CIS 200 - Lab 8

Lab Instructors: Mehdi Mohammadi, John P. Baugh

#include <iostream>

#include <stack>

#include <string>

using namespace std;

void Add(stack<float> & result)

{

float num1 = result.top();

result.pop();

float num2 = result.top();

result.pop();

float temp = num1 + num2;

result.push(temp);

}

void Subtraction(stack<float> & result)

{

float num1 = result.top();

result.pop();

float num2 = result.top();

result.pop();

float temp = num2 - num1;

result.push(temp);

}

void Divison(stack<float> & result)

{

float num1 = result.top();

result.pop();

float num2 = result.top();

result.pop();

float temp = num2 / num1;

result.push(temp);

}

void Multiplication(stack<float> & result)

{

float num1 = result.top();

result.pop();

float num2 = result.top();

result.pop();

float temp = num2 \* num1;

result.push(temp);

}

void Equals(stack<float> & result)

{

cout << result.top() << endl;

result.pop();

}

int main()

{

string problem;

cout << "Enter a math problem that you would like solved then finish it with the word \"stop\"" << endl;

cin >> problem;

stack<float> result;

while (problem != "stop")

{

if (problem == "+")

{

Add(result);

}

else if (problem == "-")

{

Subtraction(result);

}

else if (problem == "\*")

{

Multiplication(result);

}

else if (problem == "/")

{

Divison(result);

}

else if (problem == "=")

{

}

else

{

result.push(stof(problem));

}

cin >> problem;

}

cout << result.top() << endl;

return 0;

}

This lab consists of two parts:

* Postfix calculator
* Algorithm Analysis problems

# PostFix Calculation

In this lab you are going to write reverse postfix calculator using a stack. Postfix notation and is parenthesis-free as long as operator [arities](http://en.wikipedia.org/wiki/Arity) are fixed.

In postfix notation the operators *follow* their operands; for instance, to add 3 and 4, one would write "3 4 +" rather than "3 + 4". If there are multiple operations, the operator is given immediately after its second operand; so the expression written "3 − 4 + 5" in conventional notation would be written "3 4 − 5 +" in postfix: first subtract 4 from 3, then add 5 to that. An advantage of postfix is that it eliminates the need for parentheses that are required by infix. While "3 − 4 \* 5" can also be written "3 − (4 \* 5)", that means something quite different from "(3 − 4) \* 5". In postfix, the former could be written "3 4 5 \* −", which unambiguously means "3 (4 5 \*) −" which reduces to "3 20 −"; the latter could be written "3 4 - 5 \*" (or 5 3 4 - \*, if you wish to keep similar formatting), which unambiguously means "(3 4 -) 5 \*".

|  |  |  |
| --- | --- | --- |
| **Standard expression (infix notation)** | **Postfix Notation** | **Value** |
| 1 + 3 | 1 3 + | 4 |
| 10 / 5 | 10 5 / | 2 |
| 10 / (6 + 2 -3) | 10 6 2 + 3 - / | 2 |

**Don’t do everything in main. You should use a function (or more than one) to make the code more clean and organized.**

## Program Logic

**read in a string**

while the string is not “stop”

if the string is +

pop the last 2 values from the stack

push back their sum

else if the string is \*

pop the last 2 values from the stack and

push back their product

else if the string is –

pop the last 2 values from the stack and

push back the difference (second value – first value)

else if the string is /

pop the last 2 values from the stack

push back the quotient (second value / first value)

else if string is =

print the top of the stack

pop the stack

else if the string is a number

convert to a double and push it on the stack;   
**read the next string**

## Example Input/Output

**type in a postFix express or stop to stop**

**1 3 + =**

**4.00000**

**10 5 / =**

**2.00000**

**10 6 2 + 3 - / =**

**2.00000**

**1.1 2.2 \* =**

**2.42000**   
**stop**

Hints

* To include stack ADT from the standard template library   
  **#include <stack>**
* To declare a stack of Type   
  **stack<Type> nameOfStack;**
* The stack ADT supports pop(), push( value), top()
* To convert a string **word** into a double use   
  **double num = atof(word.c\_str());**
* For pretty output include **<iomanip>**
* And in the string put   
  **fixed<<showpoint<<setprecision(5)**

## Deliverables for Postfix Implementation

* Inside of a **Word document,** turn in your implementation (main.cpp) that shows your implementation
* Also, turn in a screenshot of an interaction using at least **three** different interactions with the program (in other words, three different postfix expressions, which the program uses to determine an output.)

# Algorithm Analysis Problems

Do these problems and turn them in in a Word document or PDF. Make sure to show your work.

|  |
| --- |
| Big O (Definition)  We say that g(n) is O(f(n)) - read as, “g of n is Big O of f of n” – when there exists some constants **k** and **n0** (pronounced as “n nought, or as n-sub-zero”) such that:  **g(n)** is bound by **k \* f(n)** for problem size **n >= n0.** |

1. Given the growth function, **g(n) = 2n2 + 4n +** 3, determine the Big O. Prove that your analysis is correct by finding a constant **k**, and a constant **n0** that satisfy the definition of Big O.  
   **g(n) = 2n^2 + 4n + 3**

**f(n) = 2n^2 + 4n^2 + 3n^2 = 7n^2**

**so k = 7, n nought = 1.**

**O(n^2)**

1. Given the growth function, **g(n) = 4n3 + n + 1**, determine the Big O. Prove that your analysis is correct by finding a constant **k**, and a constant **n0** that satisfy the definition of Big O.  
   **f(n) = 4n^3+n^3 +n^3 = 6n^3**

**So k = 6 and nought = 1**

**O(n^3)**

1. Given the following code segment, determine the Big O. Assume *n* is the number of elements in the array, myArray.

|  |
| --- |
| **void myMethod(int myArray[ ], int n)**  **{  int num1 = 15;  int num2 = 20;  int result = num1 + num2;**  **for(int i = 0; i < n; i++)   {**  **for(int j = 0; j < n; j++)  {**  **int temp = i \* j;  result += temp;**  **}**  **} }** |

**1+ n + n + n + n^2 + n^2 + n^2 + n^2 = 4n^2 + 3n + 1 = 4n^2 + n^2**

**O(n^2)**