POSTFIX EXPRESSIONS CALCULATOR DEMO

Data Structures

CSC 223

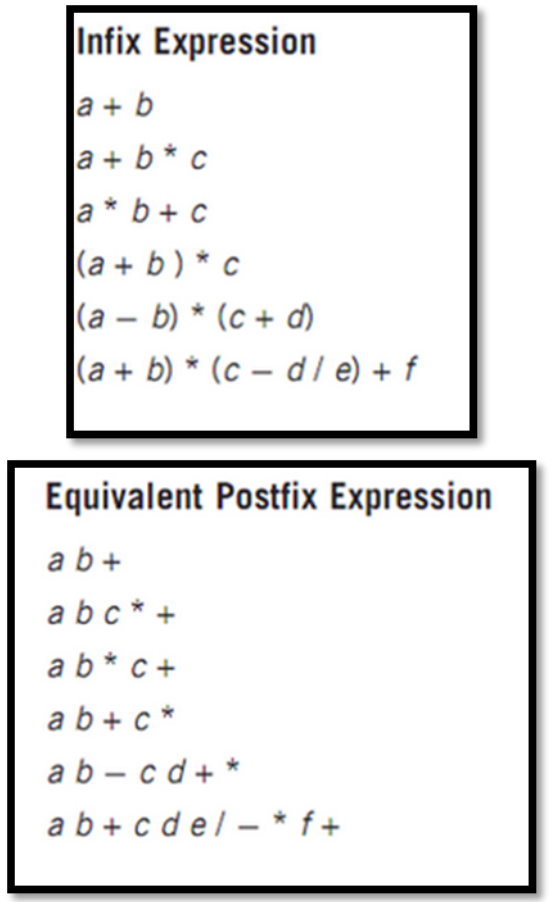
Writing Mathematical Expressions

- **Infix notation** is usually used for writing arithmetic expressions
- The operator is written between the operands: *a* + *b*
- The expression evaluates from left to right
- The operators have precedence
- Parentheses can be used to override precedence

- **Prefix notation** is an alternate notation that was introduced by A Polish mathematician named Jan Lukasiewicz
  - Prefix notation was often referred to as Polish notation.
- **Postfix notation** was proposed by the Austrailan Philospher and early computer scientist, Charles L. Hambin in the late 1950s.
  - This notation was know as Reverse Polish Notation (RPN) and was widely used in early HP calculators
- In this notation, the operators appear in the order required for computation: *a* + *b* * *c* becomes *a b c* * +

