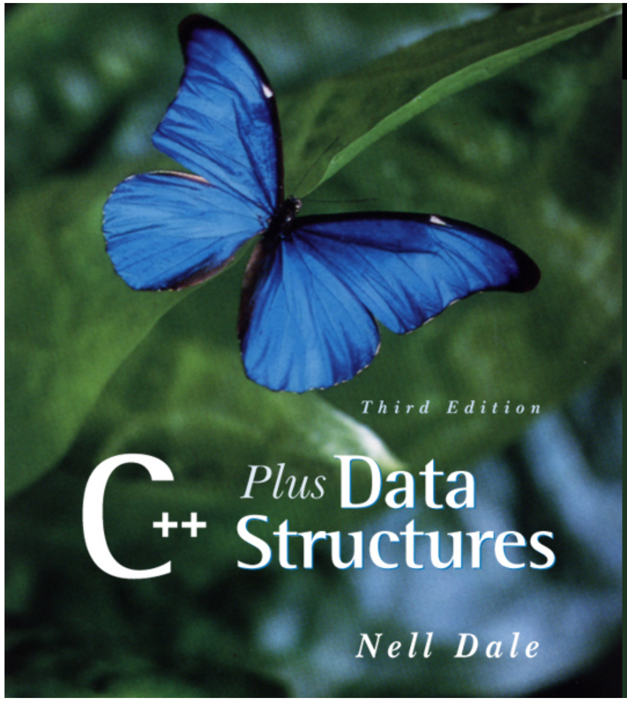
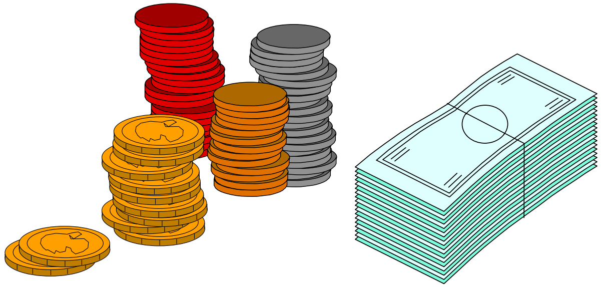


Chapter

4

ADTs Stack and Queue

Stacks of Coins and Bills

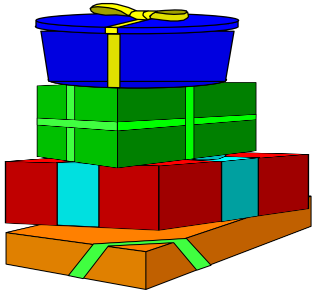


What is a Stack?

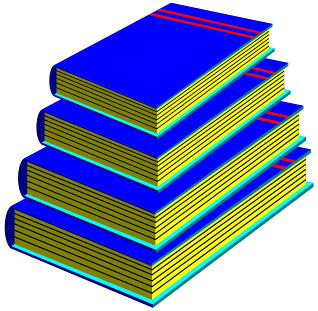
- ***Logical (or ADT) level:*** A stack is an ordered group of homogeneous items (elements), in which the removal and addition of stack items can take place only at the top of the stack.
- A stack is a LIFO "last in, first out" structure.

Stacks of Boxes and Books

TOP OF THE STACK



TOP OF THE STACK



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

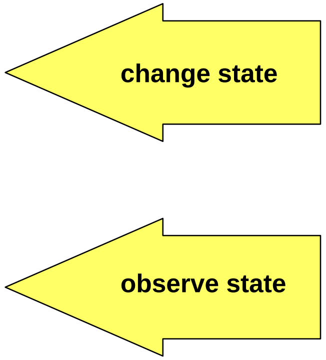
ADT Stack Operations

Transformers

- Push
- Pop

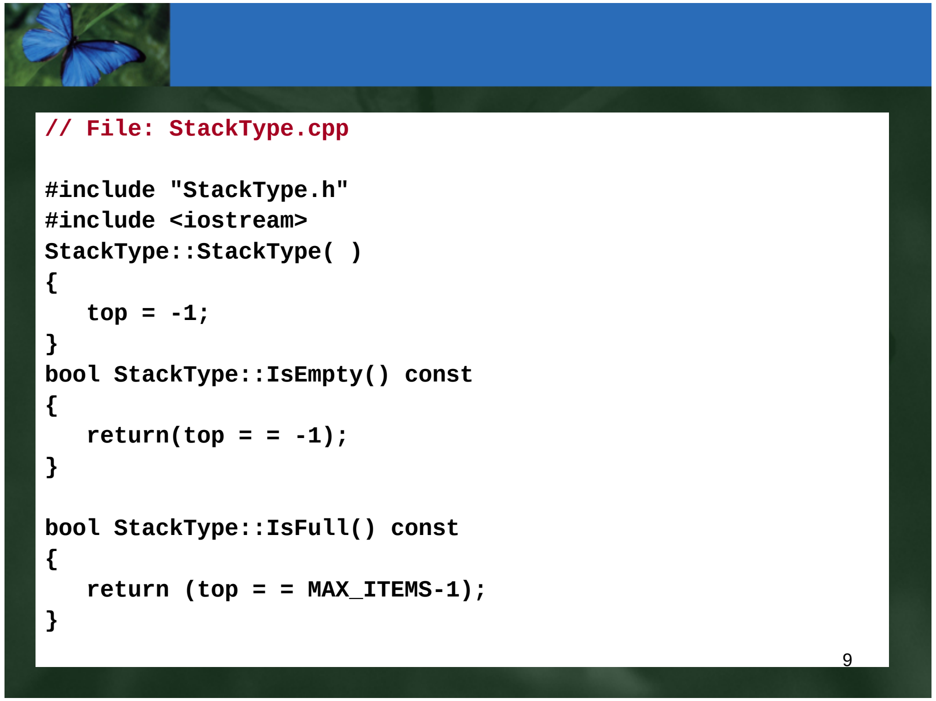
Observers

- IsEmpty
- IsFull
- IsFull

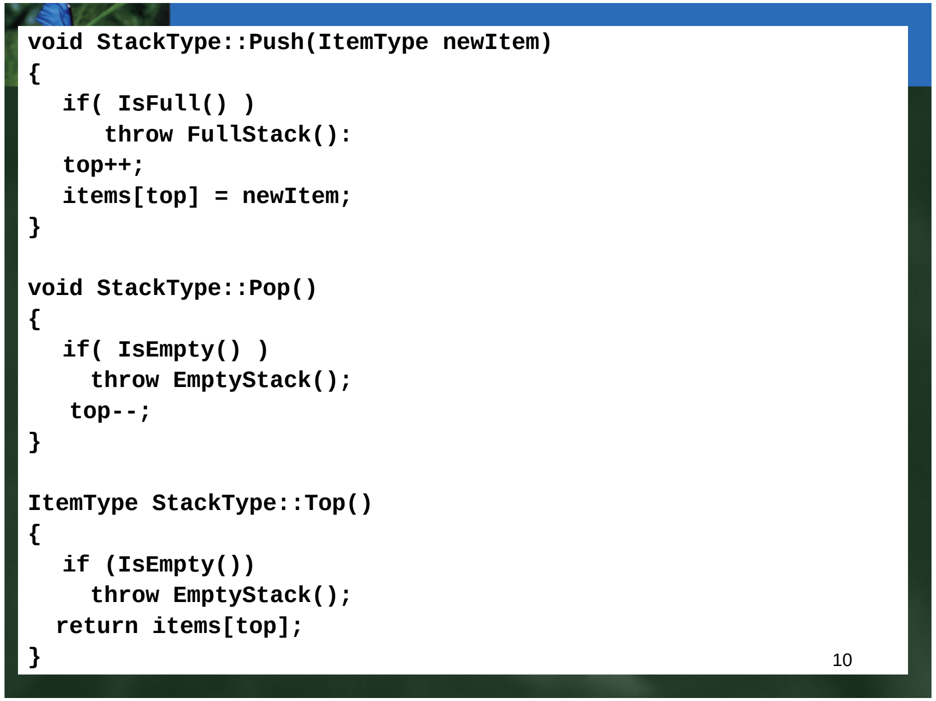


```
// Class specification for Stack ADT in file StackType.h

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)
```

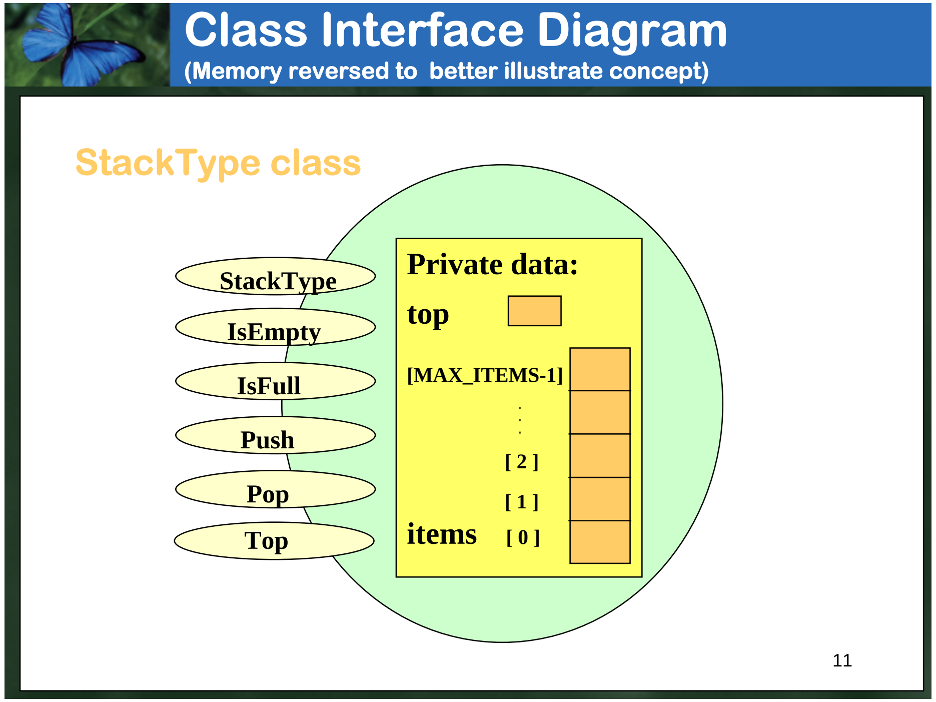
```
   bool IsEmpty() const;
   // Function: Determines whether the stack is empty.
   // Pre:   Stack has been initialized.
   // Post:  Function value = (stack is empty)
   void Push( ItemType item );
   // Function: Adds newItem to the top of the stack.
   // Pre: Stack has been initialized.
   // 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);
}
```

