Data Structures

Lecture 2 - Stacks

Stacks

- A ***stack*** is an ordered set of data items that are inserted or deleted ***at one end only*** – the ***top***.
- A stack is dynamic, i.e., its size changes and its contents change.
- We cannot know the history of a stack; we only see its top.
- There are three primitive operations:
  - ***pop*** – remove an item from the top
  - ***push*** – insert an item on the top
  - ***empty*** – true if the stack has no contents

Abstract Data Types

- An ***abstract data type*** is a mathematical concept of data that does not necessarily have to be implemented on a computer.
- It allows us to study a data type's properties without worrying about implementation issues.
- In an abstract function or procedure, we will define them by use of preconditions and postconditions.
- In the case of a stack, we will define push, pop and empty using abstract data types.

ADT Definitions of Stack Operations

Type stackitem = …. //type of item on stack – it // varies

Abstract type stack = sequence of stackitem

Abstract function empty (s : stack) : boolean

postcondition: empty = (length(s) = = 0)

Abstract function pop(s : stack) : stackitem

precondition: ! empty(s)

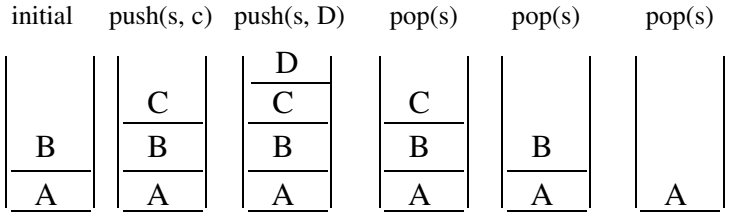
postcondition: pop = last (s')

s = substr(s', 1, length (s') - 1)

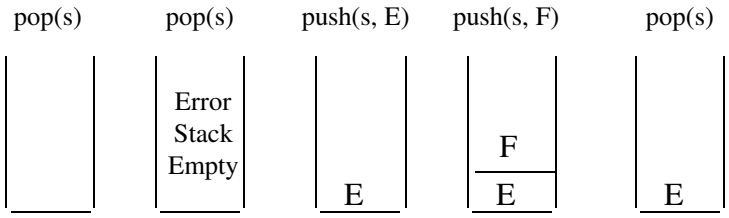
ADT Definitions of Stack Operations (continued)

Abstract procedure push(s: stack, elt: stackitem)

postcondition: s = s' + <elt>



ADT Definitions of Stack Operations (continued)



The Stack Class For Integers

```
#include    <iostream>
#include    <cstdlib>

using namespace std;

const int   StackSize = 100; // An arbitrary value
```

```
class  stack      {
public:
      bool empty(void);
      int         pop(void);
      void  push(int x);
      int         stacktop(void);
      bool  popandtest(int    &x);
      bool  pushandtest(int x);
      stack(void);
      stack(int x);
private:
      int   top;
      int   item[StackSize];
};
```

```
#include    "stacks.h"

// empty() -      Returns true if stack is empty
//                Returns false is stack is not
bool stack::empty(void)
{
      return((top == -1)? true: false);
}
```

```
// pop() -  Removes an item from the top of the
//          stack and returns it
//          Precondition - stack is nonempty
int   stack::pop(void)
{
      if (empty())      {
            cerr << "Stack underflow" << endl;
            exit(1);
      }

      return(item[top--]);
}
```

```
//push() -  Inserts an item from the top of the
//           stack
//          Precondition - stack has room for at
//           least one addition item
void  stack::push(int x)
{
      if (top == StackSize -1)      {
            cerr << "Stack overflow" << endl;
            exit(2);
      }
        item[++top] = x;
}
```

```
//stacktop() -    Returns a copy of the top item
//                without removing it from the stack
//          Precondition - stack is not empty
int   stack::stacktop()
{
      if (empty())      {
            cerr << "Stack underflow" << endl;
            exit(1);
      }

      return(item[top]);
}
```

```
//popandtest() -  Tests to see if there is anything
//                to pop and if there is, returns
//                it.
//                Using this with an empty stack is
//                NOT an unrecoverable error.
bool  stack::popandtest(int   &x)
{
      if (empty())
            return(true);
      else  {
            x = item[top--];
            return(false);
        }
}
```

```
//pushandtest() - Tests to see if there is room to
//                push another item and if there is,
//                does so.
//                Using this with a full stack is
//                NOT an unrecoverable error.
bool stack::pushandtest(int x)
{
      if (top == StackSize-1)
            return(true);
      else    {
            item[++top] = x;
                return(false);
      }
}
```

```
//stack() - Default constructor; initializes the
//          stack as empty
stack::stack(void)
{
      top = -1;
}

//stack() - Constructor; initializes the stack as
//          containing x as its only item.
stack::stack(int x)
{
      item[top = 0] = x;

}
```