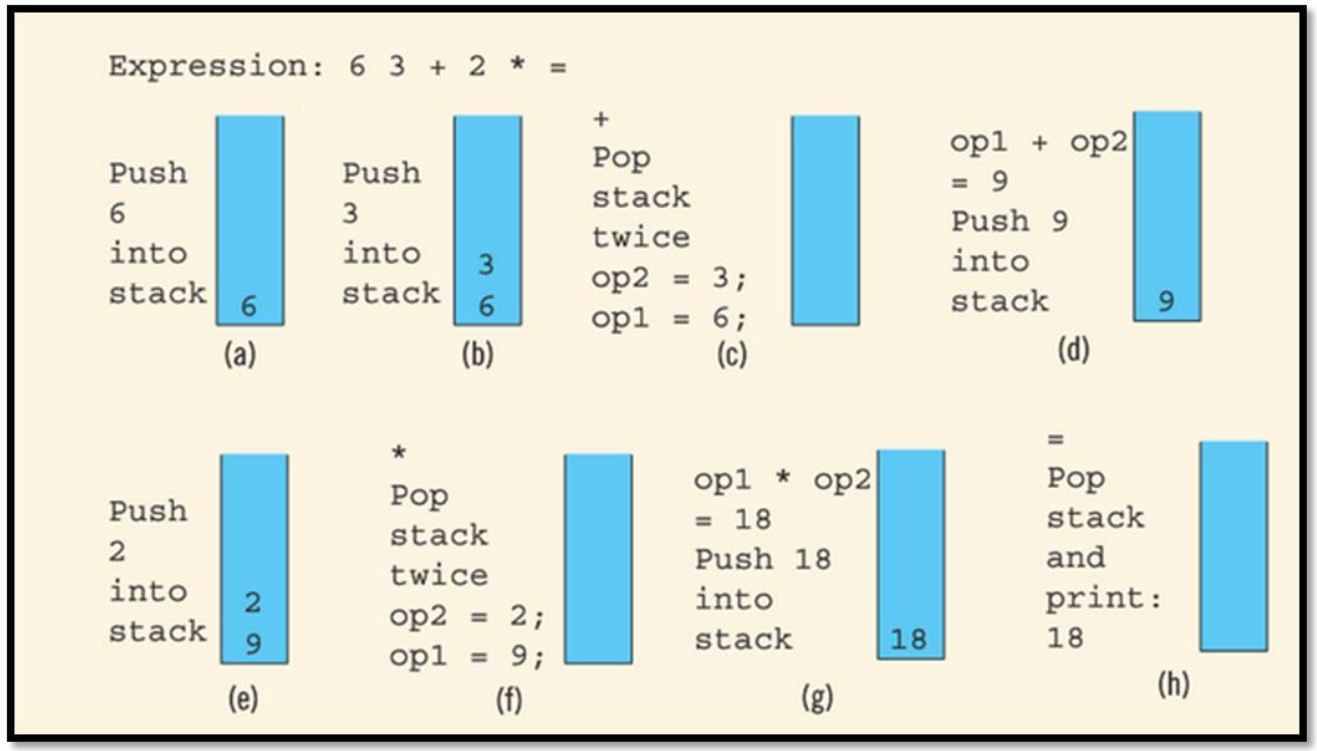
Postfix Expressions in Computation

- Many compilers convert arithmetic expressions to postfix notation before translating the expressions into machine code
- The general algorithm to evaluate a postfix expression is:
  - Scan expression from left to right
  - When an operand is found, push it onto the operand stack
  - When an operator is found, pop the top two operands on the stack
  - Perform the operation
  - Push the result back onto the stack
  - Continue until the end of the expression is reached
- The result will be the final value on the stack



Infix Expression	Equivalent Postfix Expression
a + b	a b +
a + b * c	a b c * +
a * b + c	a b * c +
(a + b) * c	a b + c *
(a – b) * (c + d)	a b – c d + *
(a + b) * (c – d / e) + f	a b + c d e / – * f +

Example



Errors

- Errors occur when -
  - An operator is encountered and there are fewer than two operands on the stack
  - The end of the expression is reached and there is no result on the stack
  - The end of the expression is reached and there is more than one value on the stack
  - An illegal operator is part of the expression
- These errors are caused by mal-formed expressions

PostFixCalculator Class - Analysis

- In this activity, we will look at the development of a class that evaluates a postfix expression
- There are some things to consider before we design the ADT for this class
- We will use a stack as part of the evaluation and for this design, we will use the stack class from the C++ Standard Template Library (STL)
    - The documentation for this class can be found in the reference section of the Cplusplus.com website under the Containers tab
- The member functions are like what we used in the classes we developed but there are some differences, which can be found by looking at the reference page.

PostFixCalculator Class - Analysis

- Another aspect to consider is how to split the expression into its tokens
- A function called `strtok` in the `<cstring>` library that will do that for us
- The preconditions for this function are:
  - A C string to truncate is provided, which will be modified by being broken into smaller strings (tokens).
  - Alternatively, a null pointer may be specified, in which case the function continues scanning where a previous successful call to the function ended.
  - Another C string containing the delimiter characters is provided and these can be different from one call to another
- The postconditions are:
  - If a token is found, a pointer to the beginning of the token is returned, otherwise, a null pointer is returned
  - A null pointer is always returned when the end of the string (i.e., a null character) is reached in the string being scanned.

