COP 3502: Computer Science I
Spring 2004

– Note Set 13 –
Data Structures: Stacks

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

Abstract Data Types and Data Structures

- Data structures are typically assembled from hierarchies.
  - The primitive data types of int, char, double, etc. occupy the lowest level of the hierarchy.
  - To represent more complex data/information these primitive types are combined to form larger structures.
  - You build each new level in the hierarchy by using one of the three basic primitives for constructing types: arrays, records, and pointers.
- Often, a type is defined in terms of its behavior rather than its implemented representation. This is called an Abstract Data Type or ADT.
- Many people also refer to ADTs as data structures.

Abstract Data Types and Data Structures (cont.)

- Most ADTs are defined by an interface that exports the ADT along with a collection of functions that define the behavior of that type.
- This has several advantages:

1. Simplicity – Hiding the internal representation of a type from the user means that the user has fewer details to understand.
2. Flexibility – Since the ADT is defined in terms of its behavior, the representation of the type can be changed in the implementation. As long as the interface remains the same the user is unconcerned with the implementation. This is a form of information hiding.
3. Security – The interface boundary acts like a wall that protects the implementation and the user from each other. If the user has knowledge of the representation they might be tempted to write applications based on the implementation rather than the behavior of the type, thus causing trouble should the implementation change.

Abstract Data Types and Data Structures (cont.)

- Most of the rest of the term will be devoted to studying many of the classical data structures and applications to which they are suitable.
- Part of the behavior of any ADT is a description of how the data type can be accessed. This is a major component in defining many different ADTs.
- We'll begin by looking at some fairly simply ADTs, albeit with many applications and move toward looking at some very general ADTs suitable to an even wider range of applications.
- As we did with the applications of searching and sorting, we will also be interested in the performance of our data structures and the applications to which they will be applied.