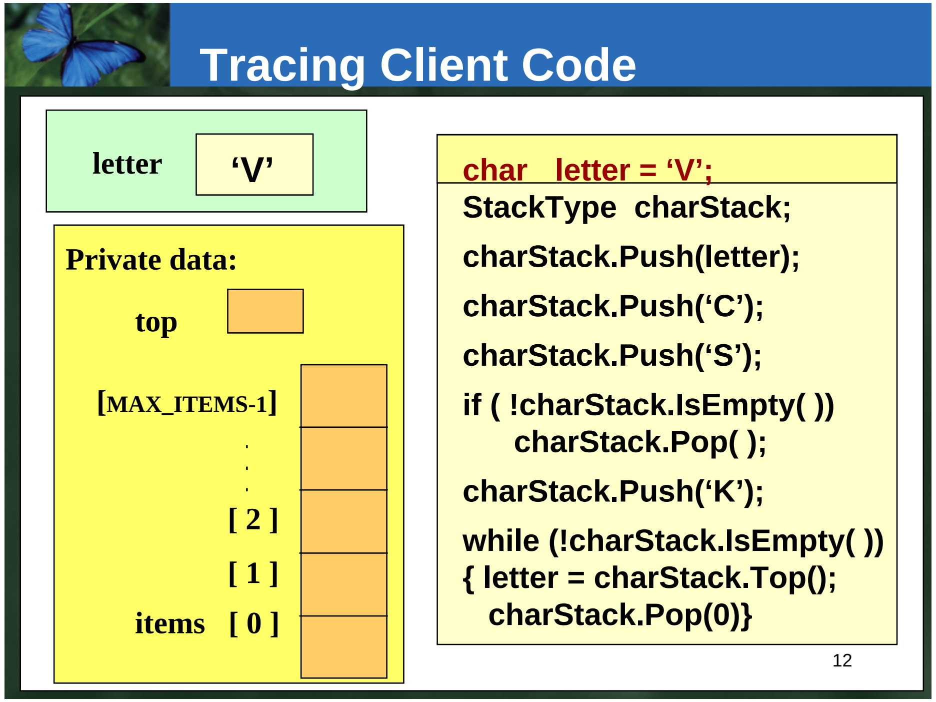


```
void StackType::Push(ItemType newItem)
{
   if( IsFull() )
      throw FullStack():
   top++;
   items[top] = newItem;
}

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

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



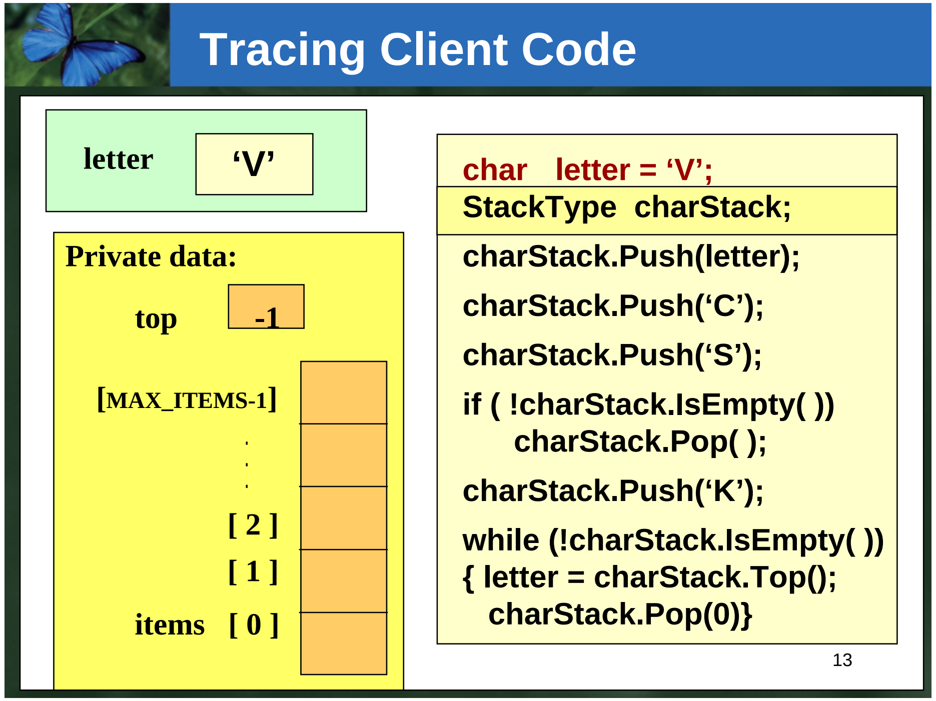
Class Interface Diagram

(Memory reversed to better illustrate concept)



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)}
```



Tracing Client Code

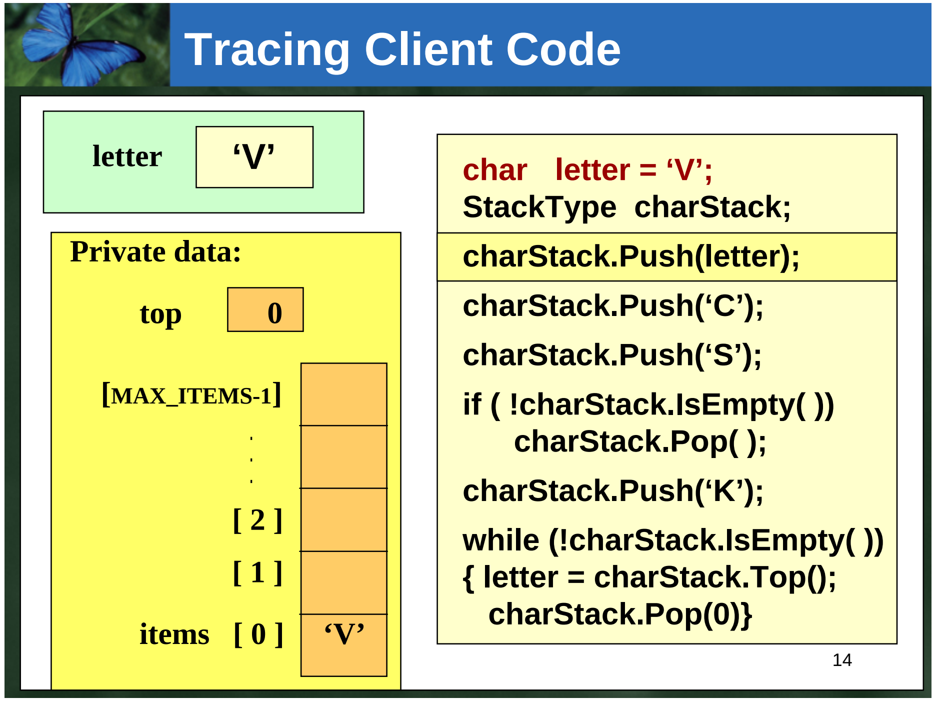
letter ‘V’

Private data:

