Stacks and their Applications

Lecture 23
Sections 18.1 - 18.2

Robb T. Koether

Hampden-Sydney College

Fri, Mar 16, 2018

1. Stacks

2. The Stack Interface

3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions

4. Assignment

Outline

1. Stacks
2. The Stack Interface
3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions
4. Assignment

Stacks

Definition (Stack)

A stack is a list that operates under the principle “last in, first out” (LIFO). New elements are pushed onto the stack. Old elements are popped off the stack.

- To enforce the LIFO principle, we use a list and push and pop at the same end.

Outline

1. Stacks

2. The Stack Interface

3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions

4. Assignment

Stack Constructors

Stack Constructors

```
Stack();
Stack(const Stack& s);
```

- `Stack()` constructs an empty stack.
- `Stack(Stack&)` constructs a copy of the specified stack.

Stack Inspectors

Stack Inspectors

```
T top() const;
int size() const;
bool isEmpty() const;
```

- `top()` gets a copy of the element at the top of the stack (but does not remove it).
- `size()` gets the number of elements in the stack.
- `isEmpty()` determines whether the stack is empty.

Stack Mutators

Stack Mutators

```
void push(const T& value);
T pop();
void makeEmpty();
```

- `push()` pushes the specified value onto the top of the stack.
- `pop()` pops and returns the element off the top of the stack.
- `makeEmpty()` makes the stack empty.

Other Stack Member Functions

Other Stack Member Functions

```
    bool isValid() const;
```

- `isValid()` determines whether the stack has a valid structure.

Other Stack Functions

Other Stack Functions

```
istream& operator>>(istream& in, Stack& s);
ostream& operator<<(ostream& out, const Stack& s);
```

- **operator**>>() reads a Stack object from the input stream.
- **operator**<<() writes a Stack object to the output stream.

Implementation of Stacks

- Which push and pop functions should we use?
  - `pushFront()` and `popFront()`, or
  - `pushBack()` and `popBack()`.
- Choose a List class for which pushing and popping at one end will be efficient.

The Input Facilitator

- One must be careful when reading a stack.

```
{10, 20, 30, 40, 50}
```

- As the values are read from left to right, they should be pushed onto the stack (at one end or the other).
- Which end, left or right, is the "top" of the stack? (It matters.)
- When we display the stack, it should look the same regardless of the kind of List we used.
- Do we need to write new `input()` and/or `output()` functions?

Outline

1. Stacks

2. The Stack Interface

3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions

4. Assignment

Outline

1. Stacks

2. The Stack Interface

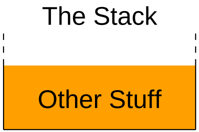
3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions

4. Assignment

Handling Function Calls

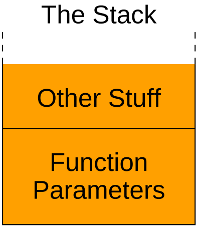
- When a function is called, the program
  - Pushes the values of the parameters.
  - Pushes the address of the next instruction (to which the function should return later).
  - Allocates space on the stack for the local variables.
  - Branches to the first line in the function.

Handling Function Calls



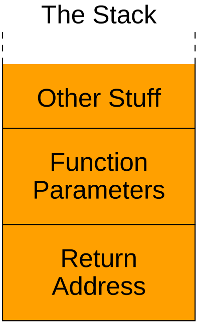
Begin with the current stack

Handling Function Calls



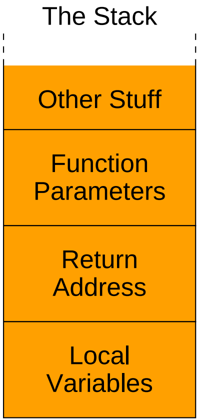
Push the function parameters

Handling Function Calls



Push the return address

Handling Function Calls

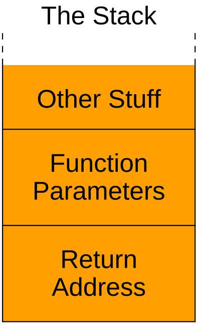


Push the local variables

Handling Function Calls

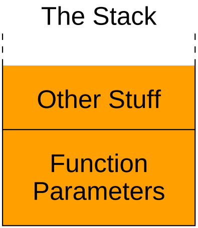
- When a function returns, the program
  - Pops the values of the local variables.
  - Pops the return address and stores it in the IP register.
  - Pops the parameters.
- The stack has now been returned to its previous state.
- Execution continues with the instruction in the IP register.