Stacks

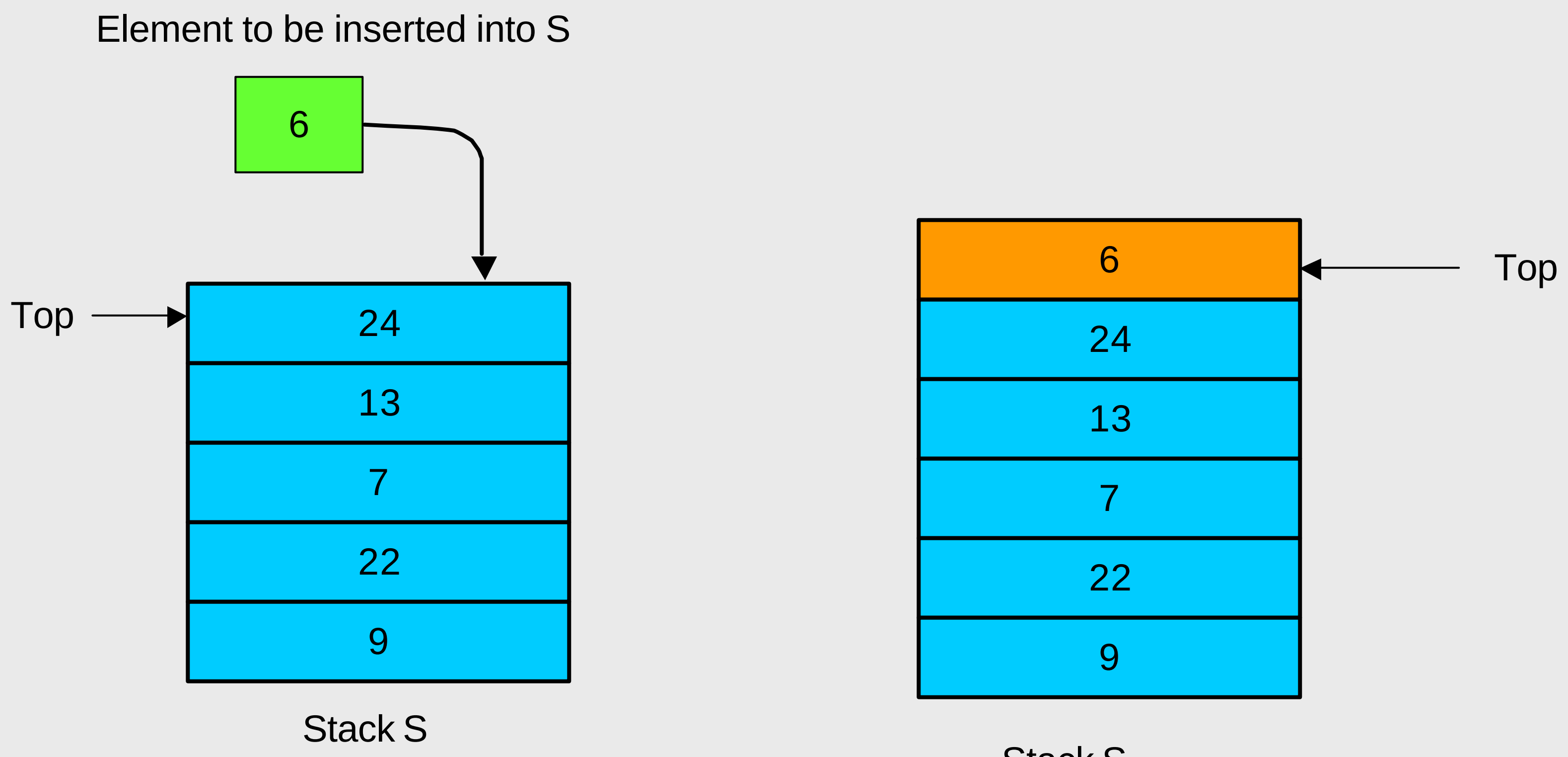
- The stack is a widely used data structure.
- It consists of a variable number of homogeneous elements (i.e., elements of the same type).
- The access policy for a stack is simple – the first element to be removed from the stack is the last element that was placed onto the stack.
- This access policy is also known as LIFO (Last In – First Out). This is analogous to a stack of trays in a cafeteria where you take the top tray and the stack moves up so that the next person in line would take the tray that was in the second position before you took your tray off the top, but now the tray they take is on the top.

Stacks (cont.)

- The basic insertion and deletion operations on a stack are given special names as is the end of the stack at which the insertions and deletions occur.
  - Insertion into a stack is called pushing the stack. So rather than an insert operation we have a push operation.
  - Deletion from a stack is called popping the stack. so rather than a delete operation we have a pop operation.
  - The end of the stack at which pushes and pops occur is referred to as the top of the stack.