top -1

[MAX_ITEMS-1]	
. . .	
[2]	
[1]	
items [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.Top();
  charStack.Pop(0)}
```



Tracing Client Code

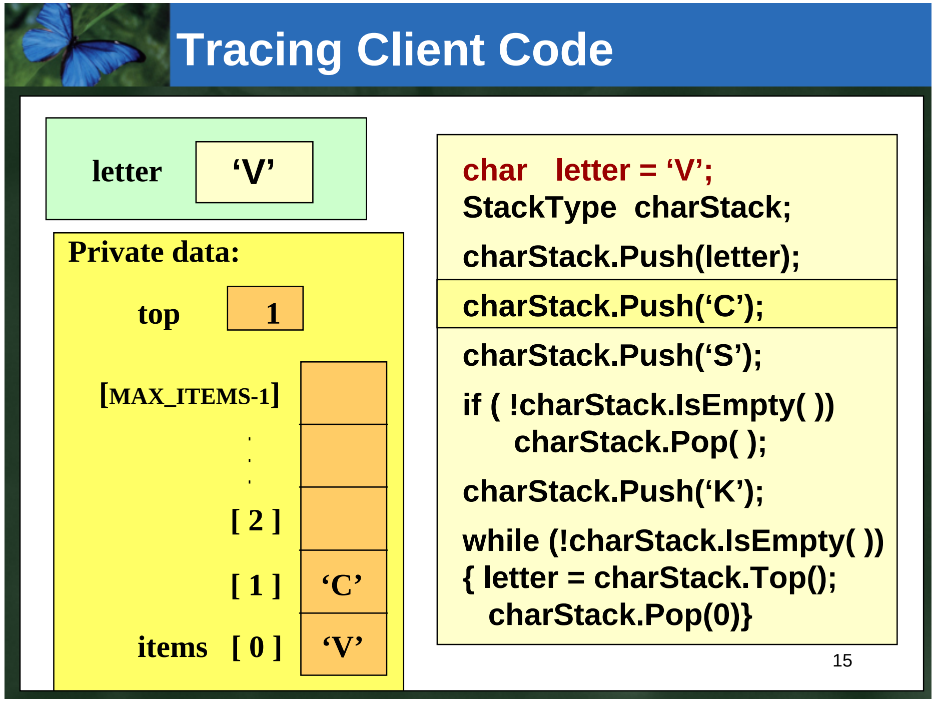
letter ‘V’

Private data:

top 0

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

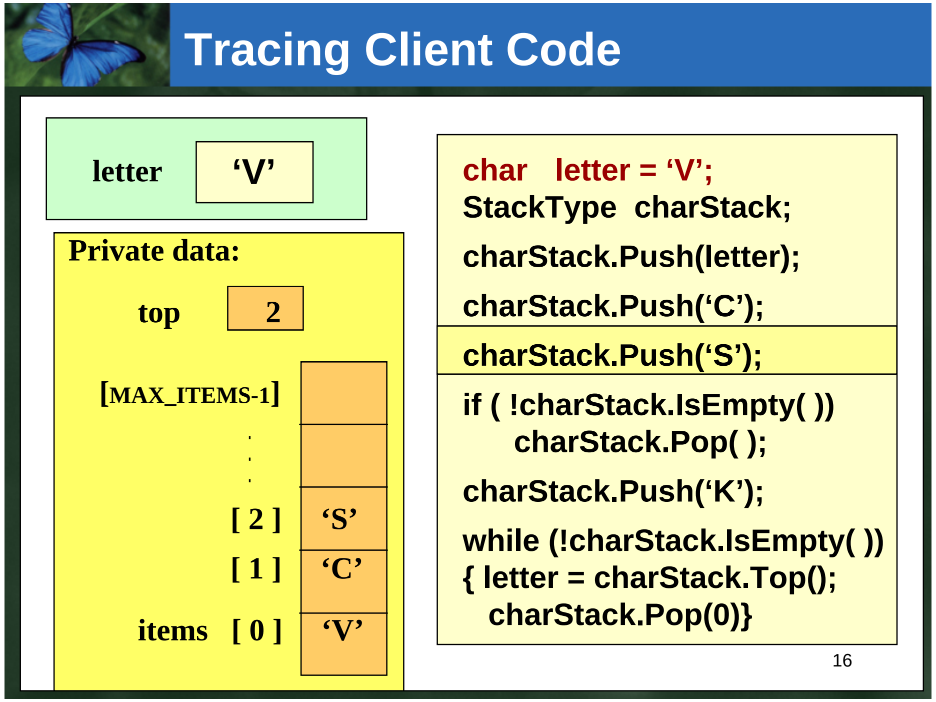
letter ‘V’

Private data:

top 1

[MAX_ITEMS-1]	
. . .	
[2]	
[1]	‘C’
items [0]	‘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

letter ‘V’

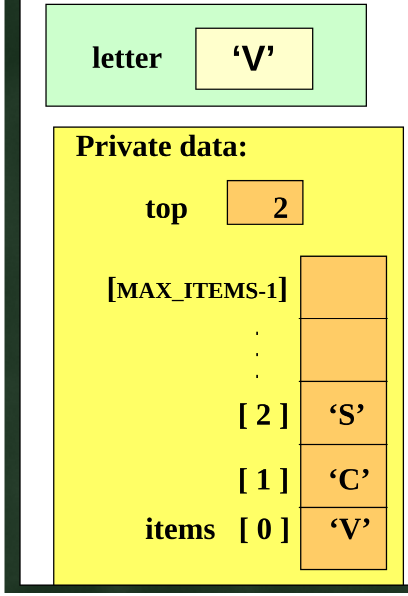
Private data:

top 2

[MAX_ITEMS-1]	
. . .	
[2]	‘S’
[1]	‘C’
items [0]	‘V’

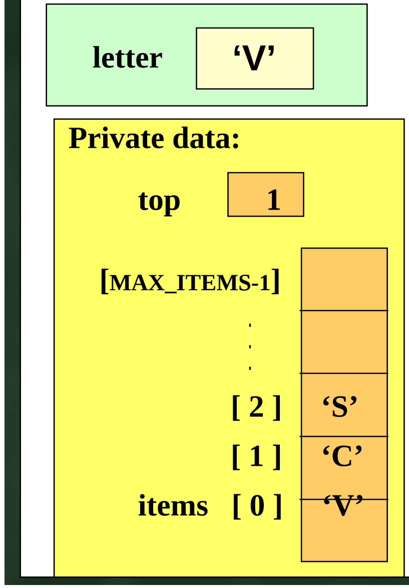
```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
    charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
  charStack.Pop(0)}
```

Tracing Client Code



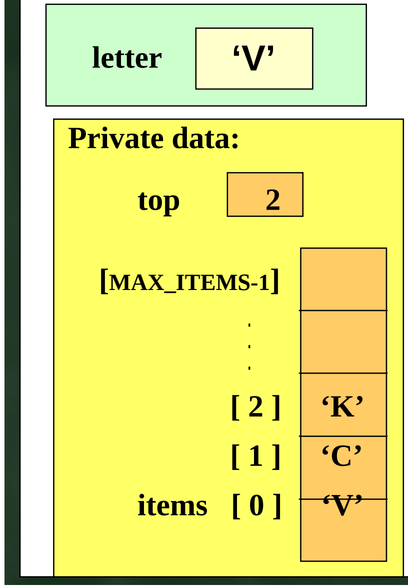
```
char   letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
  charStack.Pop(0)}
```

Tracing Client Code



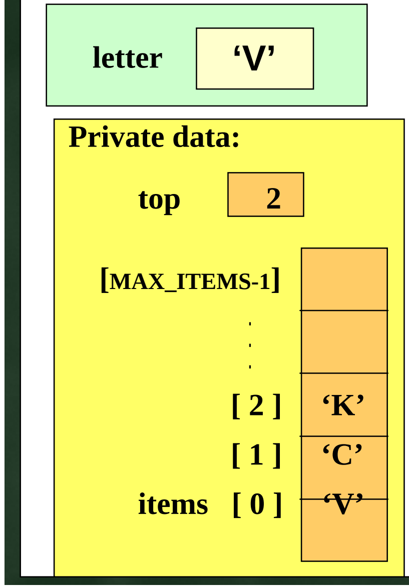
```
char   letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
  charStack.Pop(0)}
```

Tracing Client Code

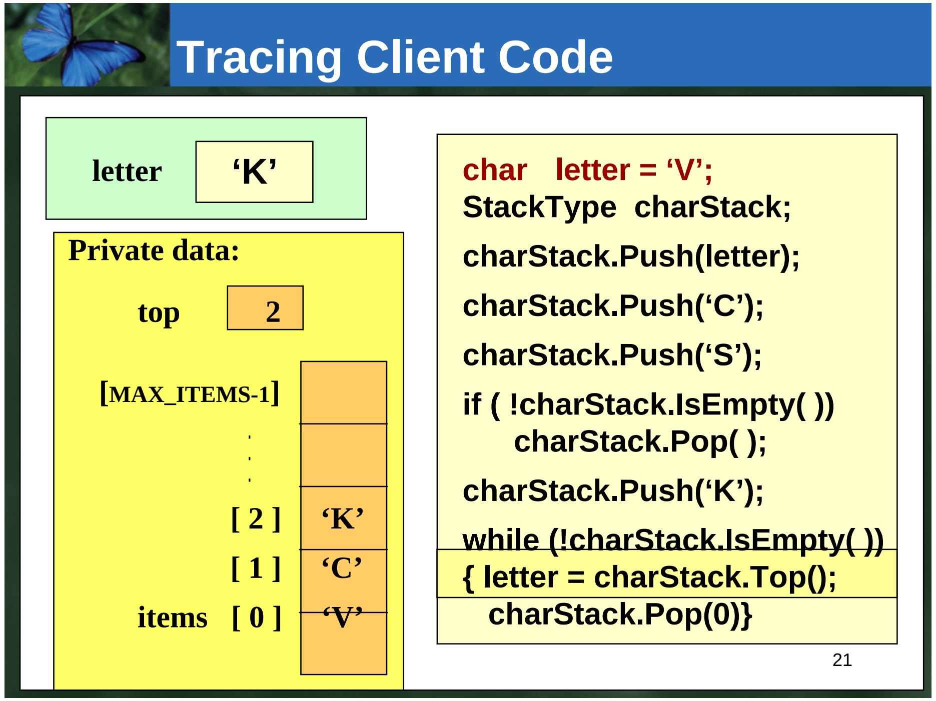


```
char   letter = 'V';
StackType  charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if ( !charStack.IsEmpty( ))
    charStack.Pop( );
charStack.Push('K');
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
  charStack.Pop(0)}
```

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(0)}
```



Tracing Client Code

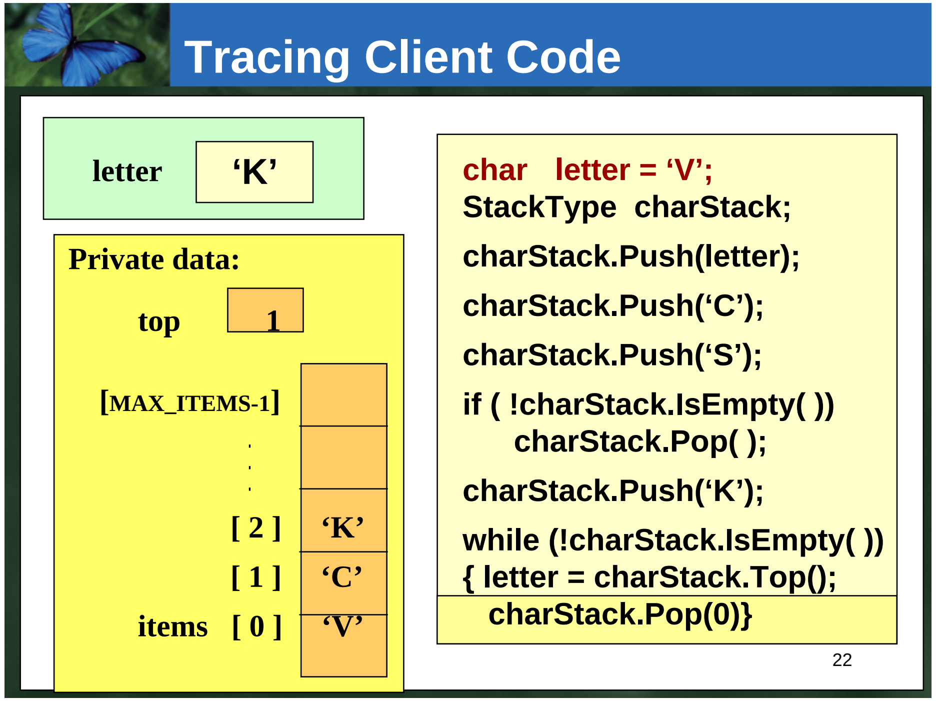
letter ‘K’

Private data:

top 2

[MAX_ITEMS-1]
.
.
.
[2] ‘K’
[1] ‘C’
items [0] ‘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

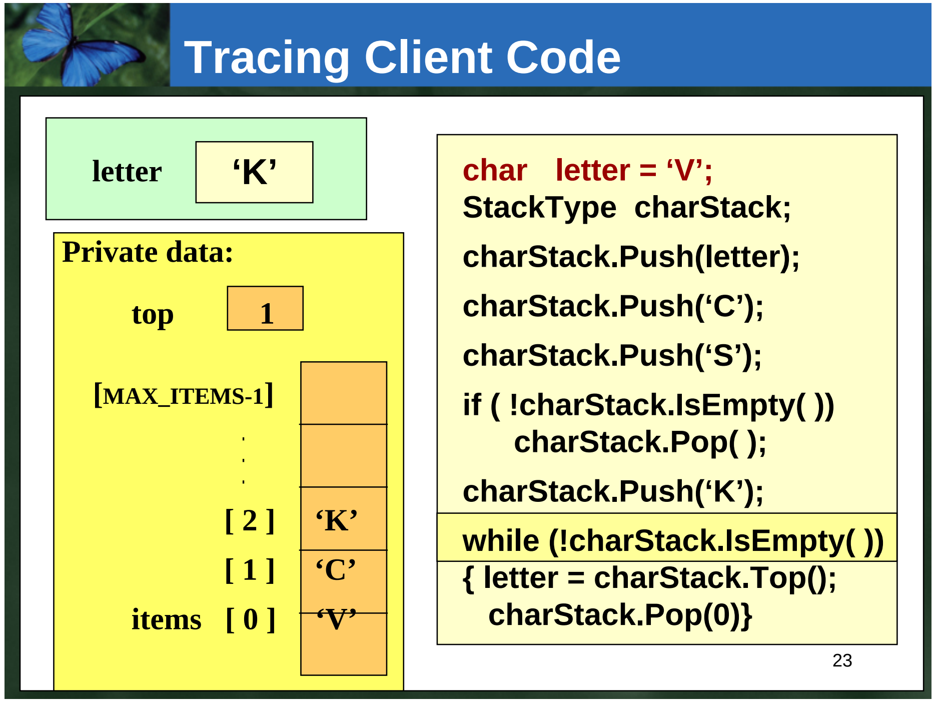
letter ‘K’

Private data:

top 1

[MAX_ITEMS-1]
.
.
.
[2] ‘K’
[1] ‘C’
items [0] ‘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

