denoted by NULL in cstddef. But NULL is not memory address 0.

**NOTE:** It is an error to dereference a pointer whose value is NULL. Such an error may cause your program to crash, or behave erratically. It is the programmer’s job to check for this.

```
while (ptr != NULL)
{
    . . .
    // ok to use *ptr here
}
```
### Allocation of memory

<table>
<thead>
<tr>
<th><strong>STATIC ALLOCATION</strong></th>
<th><strong>DYNAMIC ALLOCATION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Static allocation is the allocation of memory space at <strong>compile time</strong>.</td>
<td>Dynamic allocation is the allocation of memory space at <strong>run time</strong> by using operator <strong>new</strong>.</td>
</tr>
</tbody>
</table>
3 Kinds of Program Data

- **STATIC DATA**: memory allocation exists throughout execution of program.
  
  ```
  static long SeedValue;
  ```

- **AUTOMATIC DATA**: automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function.

- **DYNAMIC DATA**: explicitly allocated and deallocated during program execution by C++ instructions written by programmer using unary operators `new` and `delete`
Using operator `new`

If memory is available in an area called the free store (or heap), operator `new` **allocates the requested object or array, and returns a pointer to (address of ) the memory allocated.**

Otherwise, the null pointer 0 is returned.

The dynamically allocated object exists until the delete operator destroys it.
Dynamically Allocated Data

```cpp
char* ptr;
ptr = new char;
*ptr = 'B';
std::cout << *ptr;
```
Dynamically Allocated Data

char* ptr;

ptr = new char;

*ptr = ‘B’;

std::cout << *ptr;

NOTE: Dynamic data has no variable name
Dynamically Allocated Data

```cpp
char* ptr;
ptr = new char;
*ptr = 'B';
std::cout << *ptr;
```

NOTE: Dynamic data has no variable name
char* ptr;
ptr = new char;
*ptr = ‘B’;
std::cout << *ptr;
delete ptr;

NOTE: Delete deallocates the memory pointed to by ptr.
Using operator delete

The **object or array currently pointed to by the pointer is deallocated**, and the pointer is considered unassigned. The memory is returned to the free store.

Square brackets are used with delete to deallocate a dynamically allocated array of classes.
# Some C++ pointer operations

## Precedence

<table>
<thead>
<tr>
<th>Higher</th>
<th>Unary:</th>
<th>Lower</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-&gt;</td>
<td>=</td>
</tr>
<tr>
<td></td>
<td>++ -- ! * new delete</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Increment, Decrement, NOT, Dereference, Allocate, Deallocate</td>
<td>Tests for equality, inequality</td>
</tr>
<tr>
<td></td>
<td>+ - Add Subtract</td>
<td>Assignment</td>
</tr>
<tr>
<td></td>
<td>&lt; &lt;= &gt; &gt;= Relational operators</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Array Allocation

```c
char *ptr; // ptr is a pointer variable that
        // can hold the address of a char

ptr = new char[5]; // dynamically, during run time, allocates
                   // memory for 5 characters and places into
                   // the contents of ptr their beginning address
```

![Diagram showing memory allocation and pointer access]

ptr

6000
Dynamic Array Allocation

```cpp
char *ptr;
ptr = new char[5];
strcpy(ptr, "Bye");
ptr[1] = 'u';  // a pointer can be subscripted
std::cout << ptr[2];
```

![Diagram showing dynamic array allocation]
Dynamic Array Deallocation

char *ptr;
ptr = new char[5];
strcpy(ptr, "Bye");
ptr[1] = 'u';
delete ptr;  // deallocates array pointed to by ptr  
// ptr itself is not deallocated, but  
// the value of ptr is considered unassigned
What happens here?

```c
int* ptr = new int;
*ptr = 3;

ptr = new int;  // changes value of ptr
*ptr = 4;
```
A memory leak occurs when dynamic memory (that was created using operator `new`) has been left without a pointer to it by the programmer, and so is inaccessible.

```cpp
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```

How else can an object become inaccessible?
Causing a Memory Leak

```cpp
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;

ptr = ptr2;  // here the 8 becomes inaccessible
```
A Dangling Pointer

- occurs when two pointers point to the same object and delete is applied to one of them.