Handling Function Calls



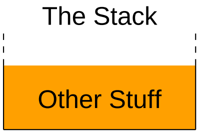
Pop the local variables

Handling Function Calls



Pop the return address

Handling Function Calls



Pop the function parameters

Outline

1 Stacks

2 The Stack Interface

3 Stack Applications
- Function Calls
- Infix, Postfix, and Prefix Notation
- Infix Expression Evaluation
- Postfix Expressions

4 Assignment

Infix Notation

- An infix expression is an arithmetic expression in which the binary operators are written in between the operands.
- For example, to add 3 and 4, we write

$$3 + 4.$$

Postfix Expressions

- In a postfix expression, the operator is written *after* the operands.
- For example, to add 3 and 4, we write

$$3\;4\;+.$$

- The infix expression $2 * 3 + 4 * 5$ would be written as

$$2\;3\;*\;\;4\;5\;*\;+$$

  in postfix notation.

Prefix Expressions

- In a prefix expression, the operator is written *before* the operands.
- For example, to add 3 and 4, we write

$$+\ 3\ 4.$$

- The infix expression $2 * 3 + 4 * 5$ would be written as

$$+ * 2\ 3 * 4\ 5$$

  in prefix notation.

Outline

1. Stacks
2. The Stack Interface
3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions
4. Assignment

Fully Parenthesized Infix Expressions

- With infix expressions, the operations are not necessarily performed from left to right.

Fully Parenthesized Infix Expressions

- With infix expressions, the operations are not necessarily performed from left to right.
- Infix expressions may require parentheses to specify the order of operation.

Fully Parenthesized Infix Expressions

- With infix expressions, the operations are not necessarily performed from left to right.
- Infix expressions may require parentheses to specify the order of operation.
- Precedence and associativity rules allow us to omit some of the parentheses.

Fully Parenthesized Infix Expressions

- With infix expressions, the operations are not necessarily performed from left to right.
- Infix expressions may require parentheses to specify the order of operation.
- Precedence and associativity rules allow us to omit some of the parentheses.
- A fully parenthesized expression requires no precedence or associativity rules.

Fully Parenthesized Infix Expressions

- With infix expressions, the operations are not necessarily performed from left to right.
- Infix expressions may require parentheses to specify the order of operation.
- Precedence and associativity rules allow us to omit some of the parentheses.
- A fully parenthesized expression requires no precedence or associativity rules.
- In a fully parenthesized expression, there is a pair of parentheses for every operator.

Examples

- The expression $1 + 2 * 3$ would be fully parenthesized as
$$(1 + (2 * 3)).$$
- The expression $2 * 3 + 4/5 - 6$ would be fully parenthesized as
$$(((2 * 3) + (4/5)) - 6).$$

Infix Expression Evaluation

- We may use a pair of stacks to evaluate a fully parenthesized infix expression.
- The expression contains four types of token:
    - Left parenthesis (
    - Right parenthesis)
    - Number, e.g., 123
    - Operator $+$, $-$, $*$, $/$

Infix Expression Evaluation

- To evaluate the expression we need a stack of numbers and a stack of operators.
- Read the tokens from left to right and process them as follows:

Token	Action
Left parenthesis	No action
Number	Push the number onto the number stack
Operator	Push the operator onto the operator stack
Right Parenthesis	1. Pop two numbers off the number stack 2. Pop one operator off the operator stack 3. Perform the operation on the numbers 4. Push the result onto the number stack

Example

- Use the algorithm to evaluate the expression

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack

Begin with an empty stack

Example

Token	Number Stack	Operator Stack
(		

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$\ast$	2	$\ast$
5	2 5	$\ast$
)	10	
$+$	10	$+$

$$(((2 \ast 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$
/	10 6	$+$ /

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$
/	10 6	$+$ /
3	10 6 3	$+$ /

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$
/	10 6	$+$ /
3	10 6 3	$+$ /
)	10 2	$+$