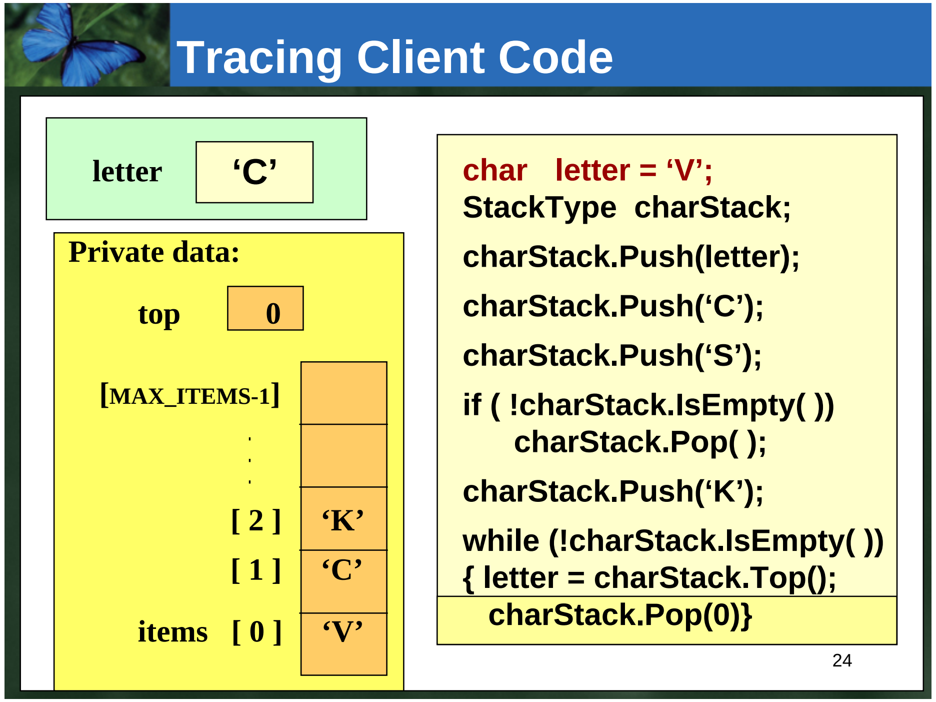
letter ‘K’

Private data:

top 1

[MAX_ITEMS-1]
.
.
.
[2] ‘K’
[1] ‘C’
items [0] ‘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

letter ‘C’

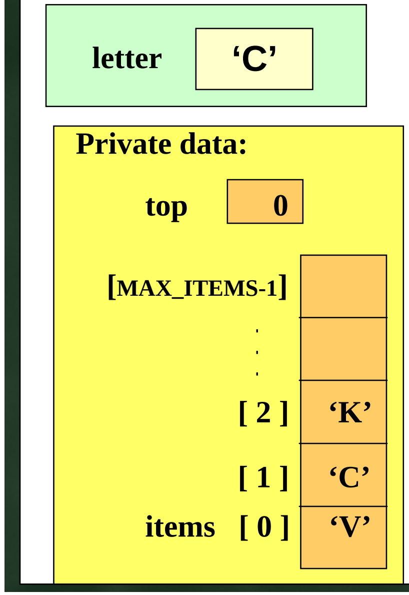
Private data:

top 0

[MAX_ITEMS-1]
.
.
.
[2] ‘K’
[1] ‘C’
items [0] ‘V’

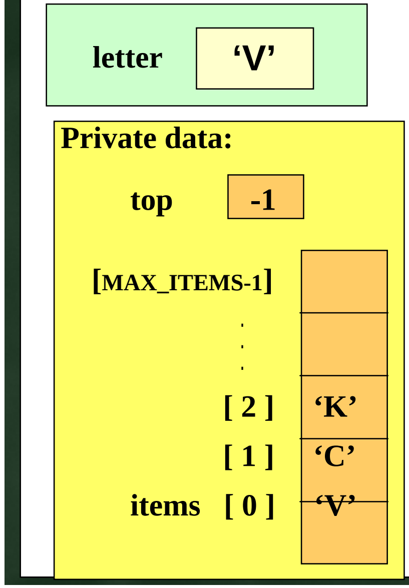
```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
     charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

Tracing Client Code



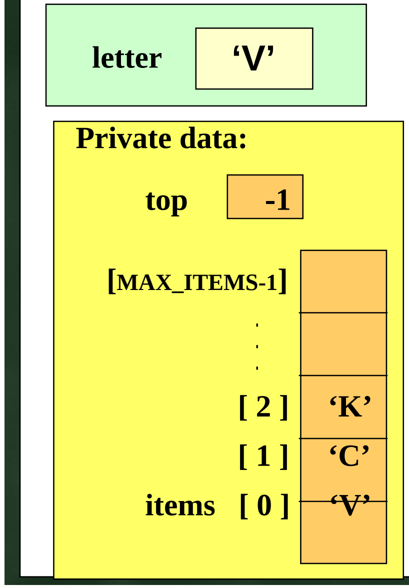
```
char   letter = ‘V’;
StackType  charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if ( !charStack.IsEmpty( ))
     charStack.Pop( );
charStack.Push(‘K’);
while (!charStack.IsEmpty( ))
{ letter = charStack.Top();
   charStack.Pop(0)}
```

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(0)}
```

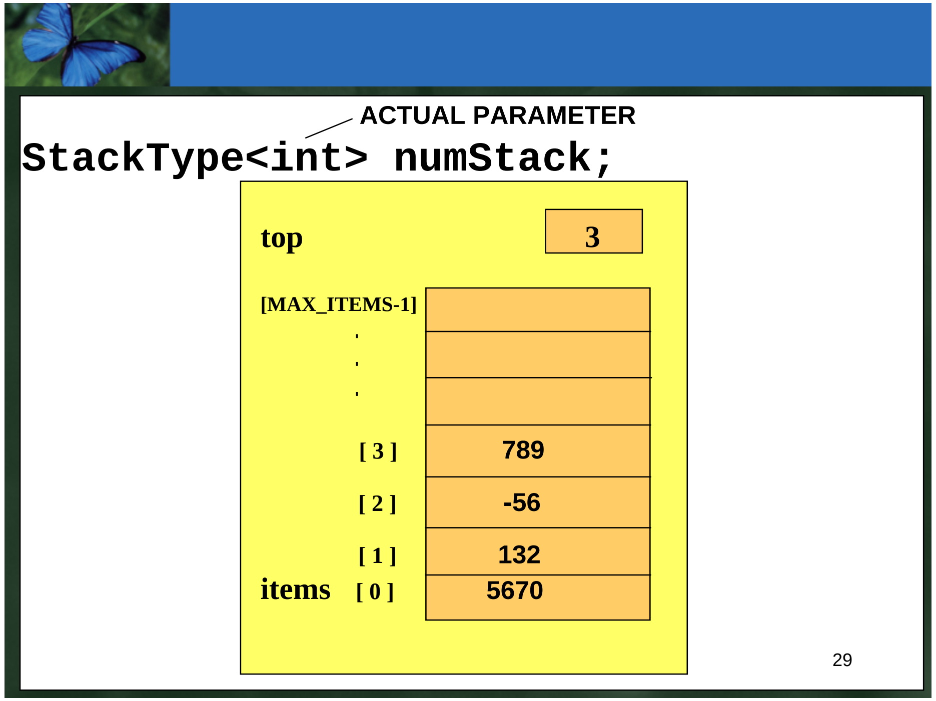
End of Trace



```
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)}
```

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, < >.