LIFO Nature of a Stack

PUSH Operation

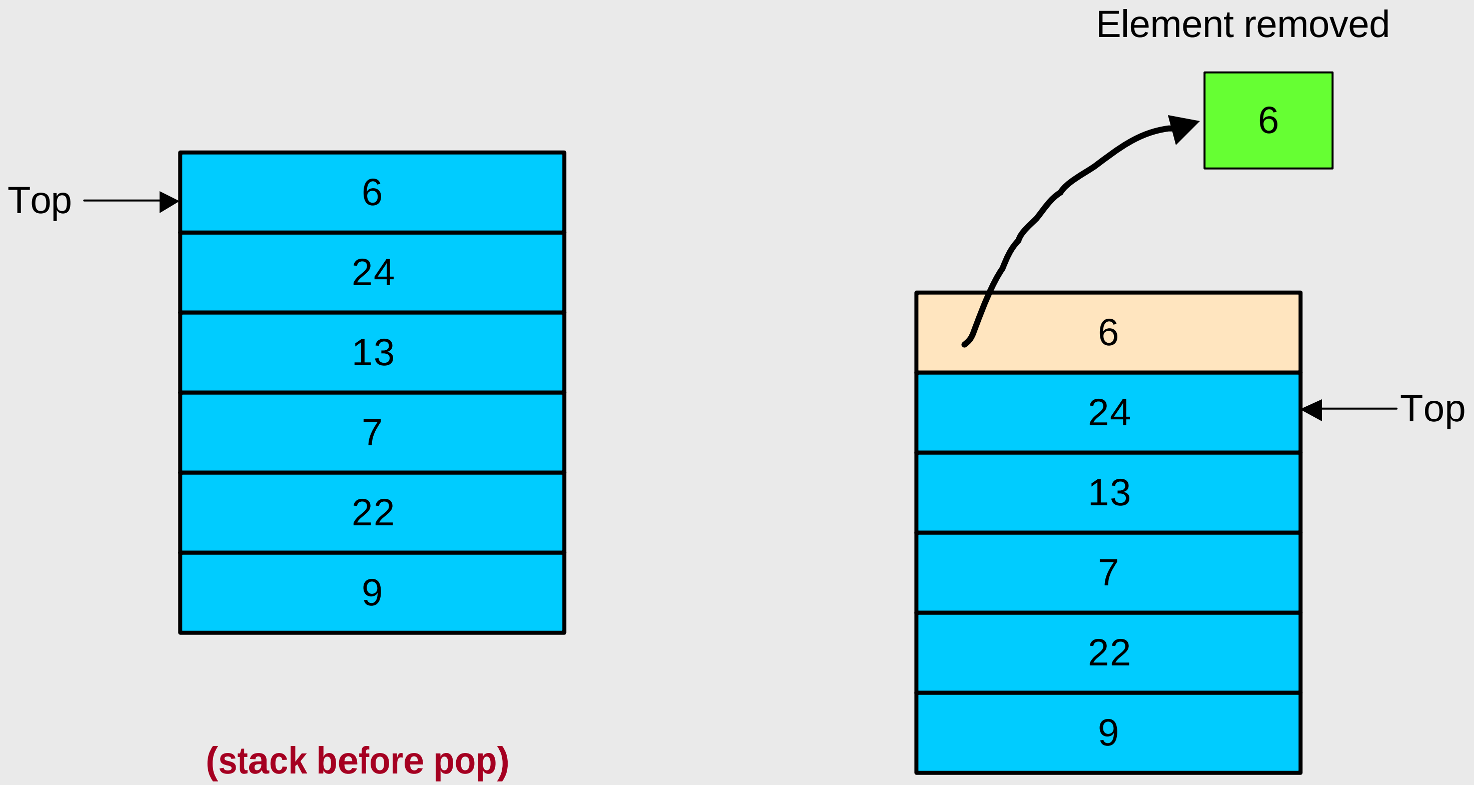


(before push)

(after push)

LIFO Nature of a Stack

POP Operation



(stack before pop)

(stack after pop)

Defining the Behavior of a Stack

- A more formal definition for a stack is that it is a restricted list in which entries are added to and removed from one designated end called the top.
- The operations which define the behavior of a stack are:
  - create(s): creates an empty stack named s.
  - isempty(s): returns true if stack s is empty, false otherwise.
  - push(s,x): Item x is added to stack x.
  - pop(s,x): Item x is loaded with the value from the top of stack s.
  - peek(s,x): The value on the top of stack s is loaded into x but is not removed from the top of stack x.

Implementing a Stack

- A stack can be implemented using static or dynamic memory.
- If static memory is utilized, there is the possibility that the stack can become full. When this occurs, it is not possible to add any additional elements to the stack until room becomes available via pop operations.
- Although it is also possible to exhaust dynamic memory, this is much less likely to occur and overflowing the stack becomes less of a concern with dynamic memory implementations.
- We'll see details of both implementations later, but for now let's concentrate on some applications for stacks and not worry about the implementation issues.