```cpp
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```

FOR EXAMPLE,
Leaving a Dangling Pointer

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
delete ptr2; // ptr is left dangling
ptr2 = NULL;
```
class StackType

Private Data:
- top: 2
- maxStack: 5
- items: [50, 43, 80, ...]

~StackType
Push
Pop
// StackType class template

template<class ItemType>
class StackType
{

public:
    StackType(int max );    // max is stack size
    StackType(); // Default size is 500
      // Rest of the prototypes go here.

private:
    int top;
    int maxStack;    // Maximum number of stack items.
    ItemType*  items; // Pointer to dynamically
    // allocated memory

};
// Templated StackType class variables declared

StackType<int> myStack(100);
// Stack of at most 100 integers.

StackType<float> yourStack(50);
// Stack of at most 50 floating point values.

StackType<char> aStack;
// Stack of at most 500 characters.
// Implementation of member function templates

template<class ItemType>
StackType<ItemType>::StackType(int max)
{
    maxStack = max;
    top = -1;
    items = new ItemType[maxStack];
}

template<class ItemType>
StackType<ItemType>::StackType()
{
    maxStack = 500;
    top = -1;
    items = new ItemType[max];
}
What is a Queue?

• **Logical (or ADT) level:** A queue is an ordered group of homogeneous items (elements), in which new elements are added at one end (the **rear**), and elements are removed from the other end (the **front**).

• A queue is a **FIFO** “first in, first out” structure.
Queue ADT Operations

- **MakeEmpty** -- Sets queue to an empty state.
- **IsEmpty** -- Determines whether the queue is currently empty.
- **IsFull** -- Determines whether the queue is currently full.
- **Enqueue (ItemType newItem)** -- Adds newItem to the rear of the queue.
- **Dequeue (ItemType& item)** -- Removes the item at the front of the queue and returns it in item.
ADT Queue Operations

Transformers
- MakeEmpty
- Enqueue
- Dequeue

Observers
- IsEmpty
- IsFull

change state

observe state
class QueType

Private Data:
- front: 1
- rear: 4
- maxQue: 5
- items

items [0] [1] [2] [3] [4]

RESERVED 'C' 'X' 'J'

QueType
~QueType
Enqueue
Dequeque
// CLASS TEMPLATE DEFINITION FOR CIRCULAR QUEUE
#include "ItemType.h" // for ItemType

template<class ItemType>
class QueType
{
public:
    QueType( ) ;
    QueType( int max );     // PARAMETERIZED CONSTRUCTOR
    ~QueType( ) ;          // DESTRUCTOR

    ... bool IsFull( ) const;
    void Enqueue( ItemType item );
    void Dequeue( ItemType&  item );

private:
    int      front;
    int      rear;
    int      maxQue;
    ItemType*  items;   // DYNAMIC ARRAY IMPLEMENTATION
};
template<class ItemType>
QueType<ItemType>::QueType( int max ) // PARAMETERIZED
{
    maxQue = max + 1;
    front = maxQue - 1;
    rear = maxQue - 1;
    items = new ItemType[maxQue]; // dynamically allocates
}

template<class ItemType>
bool QueType<ItemType>::IsEmpty( )
{
    return ( rear == front )
}
template<class ItemType>
QueType<ItemType>::~QueType( )
{
    delete [ ] items;        // deallocates array
}

// WRAP AROUND
bool QueType<ItemType>::IsFull( )
{
    // WRAP AROUND
    return ( (rear + 1) % maxQue == front )
}
// DERIVED CLASS CountedQueType FROM BASE CLASS QueType

template<class ItemType>
class CountedQueType : public QueType<ItemType>
{
public:
    CountedQueType( );
    void Enqueue( ItemType newItem );
    void Dequeue( ItemType& item );
    int LengthIs( ) const;
    // Returns number of items on the counted queue.

private:
    int length;
};
class CountedQueType<

C D reserved A B
[0] [1] [2] [3] [4]

front = 2
rear = 1
// Member function definitions for class CountedQue

template<class ItemType>
CountedQueType<ItemType>::CountedQueType( ) : QueType<ItemType>( )
{
    length = 0 ;
}

template<class ItemType>
int CountedQueType<ItemType>::LengthIs( ) const
{
    return length ;
}
template<class ItemType>
void CountedQueType<ItemType>::Enqueue( ItemType newItem )
    // Adds newItem to the rear of the queue.
    // Increments length.
{
    length++;
    QueType<ItemType>::Enqueue( newItem );
}

template<class ItemType>
void CountedQueType<ItemType>::Dequeue(ItemType& item )
    // Removes item from the rear of the queue.
    // Decrements length.
{
    length--;
    QueType<ItemType>::Dequeue( item );
}