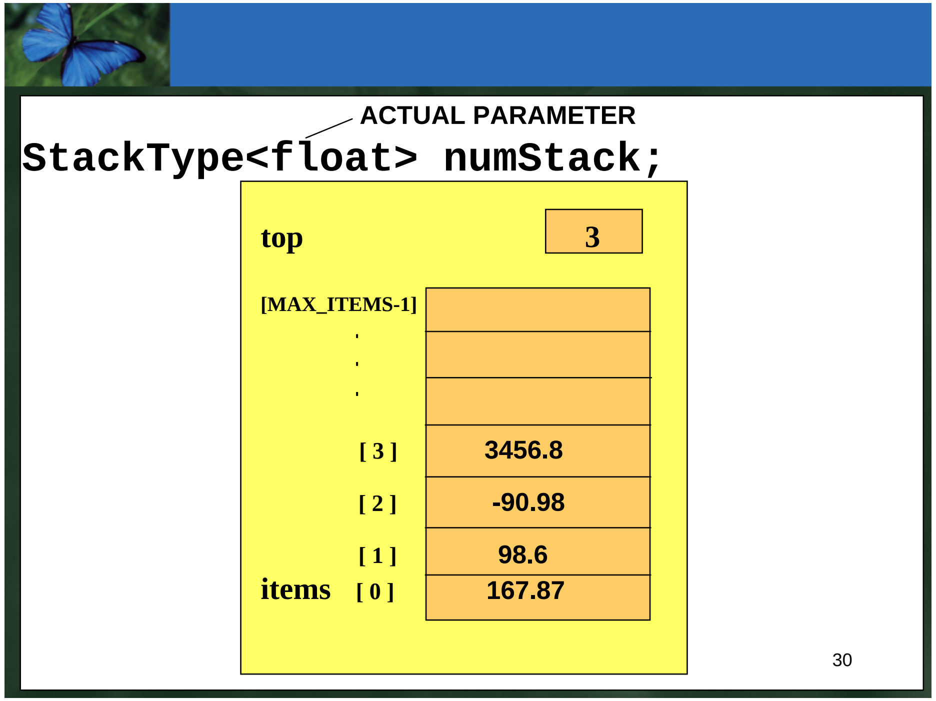


ACTUAL PARAMETER

`StackType<int> numStack;`

top 3

	[MAX_ITEMS-1]	
	.	
	.	
	.	
	[3]	789
	[2]	-56
	[1]	132
items	[0]	5670

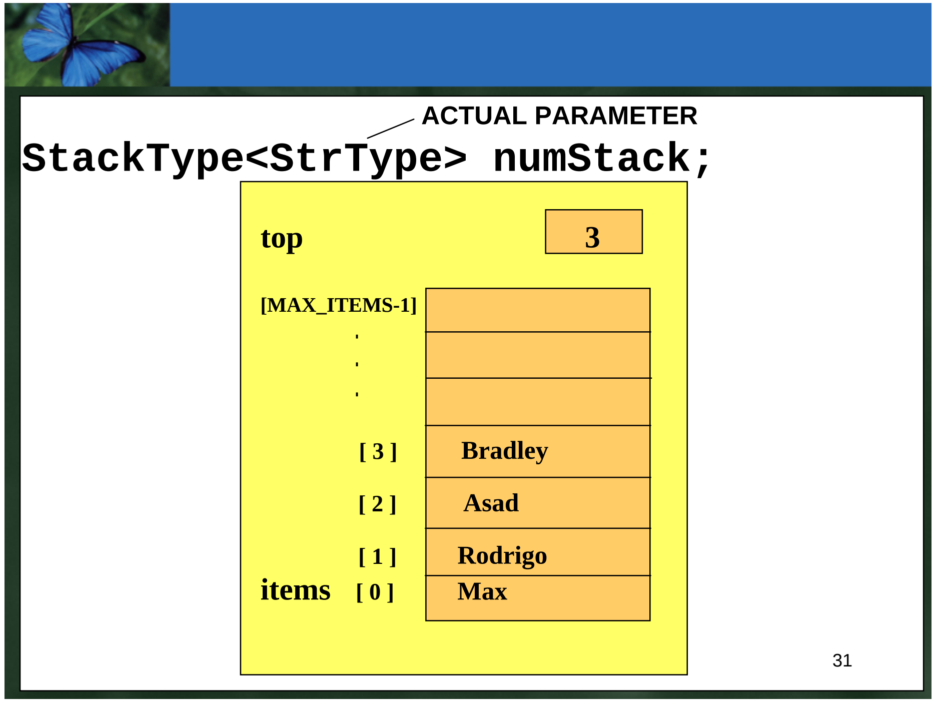


ACTUAL PARAMETER

`StackType<float> numStack;`

top 3

	[MAX_ITEMS-1]	
	.	
	.	
	.	
	[3]	3456.8
	[2]	-90.98
	[1]	98.6
items	[0]	167.87

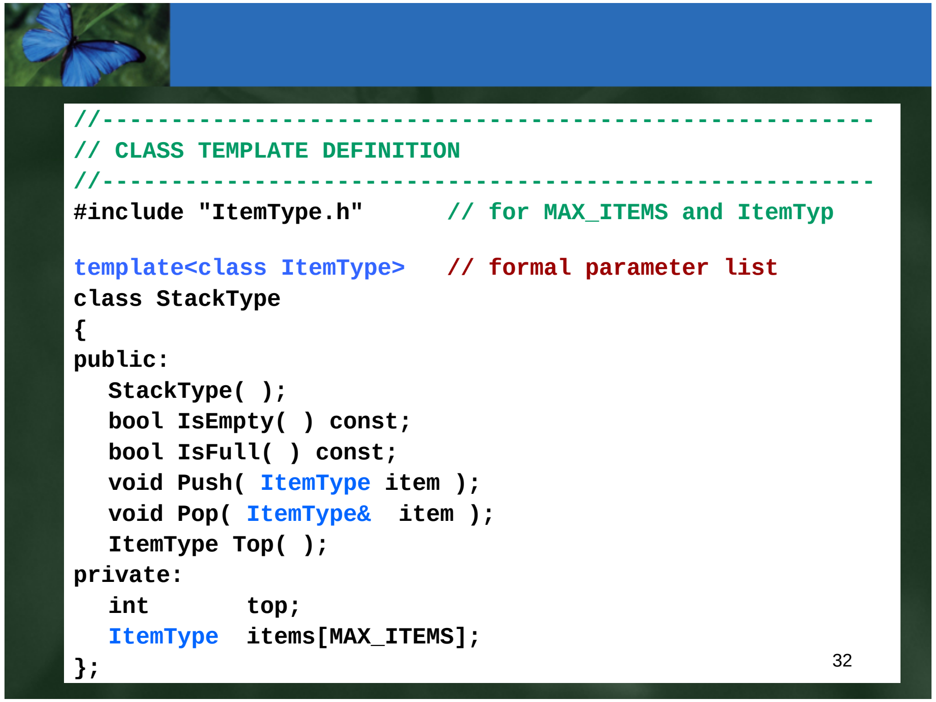


ACTUAL PARAMETER

`StackType<StrType> numStack;`

top 3

	[MAX_ITEMS-1]	
	.	
	.	
	.	
	[3]	Bradley
	[2]	Asad
	[1]	Rodrigo
items	[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
//-------------------------------------------------------

template<class ItemType>   // formal parameter list
StackType<ItemType>::StackType( )
{
   top = -1;
}
template<class ItemType>   // formal parameter list
void StackType<ItemType>::Push ( ItemType newItem )
{
   if (IsFull())
      throw FullStack();
   top++;
   items[top] = newItem;    // STATIC ARRAY IMPLEMENTATION
}
```

Notice that the class name is StackType<ItemType>

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

Recall that . . .

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.

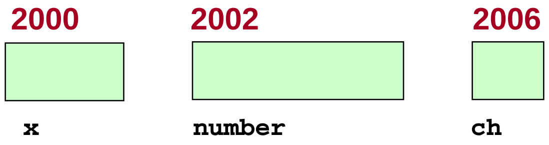
6000

‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’		
msg [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

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;
```



Obtaining Memory Addresses

- The address of a non-array variable can be obtained by using the address-of operator &.

```
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;
```

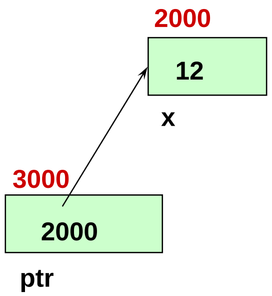
What is a pointer variable?

- 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,

```
int*    ptr; // ptr will hold the address of an int

char*   q;   // q will hold the address of a char
```

Using a pointer variable

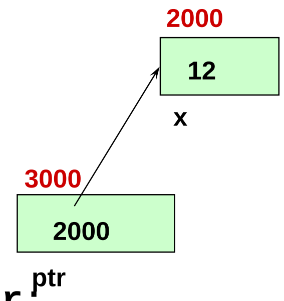
```
int  x;
x = 12;


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

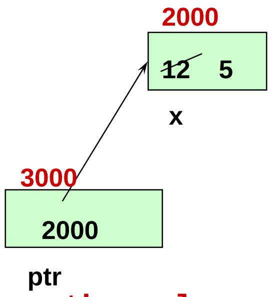
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

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



NOTE: The value pointed to by `ptr` is denoted by `*ptr`

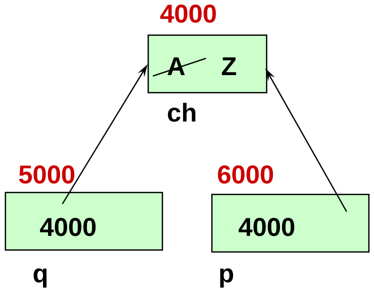
Using the dereference operator

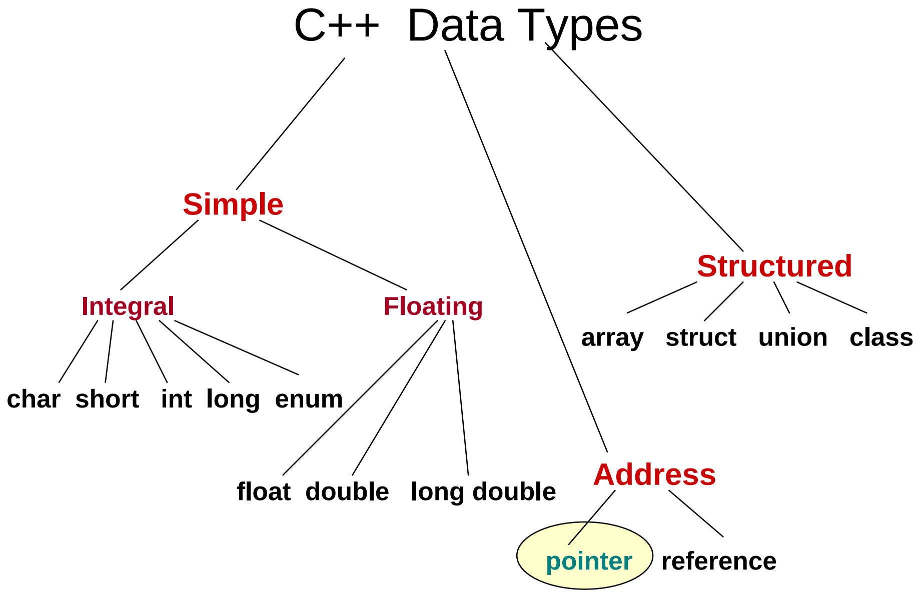
```
int  x;
x = 12;


int*  ptr;
ptr = &x;

*ptr = 5;    // changes the value
             // at adddress ptr to 5
```



Another Example

```
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

Address

pointer reference

Structured

array struct union class

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.

NULL allows a pointer to point nothing

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

STATIC ALLOCATION

Static allocation is the allocation of memory space at compile time.

DYNAMIC ALLOCATION

Dynamic allocation is the allocation of memory space at run time by using operator new.

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

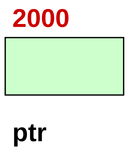
If memory is available in an area called the free store (or heap), operator new allocates the requested object or array, and returns a pointer to (address of) the memory allocated.

Otherwise, the null pointer 0 is returned.

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

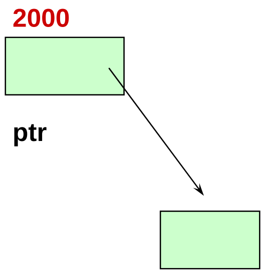
Dynamically Allocated Data

```
char*  ptr;

ptr = new char;
*ptr = ‘B’;
std::cout << *ptr;
```



Dynamically Allocated Data

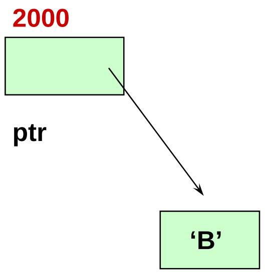
```
char*  ptr;


ptr = new char;
*ptr = ‘B’;
std::cout << *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

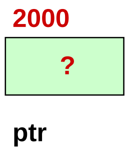
```
char*  ptr;


ptr = new char;
*ptr = ‘B’;
std::cout << *ptr;
```



NOTE: Dynamic data has no variable name

Dynamically Allocated Data

```
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

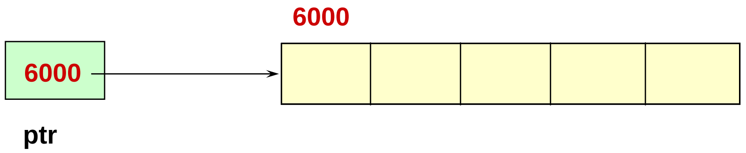
```
MoneyType* moneyPtr = new MoneyType;
moneyPtr->dollars = 3245;
(*moneyPtr).cents = 33;  // NO *moneyPtr.cents = 33;
```

Precedence

Higher	->	**Select member of class pointed to**
Unary:	++ -- ! * new delete	Increment, Decrement, NOT, Dereference, Allocate, Deallocate
	+ -	Add Subtract
	< <= > >=	Relational operators
	== !=	Tests for equality, inequality
Lower	=	Assignment

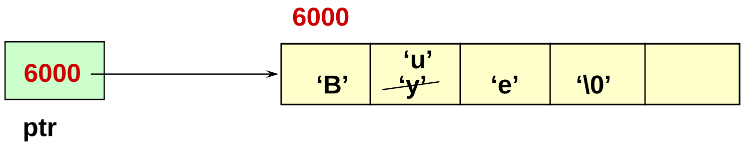
Dynamic Array Allocation

```
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
```



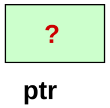
Dynamic Array Allocation

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, “Bye” );

ptr[ 1 ] = ‘u’;          // a pointer can be subscripted

std::cout  << ptr[ 2] ;
```



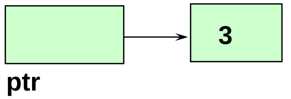
Dynamic Array Deallocation

```
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
```

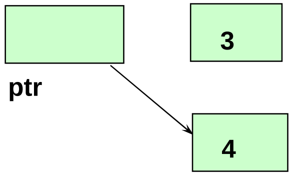


What happens here?

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



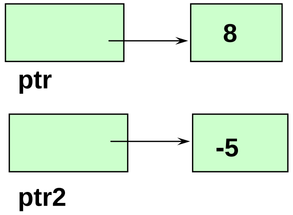
```
ptr = new int;     // changes value of ptr
*ptr = 4;
```



Memory Leak

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.

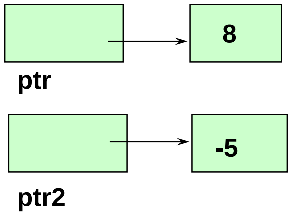
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
```



How else can an object become inaccessible?