Parsing and Evaluating Arithmetic Expressions

- Often the logic of problems for which stacks are a suitable data structure involves the need to backtrack and return to a previous state.
  - For example, consider the problem of finding your way out of a maze. One approach is to probe a given path in the maze as deeply as possible. On finding a dead end, you need to backtrack to the previously visited maze locations to try other available paths. Such backtracking requires recalling the previous locations in the reverse order from which you visited them first time.
- While you don’t often need to extract yourself from a maze, designers of compilers are faced with a similar backtracking problem when evaluating arithmetic expressions.

Parsing and Evaluating Arithmetic Expressions (cont.)

- Consider the following expression: A + B / C + D. When you scan this in left-to-right order, it is impossible to tell upon initially encountering the plus sign whether or not you should apply the indicated addition operation to A and operand which follows the + operation symbol.
- Instead, you need to probe further into the expression to determine if an operation with higher precedence occurs.
- While you are probing deeper into the expression you need to stack the previously encountered operation symbols until you are certain of the operands to which they can be applied.

Parsing and Evaluating Arithmetic Expressions (cont.)

- The backtracking problem in expression evaluation is further compounded by the many different ways possible to represent the same algebraic expression.
  - For example, the following assignment statements should all result in the same order of arithmetic operations even though the expressions are written in distinctly different forms:

    Z = A * B / C + D

    Z = (A * B) / C + D

    Z = ((A * B) / C) + D

- The process of checking the syntax of such an expression and representing it in one unique form is called *parsing* the expression.
- One very common parsing method relies heavily on stacks.

Infix, Postfix and Prefix Notation

- Conventional algebraic notation is called *infix notation*. In infix notation the arithmetic operator appears between the two operands to which it is being applied.
- Infix notation may require parentheses to specify a desired order of operations.
  - For example, in the expression A / B + C, the division operation will occur first. If we want the addition to occur first, the expression must be parenthesized as:

    A / (B + C)

Infix, Postfix and Prefix Notation (cont.)

- Using *postfix notation* (also called reverse Polish notation after the nationality of its originator, Polish logician Jan Lukasiewicz), the need for parentheses is eliminated because the operator is placed directly after the two operands to which it applies.
  - Thus, the infix expression A / B + C would be written as A B / C + in postfix notation.
  - Similarly, the infix expression A / (B + C) would be written as A B C + / in postfix notation.

Infix, Postfix and Prefix Notation (cont.)

- Although simple infix expressions can be converted to postfix expressions using an intuitive process, a more systematic method is required for complicated expressions.

- Before looking at a complete algorithm for converting an infix expression into its postfix form, let's consider a simple algorithm for a human to do the same thing. Understanding this algorithm will give you a better insight as to how the computer algorithm actually works.

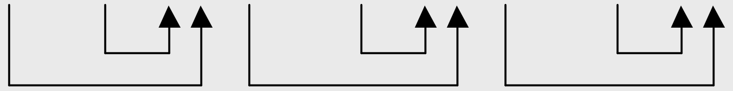
Simple Algorithm to Convert Infix to Postfix

1. Completely parenthesize the infix expression to specify the order of all operations.
2. Move each operator to the space held by its corresponding right parenthesis.
3. Remove all parentheses.

– Example: Infix expression: A / B ^ C + D * E – A * C

Step 1: (((A / (B ^ C)) + (D * E)) – (A * C))

Step 2: (((A / (B ^ C)) + (D * E)) – (A * C))



Step 2a: (((A (B C ^ / (D E * + (A C * –

Step 3: A B C ^ / D E * + A C * –

Practice Problems: Converting Infix to Postfix

- For practice convert each of the following infix expressions to postfix form using the previous algorithm. Answers are given on the next page.