How do we use stack objects?

```
if (underflow == s.popandtest(x))
      // Take corrective actions
else
      //    Use value of x
```

Similarly

```
if (overflow == s.pushandtest(x)
      //Overflow detected - x was not pushed
      //Take appropriate action
else
      //x was pushed successfully
```

Notations for Algebraic Expressions

- There are 3 different ways to write an algebraic expressions:
  - Infix – Standard algebra – 2 operands with the operator in the middle
  - Prefix – operator followed by 2 operands - aka ***Polish notation***
  - Postfix – 2 operands followed by an operator – aka ***Reverse Polish notation***

Examples of Infix, Prefix, Postfix

- Infix A + B, 3*x – y
- Prefix +AB, -*3xy
- Postfix AB+, 3x*y-

Converting to Infix to Postfix

A + B * C => A + [B*C]

=> A+[*BC**]

=> *A* [*BC**] +

=> *ABC* * +

Converting to Infix to Prefix

A + B * C => A + [B*C]

=> A+[**BC*]

=> + *A* [**BC*]

=> + *A* * *BC*

Conversion Examples

Infix	Prefix	Postfix
A + B	+ A B	A B +
A + B – C	- + A B C	A B + C –
(A+B)*(C-D)	* + A B – C D	A B + C D - *
A$B*C-D+E/F/(G+H)	+-*$ABCD//EF+GH	AB$C*D-EF/GH+/+

Evaluating Postfix Expressions

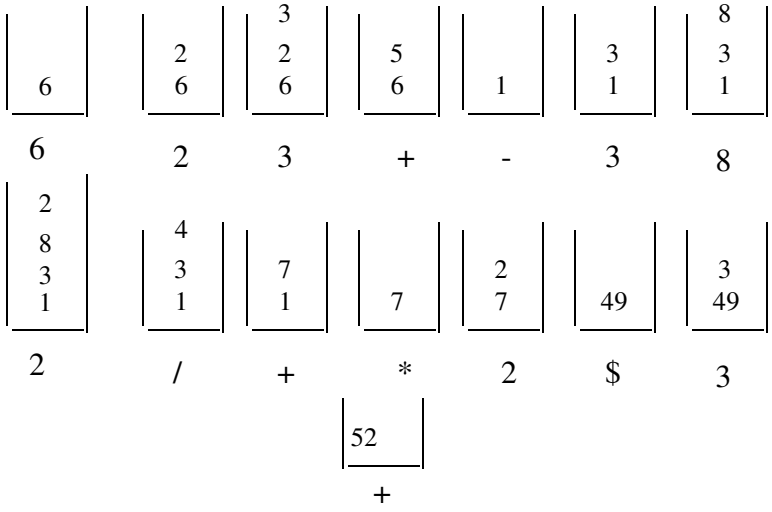
- Evaluating postfix expressions are of particular interest and can be done easily using a stack:

```
opndstk = {empty}
// Scan input string, reading one element as a time
//  into symb
while (not end-of-input)       {
   symb = next input character
```

```
    if (symb is an operand)
      push  symb onto opndstk
    else      {
              opnd2 = pop(opndstk)
              opnd1 = pop(opndstk)
              value = result of applying symb's operation  to
    opnd1                                   and opnd2
              push value onto opndstk
    }
  }
```

