Lecture Plan

• Binary Search on Sorted Lists

• Stack ADT

Stack ADT

• Describe a stack and its operations at a logical level
• Demonstrate the effect of stack operations using a particular implementation of a stack
• Implement the Stack ADT, in both an array-based implementation and a linked implementation
• Applications
  – Matching parentheses
  – Efficient way to save current state
  • Call stack
Stacks of Coins and Bills

Stacks of Boxes and Books

Stacks

• What do these composite objects all have in common?
Stacks

**Stack**
An abstract data type in which elements are added and removed from only one end (LIFO)

---

Stacks

*What operations would be appropriate for a stack?*

---

**Stack Methods**

- **Transformers**
  - Push
  - Pop

- **Observers**
  - isEmpty
  - isFull
  - Top

---
Stacks

class StackType
{
public:
    StackType();
    bool Empty() const;
    bool IsFull() const;
    void Push(ItemType item);
    void Pop();
    ItemType Top() const;
};

Array-Based Implementation

private:
    int top;
    ItemType items[MAX_ITEMS];
};

stack
.items
.top

Give a series of operations that could produce this situation.
Array-Based Implementation

Before we code, we must consider error conditions

*Stack overflow
– The condition that results from trying to push an element on to a full stack

*Stack underflow
– The condition that results from trying to pop an empty stack

---

Array-Based Implementation

```cpp
StackType::StackType()  
{  top = -1;  }

bool StackType::IsEmpty() const
{  return ( top = -1);  }

bool StackType::IsFull() const
{  return ( top == MAX_ITEMS);  }
```

---

Array-Based Implementation

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

---

What does `const` mean?

What is `FullStack()`?

Code for `Top/Pop`?
Array-Based Implementation

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

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

**What is EmptyStack?**

---

Class Interface Diagram

StackType class

---

Tracing Client Code

```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.Pop();
    charStack.Pop();
}
```
Tracing Client Code

Private data:
- `top`: -1
- `items`: 
  - [MAX_ITEMS-1] 
    - [2] 
    - [1] 
    - [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()`) 
{ `leEer = charStack.Top();` 
  `charStack.Pop();` }

Tracing Client Code

Private data:
- `top`: 0
- `items`: 
  - [MAX_ITEMS-1] 
    - [2] 
    - [1] 
    - [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()`) 
{ `leEer = charStack.Top();` 
  `charStack.Pop();` }

Tracing Client Code

Private data:
- `top`: 1
- `items`: 
  - [MAX_ITEMS-1] 
    - [2] 
    - [1] 
    - [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()`) 
{ `leEer = charStack.Top();` 
  `charStack.Pop();` }
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.Pop( );
      charStack.Pop();
    }
```

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.Pop( );
      charStack.Pop();
    }
```

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.Pop( );
      charStack.Pop();
    }
```
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.Pop();
      charStack.Pop();
    }

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.Pop();
      charStack.Pop();
    }

Tracing Client Code

char letter = 'K';
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();
    }
Tracing Client Code

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

Tracing Client Code

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

Tracing Client Code

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

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();
    }
```

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();
    }
```
Another Stack Implementation

- One advantage of an ADT is that the kind of implementation used can be changed.
- The dynamic array implementation of the stack has a weakness -- the maximum size of the stack is passed to the constructor as parameter.
- Instead we can dynamically allocate the space for each stack element as it is pushed onto the stack.

```
ItemType is char

class StackType
```

```
ItemType is float

class StackType
```
**Implementing Push**

```cpp
void StackType::Push ( ItemType newItem )
// Adds newItem to the top of the stack.
{
    if (IsFull())
        throw FullStack();
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
}
```
Implementing Pop / Top

```cpp
void StackType::Pop() // Remove top item from stack.
{
    if (!IsEmpty())
        throw EmptyStack();
    else
    {
        NodeType* tempPtr;
        tempPtr = topPtr;
        topPtr = topPtr -> next;
        delete tempPtr;
    }
}

ItemType StackType::Top() // Returns a copy of the top item in the stack.
{
    if (!IsEmpty())
        throw EmptyStack();
    else
        return topPtr->info;
}
```

Implementing IsFull

```cpp
bool StackType::IsFull() const
// Returns true if there is no room for another
// ItemType on the free store; false otherwise
{
    NodeType* location;
    try
    {
        location = new NodeType;
        delete location;
    }
    catch(std::bad_alloc exception)
    {
        return true;
    }
    return false;
}
```

Why is a destructor needed?

When a local stack variable goes out of scope, the memory space for data member topPtr is deallocated. But the nodes that topPtr points to are not automatically deallocated.

A class destructor is used to deallocate the dynamic memory pointed to by the data member.
Implementing the Destructor

stackType::~StackType()
// Post: stack is empty;
// All items have been deallocated.
{
    NodeType* tempPtr;

    while (topPtr != NULL)
    {
        tempPtr = topPtr;
        topPtr = topPtr->next;
        delete tempPtr;
    }
}