1. A + B * C – D * E / F
2. A + B ^ C / D * E + F – A
3. A ^ B ^ C / 2 * B + 4
4. A / B + C / D – E * F ^ 2 / B
5. (A + B) * (C – B) / 2 ^ 4
6. (A + B) * (C – D)
7. A – B / (C + D * E)
8. ((A + B) * C – (D – E))/(F + G)

Practice Problems: Converting Infix to Postfix

Solutions

1. A B C * + D E * F / –
2. A B C ^ D / E * + F + A –
3. A B ^ ^ C 2 / B * 4 +
4. A B / C D / + E F 2 ^ * B / –
5. A B + C B - * 2 4 ^ /
6. A B + C D – *
7. A B C D E * + / –
8. A B + C * D E – – F G + /

Algorithm for Converting Infix to Postfix

```
while there are more characters in the input
 {
Read next symbol ch in the given infix expression.
If ch is an operand put it directly into the output.
If ch is an operator (i.e.*,/,+,-, or ^)
 {
    // check the item op at the top of the stack
         while (more items in the stack && precedence(ch) <=precedence (op)
         {
                  pop op and append it to the output.
                  // op becomes the next top element
         }
         push ch onto stack
     }
}
pop any symbols remaining in the stack
```

Example Using the Algorithm to Convert Infix to Postfix

- Example: Infix expression: **A / B ^ C + D * E – A * C**

Input char = A, operand sent to output: A

Input char = /, operator and stack is empty, push / [stack /]

Input char = B, operand sent to output: A B

Input char = ^, operator, precedence higher than top, push ^ [stack ^ /]

Input char = C, operand sent to output: A B C

Input char = +, operator, precedence lower than top, pop ^ [stack /]

continuing, precedence lower than top, pop / [stack empty]

output is A B C ^ /, push + stack[+]

Input char = D, operand sent to output: A B C ^ / D

Input char = *, operator and precedence higher than top, push * [stack * +]

Input char = E, operand sent to output: A B C ^ / D E

Input char = –, operator, precedence lower than top, pop * & +, push – stack[–]

Input char = A, operand, sent to output: A B C ^ / D E * + A

Input char = *, operator, precedence higher than top, push * stack[* –]

Input char = C, operand, sent to output: A B C ^ / D E * A C

Empty stack to output: A B C ^ / + D E * A C * –

What Happens if the Infix Expression Contains Parentheses?

- Essentially, nothing would change except that we would treat left parentheses (open parentheses) as an operator that is pushed onto the stack until its corresponding right parenthesis (closed parentheses) is encountered in the input string.
- Thus our algorithm would become:

If the operator is NOT a close parenthesis), then do this:

Continue popping off items off the stack and placing them in the output expression until you hit an operator with lower precedence than the current operator or until you hit an open parenthesis. At this point, push the current operator onto the stack.

Else

Pop off all operators off the stack one by one, placing them in the output expression until you hit the first(matching) open parenthesis. When this occurs, pop off the open parenthesis and discard both ()s.

Algorithm for Converting Infix to Postfix When the Infix Expression Contains Parentheses

```
while there are more characters in the input
 {
Read next symbol ch in the given infix expression.
If ch is an operand put it directly into the output.
If ch is an operator (i.e.*,/,+,-, ^, or (  )
 {
     // check the item op at the top of the stack
             while (more items in the stack && precedence(ch) <=precedence (op)
             {
                          pop op and append it to the output.
                          // op becomes the next top element
             }
             push ch onto stack
       }
}
pop any symbols remaining in the stack
```

only modification necessary to handle parentheses

Example Using the Algorithm to Convert Infix to Postfix

- Example: Infix expression: **(((1 + 2) * 4) / 2) + (6 * 3 + 2 – 4 / 2)**

  Input char = (, operator, push (, stack[( ]

  Input char = (, operator, push (, stack [(( ]

  Input char = (, operator, push (, stack [((( ]

  Input char = 1, operand sent to output: 1

  Input char = +, operator, TOS = (so push +, stack [+ ((( ]

  Input char = 2, operand sent to output: 1 2

  Input char =), operator, pop stack until first "(" is popped, output 1 2 +, stack [(( ]

  Input char = *, operator, TOS = (so push *, stack [* (( ]