- Use the algorithm to evaluate:
  6 2 3 + - 3 8 2 / + * 2 $ 3 +

Evaluating Postfix Expressions (continued)



The Program ***Eval***

```
#include      "dstacks.h"
#include      <math.h>

const int MaxCols = 80;

double        eval(char expr[]);
inline int isdigit(char symb)
{
       return ('0' <= symb && symb <= '9');
}
double oper(int, double, double);
```

```
//main() -    A driver for the eval function
int    main(void)
{
       char   expr[MaxCols], c;

       //Read in an expression
       cin >> expr;

       //Write the original postfix expression
       cout << "The original postfix expression is "
              << expr << endl;
       // Write the value of the postfix expression
       cout << eval(expr);
       return(0);
}
```

```
// eval() -  A function which evaluates postfix
//           expressions.  The major limitation is that
//           operands can only be one digit.
double eval(char expr[])
{
       int    c, position;
       double opnd1, opnd2, value;
       stack  opndstk;
       //Scan the expression
       for (position = 0;
              (c = expr[position]) != '\0';
              position++)
              if (isdigit(c))
                 //It's an operand - push on the stack
                opndstk.push((double)(c-'0'));
```

```
              else   {
                     //It's an operator - pop two
                     // operands and perform the
                     // appropriate operation
                     opnd2 = opndstk.pop();
                     opnd1 = opndstk.pop();
                     value = oper(c, opnd1, opnd2);
                         opndstk.push(value);
              }
        return(opndstk.pop());
}
```

```
// oper() -  An auxiliary function for eval which
//           performs the appropriate operation as
//           indicated by the character
double oper(int symb, double op1,double op2)
{
       switch(symb)  {
         case '+': return(op1 + op2);
         case '-': return(op1 - op2);
         case '*': return(op1 * op2);
         case '/': return(op1 / op2);
         case '$': return(pow(op1, op2));
         default:   cerr << "Illegal operation"
                    << endl;
                    exit(1);
       }
        return((double)0);
}
```

Converting Infix to Postfix

- Infix notation uses the concept of precedence, that certain operations have to be performed before others and that this does not necessarily depend on where in the line it appears.
- In postfix notation, operations are performed based on their order in the expression.

Operator Precedence

\op2 op1	+	-	*	/	$	(	)
+	T	T	F	F	F	F	T
-	T	T	F	F	F	F	T
*	T	T	T	T	F	F	T
/	T	T	T	T	F	F	T
$	T	T	T	T	F	F	T
(	F	F	F	F	F	F	F
)	UN	UN	UN	UN	UN	UN	UN

Converting Infix to Postfix – An Example

	symb	postfix string	operator stack
1	A	A	
2	+	A	+
3	B	AB	+
4	*	AB	+*
5	C	ABC	+*
6		ABC*	+
7		ABC*+	

Converting Infix to Postfix – An Example

	symb	postfix string	operator stack
1	(		(
2	A	A	(
3	+	A	(+
4	B	AB	(+
5	)	AB+	
6	*	AB+	*
7	C	AB+C	*
8		AB+C*	

Algorithm For Converting Infix to Postfix

Initialize operator stack as empty

Initialize string to *Null*

WHILE there are more symbols

BEGIN

Read Symb

If symb is an operand

THEN Add to string

```
        ELSE BEGIN
          WHILE the stack is not empty
                    & prcd (stacktop, symb)
              DO BEGIN
                pop the stack and add to string
               Push symb on stack
              END
                { The stack is now empty or
                      sym has precedence }
        END
END
```