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$
/	10 6	$+$ /
3	10 6 3	$+$ /
)	10 2	$+$
)	12	

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$
/	10 6	$+$ /
3	10 6 3	$+$ /
)	10 2	$+$
)	12	
$-$	12	$-$

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$
/	10 6	$+$ /
3	10 6 3	$+$ /
)	10 2	$+$
)	12	
$-$	12	$-$
8	12 8	$-$

$$(((2 * 5) + (6/3)) - 8)$$

Example

Token	Number Stack	Operator Stack
(		
(		
(		
2	2	
$*$	2	$*$
5	2 5	$*$
)	10	
$+$	10	$+$
(	10	$+$
6	10 6	$+$
/	10 6	$+$ /
3	10 6 3	$+$ /
)	10 2	$+$
)	12	
$-$	12	$-$
8	12 8	$-$
)	4	

$$(((2 * 5) + (6/3)) - 8)$$

Infix Expression Evaluation

- Run the program `InfixEvalFullParen.cpp`.

Outline

1. Stacks

2. The Stack Interface

3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions

4. Assignment

Postfix Expression Evaluation

Example (Postfix Expressions)

- Expression: 3 4 $+$ 5 6 $+$ $*$.
- Left operand of $*$ is 3 4 $+$.
- Right operand of $*$ is 5 6 $+$.

- In postfix expressions, parentheses are never needed!

Postfix Expression Evaluation

- To evaluate a postfix expression we need a stack of numbers.
- Read the tokens from left to right and process them as follows:

Token	Action
Number	Push the number onto the number stack
Operator	1. Pop two numbers off the number stack 2. Pop one operator off the operator stack 3. Perform the operation on the numbers 4. Push the result onto the number stack

Postfix Expression Evaluation

Example (Postfix Expressions)

- The fully parenthesized infix expression

$$(((2 * 5) + (6/3)) - 8)$$

  can be written as

$$2 * 5 + 6/3 - 8$$

- As a postfix expression, it is 2 5 $*$ 6 3 $/$ $+$ 8 $-$

Example

Token	Number Stack
2	2

$2\ 5\ *\ 6\ 3\ /\ +\ 8\ -$

Example

Token	Number Stack
2	2
5	2 5

$$2\;5\;*\;6\;3\;/\;+\;8\;-$$

Example

Token	Number Stack
2	2
5	2 5
$*$	10

$$2\ 5\ {\color{red}*}\ 6\ 3\ /\ +\ 8\ -$$

Example

Token	Number Stack
2	2
5	2 5
$*$	10
6	10 6

$$2\ 5\ *\ 6\ 3\ /\ +\ 8\ -$$

Example

Token	Number Stack
2	2
5	2 5
$*$	10
6	10 6
3	10 6 3

$$2\;5\;*\;6\;3\;/\;+\;8\;-$$

Example

Token	Number Stack
2	2
5	2 5
$*$	10
6	10 6
3	10 6 3
/	10 2

$$2\ 5\ *\ 6\ 3\ /\ +\ 8\ -$$

Example

Token	Number Stack
2	2
5	2 5
$*$	10
6	10 6
3	10 6 3
/	10 2
+	12

$$2\ 5\ *\ 6\ 3\ /\ +\ 8\ -$$

Example

Token	Number Stack
2	2
5	2 5
$*$	10
6	10 6
3	10 6 3
$/$	10 2
$+$	12
8	12 8

$$2\ 5\ *\ 6\ 3\ /\ +\ 8\ -$$

Example

Token	Number Stack
2	2
5	2 5
$*$	10
6	10 6
3	10 6 3
/	10 2
$+$	12
8	12 8
$-$	4

$$2\ 5\ *\ 6\ 3\ /\ +\ 8\ -$$

Postfix Expression Evaluation

- Run the program `PostfixEvaluator.cpp`.

Outline

1. Stacks
2. The Stack Interface
3. Stack Applications
   - Function Calls
   - Infix, Postfix, and Prefix Notation
   - Infix Expression Evaluation
   - Postfix Expressions
4. Assignment

Assignment

Assignment

- Read Sections 18.1 - 18.2, 18.7 - 18.8.