  Input char = 4, operand sent to output: 1 2 + 4

  Input char =), operator, pop stack until first "(" is popped, output 1 2 + 4 *, stack = [( ]

  Input char = /, operator, TOS = (so push /, stack [/ ( ]

  Input char = 2, operand sent to output 1 2 + 4 * 2

  Input char =), operator, pop stack until first "(" is popped, output 1 2 + 4 * 2 /, stack empty [ ]

Example Using the Algorithm to Convert Infix to Postfix

- Example: Infix expression: **(((1 + 2) * 4) / 2) + (6 * 3 + 2 – 4 / 2)**
- Continuing…

Input char = +, operator, stack empty, push +, stack [+]
Input char = (, operator, push (, stack [(+]
Input char = 6, operand sent to output 1 2 + 4 * 2 / 6
Input char = *, operator, push *, stack [* (+]
Input char = 3, operand sent to output 1 2 + 4 * 2 / 6 3
Input char = +, operator, pop *, output 1 2 + 4 * 2 / 6 3 *, push +, stack [+ (+]
Input char = 2, operand sent to output 1 2 + 4 * 2 / 6 3 * 2
Input char = –, operator, pop +, output 1 2 + 4 * 2 / 6 3 * 2 + , push –, stack [– (+]
Input char = 4, operand sent to output 1 2 + 4 * 2 / 6 3 * 2 + 4
Input char = /, operator, push /, stack [/ – (+]
Input char = 2, operand sent to output 1 2 + 4 * 2 / 6 3 * 2 + 4 2
Input char =), pop /, pop –, pop (, output 1 2 + 4 * 2 / 6 3 * 2 + 4 2 / –, stack [+]
End of input string, pop remainder of elements from the stack
Empty stack to output: **1 2 + 4 * 2 / 6 3 * 2 + 4 2 / – +**

Evaluating a Postfix Expression

- The stack is also a very useful data structure for evaluating a postfix expression.
- Consider the simple example of the postfix expression A B C * + assuming the values for A, B, and C are 2, 4, and 6 respectively. Given these values the expression evaluates to 26.
- When evaluating such an expression the stack is used to hold the operands. So on the stack we would place the value for A, then the value for B, followed by the value for C. Next, we encounter the * operator, so we pop the top two elements from the stack apply the operation and push back onto the stack this result (in this case the value 24). Next, we encounter the + operator and once again pop the top two elements from the stack which are 24 and 2 and apply the operation which produces the value 26 which is pushed back onto the stack. Since we are at the end of the input expression, the top of the stack contains the result of evaluating this expression.

Algorithm for Evaluating a Postfix Expression

```
// Each operator in a postfix string expression refers to the previous
// two operands in the expression
repeat for every symbol in the input expression
{
    if the input symbol is an operand
        push it onto the stack
    if the input symbol is an operator
    {   pop right-hand operand from stack
        pop left-hand operand from stack
        perform the operation indicated by the operand symbol
        // this result becomes a subsequent operand
        push the result onto the stack
    }
}
```

Example Evaluating a Postfix Expression

- Example: Infix expression: **1 2 + 4 * 2 / 6 3 * 2 + 4 2 / – +**

Input symbol = 1, push 1, stack [1]

Input symbol = 2, push 2, stack [2 1]

Input symbol = +, pop right operand 2, pop left operand 1, 1 + 2 = 3, push 3, stack [3]

Input symbol = 4, push 4, stack [4 3]

Input symbol = *, pop RO= 4, pop LO= 3, 3 * 4 = 12, push 12, stack [12]

Input symbol = 2, push 2, stack [2 12]

Input symbol = /, pop RO= 2, pop LO= 12, 12/2 = 6, push 6, stack [6]

Input symbol = 6, push 6, stack [6 6]

Input symbol = 3, push 3, stack [3 6 6]

Input symbol = *, pop RO = 3, pop LO = 6, 6 * 3 = 18, push 18, stack [18 6]