Postfix Program

```
#include     "cstack.h"
#include     <cstdlib>
using namespace std;

const int    MaxCols = 80;
void   postfix(char infix[], char postr[]);
bool   prcd(char op1, char op2);

inline       bool isoperand(char c)
{
       return('0' <= c && c <= '9'
              || 'a' <= c && c <= 'z'
              || 'A' <= c && c <= 'Z');
}
```

```
// main() -  A driver program for the function postfix
int    main(void)
{
       char   infix[MaxCols], postr[MaxCols];

       cin >> infix;
       cout << "The original infix expression is :"
              << infix << endl;

       postfix(infix, postr);
       cout << postr << endl;
       return(0);
}
```

```
// postfix() -      Convert infix expressions into
//                  postfix expressions
void   postfix(char infix[], char postr[])
{
       int    position, und;
       int    outpos = 0;
       char   topsymb = '+';
       char   symb;
       stack  opstk;

        // Scan the entire string
       for  (position = 0;
                     (symb = infix[position]) != '\0';
                     position++)
```

```
            // Operands just get added to the string
            if (isoperand(symb))
                  postr[outpos++] = symb;
            else   {
            // The operator with higher precedence
            // gets written first.  Keep popping the
            // stack until it is either empty or the
            // top symbol has lower precedence than
            // the current operator
               und = opstk.popandtest(topsymb);
               while (! und && prcd(topsymb,symb))  {
                 postr[outpos++] = topsymb;
                 und = opstk.popandtest(topsymb);
               }
```

```
            // Push the top symbol on the stack if
            // it lacks precedence over the current
            // symbol.  Otherwise push the current
            // symbol.
            if (!und)
                  opstk.push(topsymb);
            if (und || (symb != ')'))
                  opstk.push(symb);
            else
                        topsymb = opstk.pop();
            }
            // Keep popping the stack and writing
            // operators on the end of the string
            while (!opstk.empty())
                  postr[outpos++] = opstk.pop();
            postr[outpos] = '\0';
}
```

```
// prcd() -  Returns true if op1 has precedence over
//           op2
//           Returns false if op2 has precedence
//           over op1
bool prcd(char op1, char op2)
{
       // Everything has precedence over (
       if (op1 == '(' || op2 == '(')
              return(false);
        // ) has precedence over everything else
       if (op2 == ')')
              return(true);
```

```
       switch(op1)   {
         // + and - have precedence left to right
         // They have no precedence over other
         // operators
         case '+':
         case '-':  if (op2 == '*' || op2 == '/'
                           || op2 == '$')
                           return(false);
                    else
                           return(true);
```

```
            // * and / have left to right precedence
            // $ has precedence over them
            // $ has right to left precedence
            case '*':
            case '/':
            case '$':   if (op2 == '$')
                              return(false);
                        else
                              return(true);
           }

}
```

Generic Stack Class Using A Template

```
// a Generic Stack class
template <class StackType> class stack  {
public:
       stack(void);
       void          push(StackType x);
       StackType     pop(void);
       Boolean       Empty(void);
private:
       Boolean       Full(void);
       void          Error(String Message);
       StackType     s[SIZE];
       int           top;
};
```

```
//stack() -  Constuctor for the Generic stack class
template <class StackType> stack<class StackType>::
       stack(void)
{
       top = 0;

}
template <class StackType> void
       stack<class StackType>::push(StackType x)
{
       if (Full())
              Error("Stack overflow");
        s[top++] = x;
}
```

```
template <class StackType> StackType
              stack<class StackType>::pop(void)
{
       if (Empty()) Error("Stack underflow");
       return(s[--top]);
}

template <class StackType> Boolean      stack<class
StackType>::Empty(void)
{
       if (top == 0)
              return(True);
       else
              return(False);
}
```

```
template <class StackType> Boolean
       stack<class StackType>::Full(void)
{
       if (top == SIZE)
              return(True);
       else
              return(False);
}

template <class StackType> void
       stack <class StackType>::Error(String Message)
{
       cerr << Message << endl;
       exit(1);
}
```