PostFixCalculator Class - ADT

The domain (instance variables) of the PostFixCalculator class are:

- A string to store the expression to be evaluated
- The C string equivalent of the expression string
- A stack to hold the intermediate results of the calculation

The operations are:

- Functions to get the string from the client and return the string
- A function to split the string into tokens and determine if the token is an operator or operand
  - An operand token is converted to a number and pushed on the results stack
  - An operator token gets passed to a function which determines what kind of operator it is and applies the appropriate arithmetic expression, pushing the result back onto the results stack
- A function to get the result from the stack and return it to the client

PostFixCalculator Class - Exceptions

- Several errors caused by malformed postfix expressions were identified in the discussion on how to evaluate them
- The class will also contain embedded exception classes to detect and handle the various errors:
  - Too many operands
  - Too few operands
  - Illegal operators
  - Too many results on the stack
  - No result on the stack

Postfix Expression Calculator

- We'll also write a client program to read postfix expressions from a file and use a PostFixCalculator object to evaluate them.
- The client will write the expression and the result (or errors if detected) to a file.
- Test data is on a file called "RPNData.txt" and looks like this:

```
35 27 + 3 *
26 28 + 32 2 ; - 5 /
23 30 15 * /
2 3 4 +
20 29 9 * ;
25 23 - +
34 24 12 7 / * + 23 -
```

- Lines 1, 3, and 7 produce the following results: 186.00, 0.05, 52.14
- The other lines are malformed and produce error messages

UML Diagram

PostFixCalculator	throws
-pFixExp: string	
-expression: *char	
-resultStack: stack<double>	
+getPFixExp(): string	
+setPFixExp(string): void	
+evaluate(): void	
+evaluateOpr(char&) : void	TooFewOperands DivideByZero InvalidOperator
+getResult(): double	TooManyOperands ErrorInExpression

IPO Chart for client (main.cpp)

Input	Process	Output

A file containing postfix expressions	• Open and validate the input file
	• Open the output file and send formatting information
	• Instantiate a postFixCalculator object
	• Read an expression from the input file
	• While more data
	  • Send the expression to the PFC object
	  • Write the expression to the output file
	  • Get the PFC object to evaluate the expression (record errors that occur on output file)
	  • Get the result from the PFC (record error that occur on output file)
	  • Write the result to the output file
	  • Read an expression from the input file
	• Close input and output files
		A file containing the results of the evaluation

`evaluate()`

Precondition	Algorithm	Postcondition

Postfix expression string is provided	• Allocate character array based on size of string
	• Convert postfix expression string to expression character array
	• Initialize strtok and get the first token
	• While more tokens
	  • If token is a digit
	  • Convert token to double
	  • Push onto stack
	  • Else
	  • Extract operator character from token and send to evaluateOpr()
	  • Get the next token
	• Delete expression character array
		Expression has been evaluated and the result is in the stack

evaluateOpr()

Precondition	Algorithm	Postcondition

The operator as a character is provided	• Pop the second operand from the stack
	• If the stack is empty
	  • Throw "too few operands" exception
	• Else
	  • Pop the first operand from the stack
	  • If operator is +, add operands 1 and 2 and push result onto the stack
	  • Elseif operator is -, subtract operand 2 from operand 1 and push the result onto the stack
	  • Elseif operator is *, multiply operands 1 and 2 and push the result onto the stack
	  • Elseif operator is /,
	  • If operand 2 is 0, throw "Division by 0" exception
	  • Else divide operand 1 by operand 2 and push the result on the stack
	  • Else
	  • Clear the stack
	  • Throw "Invalid instruction" exception
		• The most recent expression has been evaluated and the result pushed on the stack
		• exceptions are thrown if the specified error conditions are detected

`getResult()`

Precondition	Algorithm	Postcondition

The expression has been evaluated and the result is the only item on the stack	• If the stack is not empty, pop the result from the stack
	  • If the stack is empty
	  • Return the result
	  • Else
	  • Clear the stack
	  • Throw "Too Many Operands" exception
	• Else
	  • Clear the stack
	  • Throw "Error in Expression" exception
		• Returns the result
		• exceptions are thrown if the specified error conditions are detected

Output File

RpnOutput.txt

```
1  ------------------------------------------
2  35 27 + 3 * = 186.00
3  ------------------------------------------
4  26 28 + 32 2 ; - 5 / = Illegal operator
5  Error in expression
6  ------------------------------------------
7  23 30 15 * / = 0.05
8  ------------------------------------------
9  2 3 4 + = Too many operands
10 ------------------------------------------
11 20 29 9 * ; = Illegal operator
12 Error in expression
13 ------------------------------------------
14 25 23 - + = Not enough operands
15 Error in expression
16 ------------------------------------------
17 34 24 12 7 / * + 23 - = 52.14
18
```