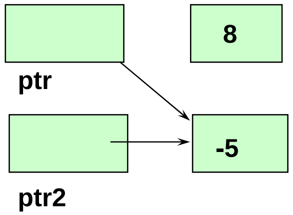
Causing a Memory Leak

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



```
ptr = ptr2;    // here the 8 becomes inaccessible
```

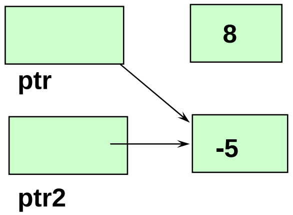
The memory cells that can no longer be accessed are called garbage



A Dangling Pointer

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

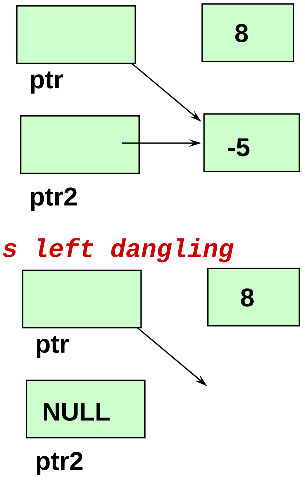
```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;
```



FOR EXAMPLE,

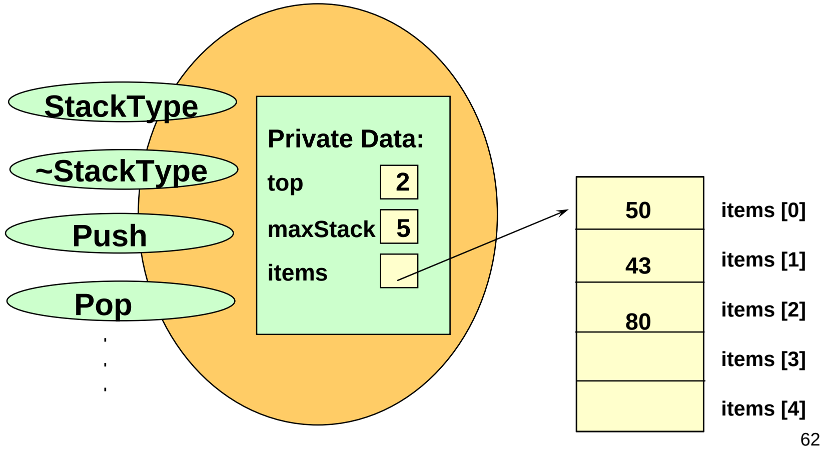
Leaving a Dangling Pointer

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

delete ptr2;    // ptr is left dangling
ptr2 = NULL;
```



DYNAMIC ARRAY IMPLEMENTATION

class StackType



```
// 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.

Example: Queue of Customers

Rear of Queue
Front of Queue
next...
PAY HERE

Enqueue (ItemType newItem)

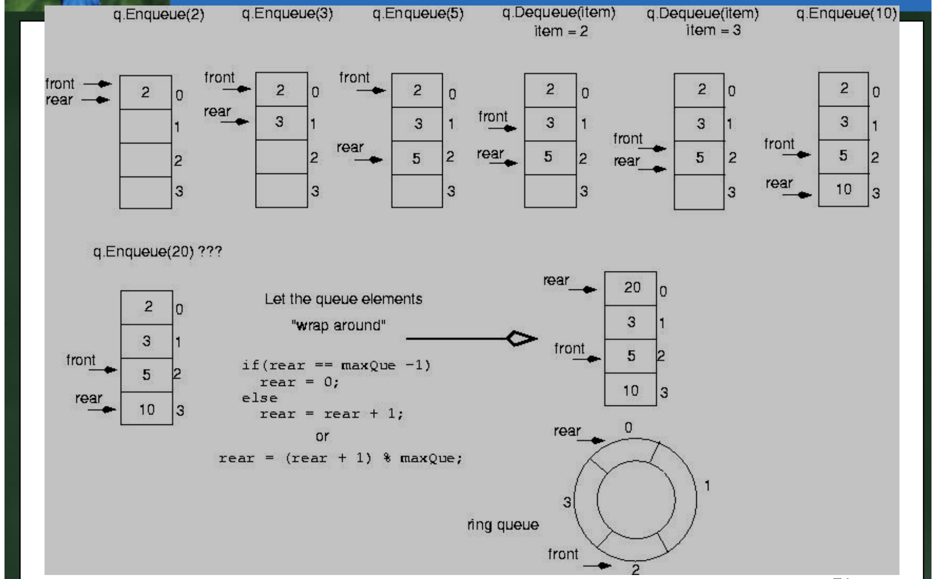
- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

Dequeue (ItemType& item)

- *Function*: Removes front item from queue and returns it in item.
- *Preconditions*: Queue has been initialized and is not empty.
- *Postconditions*: Front element has been removed from queue and item is a copy of removed element.

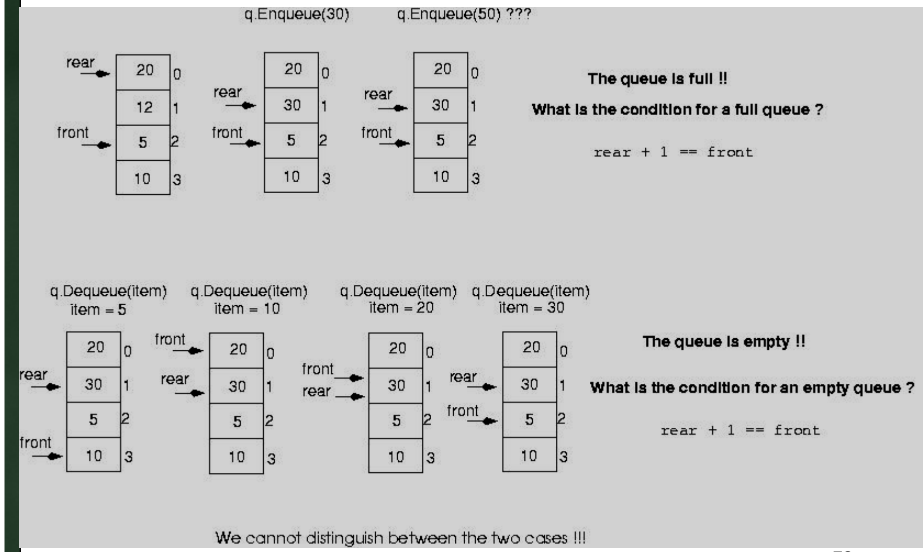
Implementation issues

- Implement the queue as a *circular structure*.
- How do we know if a queue is full or empty?
- Initialization of *front* and *rear*.
- Testing for a *full* or *empty* queue.



Let the queue elements "wrap around"

```
if(rear == maxQue -1)
  rear = 0;
else
  rear = rear + 1;
         or
rear = (rear + 1) % maxQue;
```



The queue is full !!

What is the condition for a full queue ?

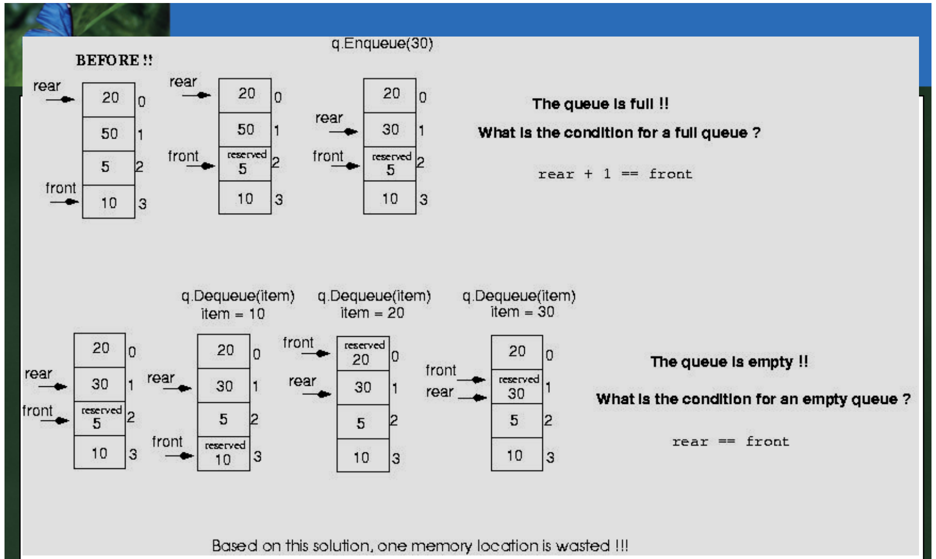
```
rear + 1 == front
```

The queue is empty !!

What is the condition for an empty queue ?

```
rear + 1 == front
```

We cannot distinguish between the two cases !!!



The queue is full !!

What is the condition for a full queue ?

```
rear + 1 == front
```

The queue is empty !!

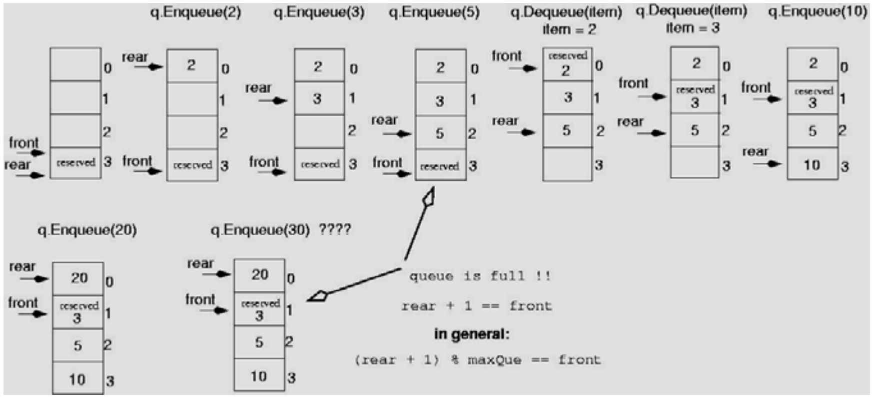
What is the condition for an empty queue ?

```
rear == front
```

Based on this solution, one memory location is wasted !!!

Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

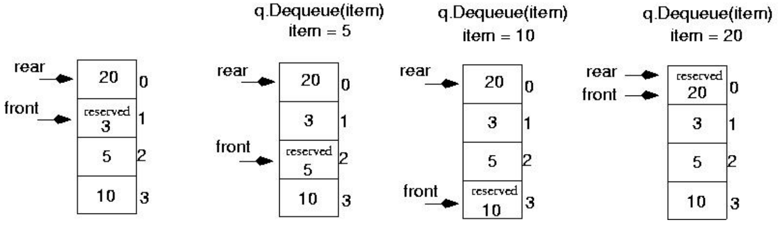
Initialize *front* and *rear*



```
queue is full !!
rear + 1 == front
```

in general:

```
(rear + 1) % maxQue == front
```

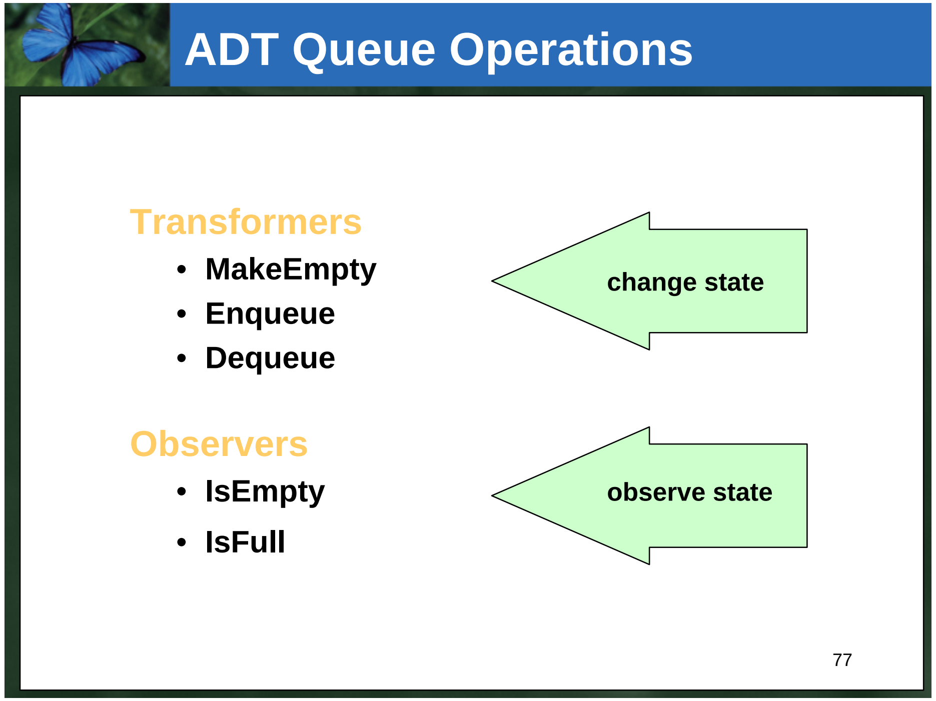


Queue is empty now!!