Input symbol = 2, push 2, stack [2 18 6]

Input symbol = +, pop RO = 2, pop LO = 18, 18+2 = 20, push 20, stack [20 6]

Input symbol = 4, push 4, stack [4 20 6]

Example Evaluating a Postfix Expression

- Example: Infix expression: **1 2 + 4 * 2 / 6 3 * 2 + 4 2 / – +**
- **Continuing…**

Input symbol = 2, push 2, stack [2 4 20 6]

Input symbol = /, pop RO = 2, pop LO = 4, 4/2 = 2, push 2, stack [2 20 6]

Input symbol = –, pop RO = 2, pop LO = 20, 20 –2 = 18, push 18, stack [18 6]

Input symbol = +, pop RO = 18, pop LO = 6, 6 + 18 = 24, push 24, stack [24]

End of input, stack hold result of expression.

C Implementation for Stacks

```
/*****************************************************************************
   Arup Guha
   2/17/04
   COP 3502
   Lecture Example: Implementation of a stack class.
*****************************************************************************/

#include "stack.h"

// Creates a new empty stack struct and returns a pointer to it.
struct stackDT* NewStack() {

  struct stackDT *s;
  s = (struct stackDT*)malloc(sizeof(struct stackDT));
  s->top = 0;
  return s;
}
```

C Implementation for Stacks

```
// Pushes the character element onto the stack pointed to by stack.
// Assumes that the stack is NOT full.
void Push(struct stackDT *stack, char element) {

  stack->values[stack->top] = element;
  stack->top++;
}

// Pops the top character off the stack pointed to by stack.
// Assumes that the stack is NOT empty.
char Pop(struct stackDT *stack) {

  char retval = stack->values[stack->top-1];
  stack->top--;
  return retval;
}
```

C Implementation for Stacks

```
// Returns the number of elements in the stack pointed to by stack.
int StackDepth(struct stackDT *stack) {
  return stack->top;
}

// Returns 1 if no elements are in the stack pointed to by stack,
// 0 otherwise.
int StackIsEmpty(struct stackDT *stack) {
  return (stack->top == 0);
}

// Returns 1 if the entire stack structure pointed to by stack is full,
// 0 otherwise.
int StackIsFull(struct stackDT *stack) {
  return (stack->top == MAX_SIZE);
}
```

Header File for C Implementation of Stacks

```
/*****************************************************************************************
   Arup Guha    2/17/04    COP 3502
   Lecture Example: stack.h for a simple stack class. This stack stores characters.
******************************************************************************************/
#ifndef _stack_h
#define _stack_h
#include <stdio.h>
#define MAX_SIZE 20

struct stackDT {
  char values[MAX_SIZE];
  int top;
};
struct stackDT* NewStack();
void Push(struct stackDT *stack, char element);
char Pop(struct stackDT *stack);
int StackDepth(struct stackDT *stack);
int StackIsEmpty(struct stackDT *stack);
int StackIsFull(struct stackDT *stack);
#endif
```

Sample Application Using Stacks

```
/*****************************************************************************
   Arup Guha   2/17/04   COP 3502
   Class Example: A short program to utilize the stack class.
                  This program reads in a string and uses a
                  stack to print out the string in reverse order.
*****************************************************************************/
#include "stack.h"

int main() {

  char word[MAX_SIZE+1];
  int i;
  struct stackDT *my_stack;

  // Create the new stack.
  my_stack = NewStack();
```

Sample Application Using Stacks

```
  // Read in a word from the user.
  printf("Enter a word to reverse.\n");
  scanf("%s", word);

  // Push all the letters in the word onto the stack, one by one.
  for (i=0; i<strlen(word); i++)
    Push(my_stack, word[i]);

  // Pop off each of these elements one by one, printing them out as
  // they get popped until the stack is empty.
  while (!StackIsEmpty(my_stack))
    printf("%c",Pop(my_stack));

  printf("\n");

  return 0;
}
```