rear == front

Queue ADT Operations

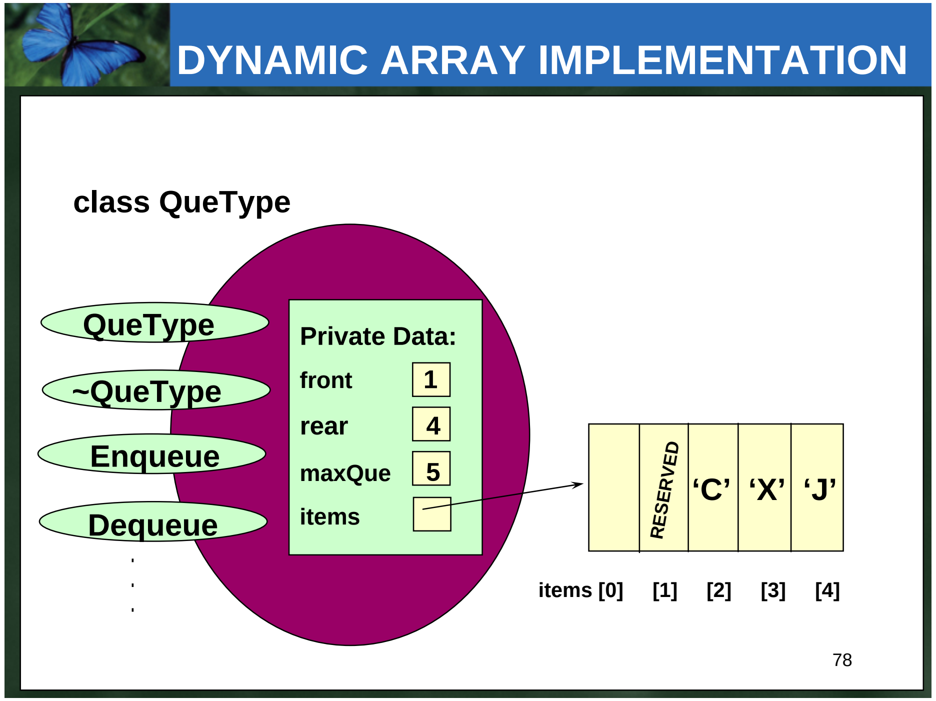
- **MakeEmpty -- Sets queue to an empty state.**
- **IsEmpty -- Determines whether the queue is currently empty.**
- **IsFull -- Determines whether the queue is currently full.**
- **Enqueue (ItemType newItem) -- Adds newItem to the rear of the queue.**
- **Dequeue (ItemType& item) -- Removes the item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



ADT Queue Operations

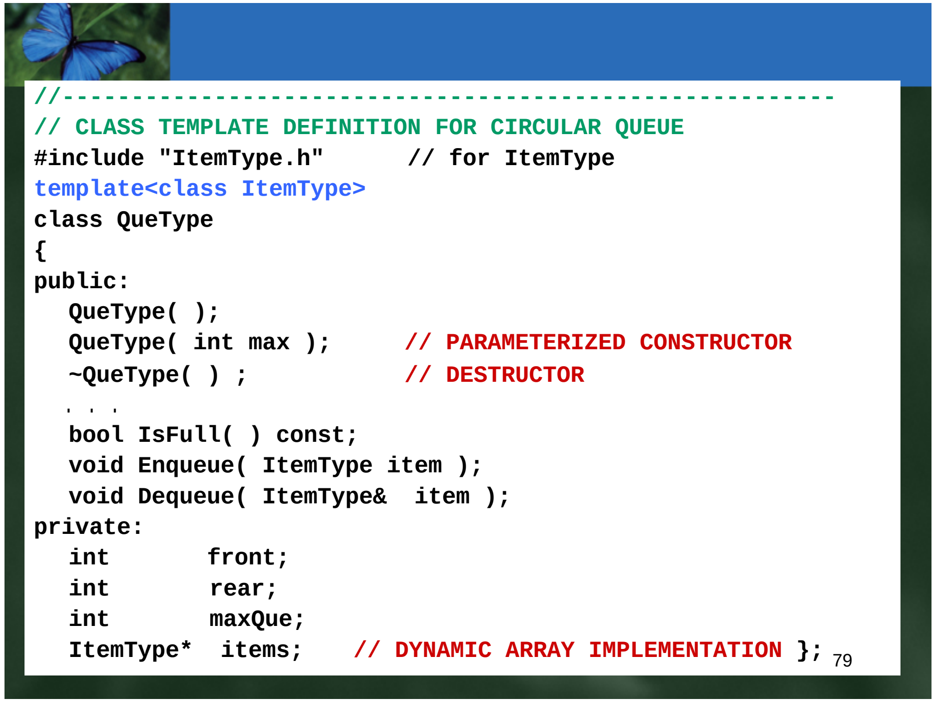
Transformers
- MakeEmpty
- Enqueue
- Dequeue

Observers
- IsEmpty
- IsFull

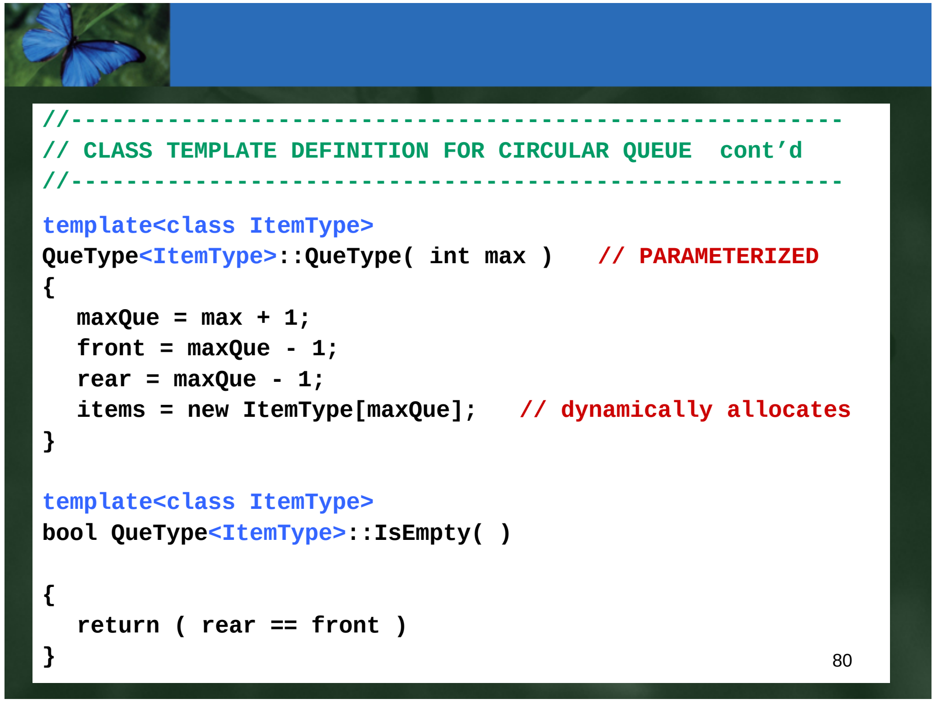


DYNAMIC ARRAY IMPLEMENTATION

class QueType



```
//----------------------------------------------------------
// 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 };
```



```
//----------------------------------------------------------
// CLASS TEMPLATE DEFINITION FOR CIRCULAR QUEUE  cont'd
//----------------------------------------------------------
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 )
}
```

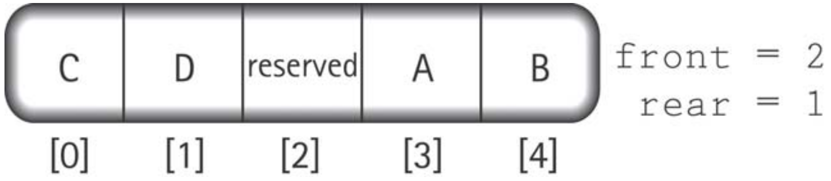
```
//----------------------------------------------------------
// CLASS TEMPLATE DEFINITION FOR CIRCULAR QUEUE  cont’d
//----------------------------------------------------------

template<class ItemType>
QueType<ItemType>::~QueType( )
{
   delete [ ] items;        // deallocates array
}

.
.
.

template<class ItemType>
bool QueType<ItemType>::IsFull( )

{                                         // WRAP AROUND
   return ( (rear + 1) % maxQue == front )
}
```

```
//----------------------------------------------------------
// CLASS TEMPLATE DEFINITION FOR CIRCULAR QUEUE  cont’d
//----------------------------------------------------------

template<class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem )
{
   rear = (rear+1) % maxQue;
   items[rear] = newItem;
}

template<class ItemType>
void QueType<ItemType>::Dequeue(ItemType &item)
{
   front = (front+1) % maxQue;
   item = items[rear];
}
```

SAYS ALL PUBLIC MEMBERS OF QueType CAN BE INVOKED FOR OBJECTS OF TYPE CountedQuType

```
// 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<char>



```
// 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 ;
}
```

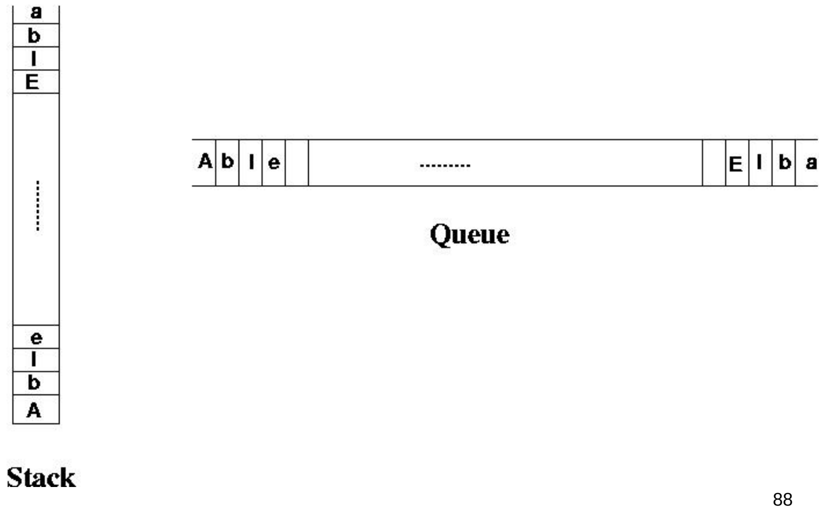
```
template<class ItemType>
void CountedQueType<ItemType>::Enqueue( ItemType newItem )
        // Adds newItem to the rear of the queue.
        // Increments length.
{
   length++;

   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 );
}
```

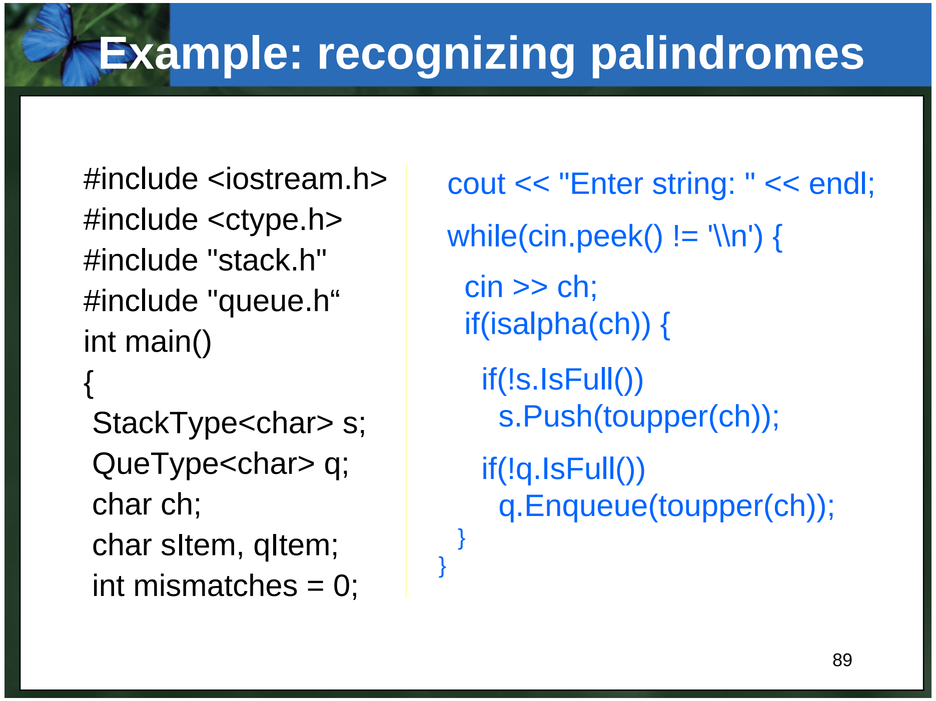
Example: recognizing palindromes

- A *palindrome* is a string that reads the same forward and backward.

  Able was I ere I saw Elba

- We will read the line of text into both a stack and a queue.
- Compare the contents of the stack and the queue character-by-character to see if they would produce the same string of characters.

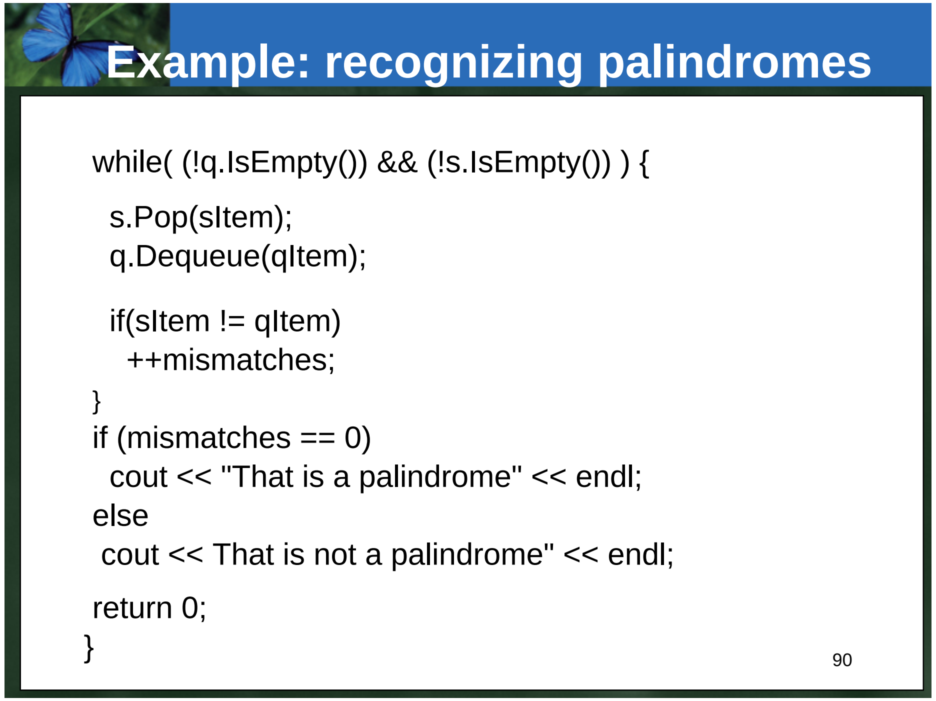




Example: recognizing palindromes

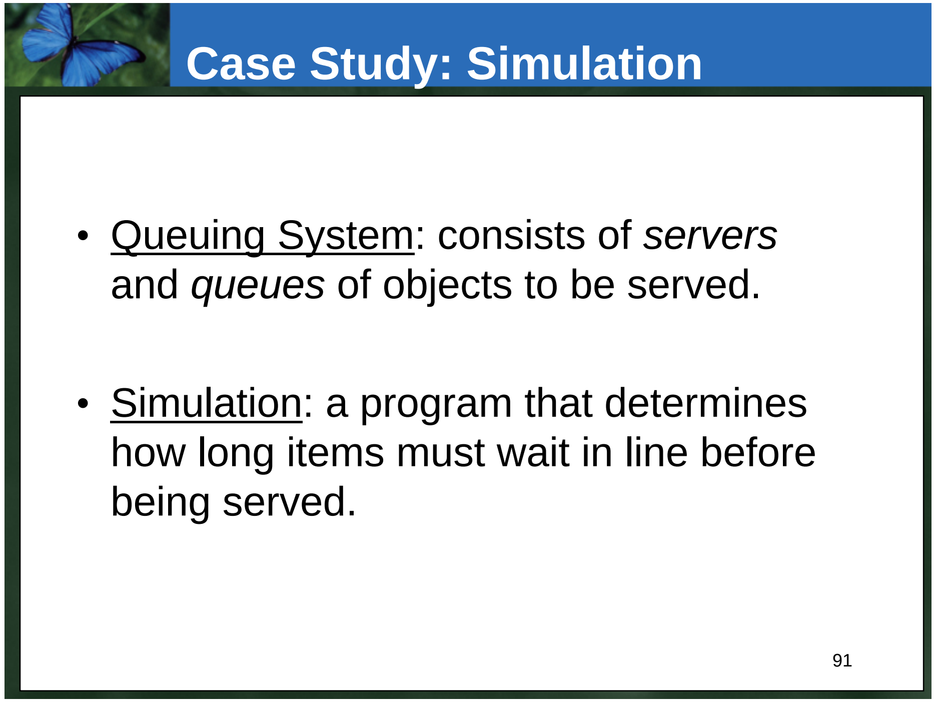
```
#include <iostream.h>
#include <ctype.h>
#include "stack.h"
#include "queue.h“
int main()
{
 StackType<char> s;
 QueType<char> q;
 char ch;
 char sItem, qItem;
 int mismatches = 0;
```

```
cout << "Enter string: " << endl;
while(cin.peek() != '\\n') {
  cin >> ch;
  if(isalpha(ch)) {
    if(!s.IsFull())
      s.Push(toupper(ch));
    if(!q.IsFull())
      q.Enqueue(toupper(ch));
  }
}
```



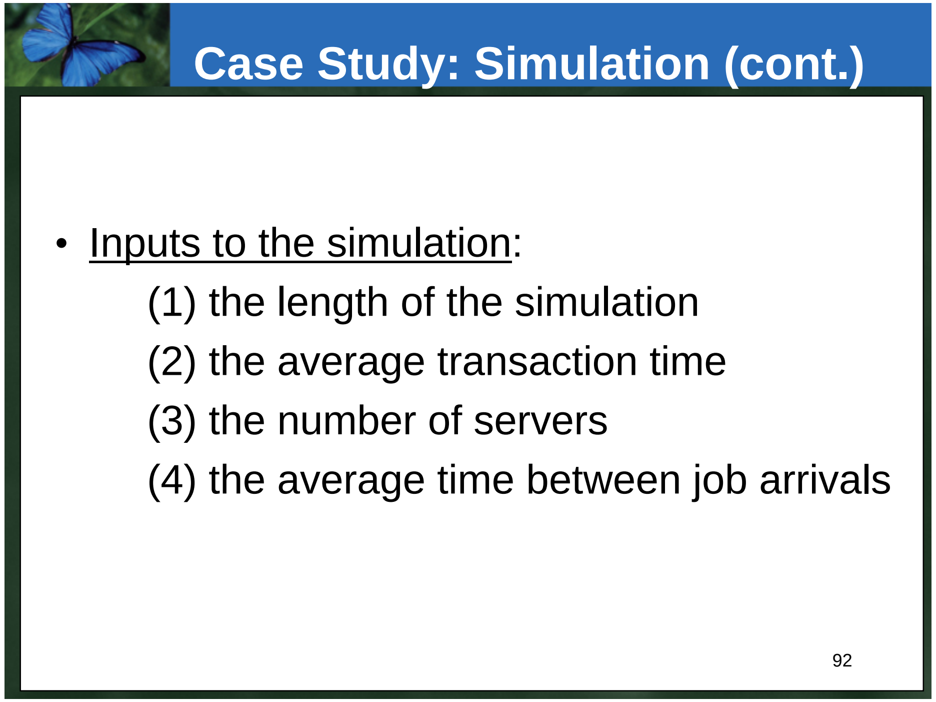
Example: recognizing palindromes

```
while( (!q.IsEmpty()) && (!s.IsEmpty()) ) {
  s.Pop(sItem);
  q.Dequeue(qItem);

  if(sItem != qItem)
    ++mismatches;
}
if (mismatches == 0)
  cout << "That is a palindrome" << endl;
else
 cout << That is not a palindrome" << endl;

return 0;
}
```



Case Study: Simulation

- Queuing System: consists of *servers* and *queues* of objects to be served.
- Simulation: a program that determines how long items must wait in line before being served.



Case Study: Simulation (cont.)

- Inputs to the simulation:
  - (1) the length of the simulation
  - (2) the average transaction time
  - (3) the number of servers
  - (4) the average time between job arrivals

Case Study: Simulation (cont.)

- Parameters the simulation must vary:
  - (1) number of servers
  - (2) time between arrivals of items
- Output of simulation: average wait time.

Case Study: Simulation (cont.)

- *Random-number generator* is used to vary parameter values
- Example: how to vary the time between arrivals of items

```
#include <cstdlib>
…
int randomInt = rand();
float value = float(rand())/float(RAND_MAX);
if (value <= arrivalProb) {
        // simulate the arrival of a new item
}
```

You may use function *srand(SEED)* to sepecify an initial seed before the first to rand()