UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science

COMS1000: Data and Data Structures
Project 1: Stack Calculator

Richard Klein

Due Date: 17:00 on Aug 8, 2013
Revision 0.1

1 Reverse Polish Notation (RPN)

Polish Notation was invented in the 1920's by Polish mathematician Jan Luasiewicz, who showed that by writing operators in front of their operands, instead of between them, brackets were made unnecessary. Although Polish Notation was developed for use in the fairly esoteric field of symbolic logic, Lukasiewicz noted that it could also be applied to arithmetic.

In the late 1950's the Australian philosopher and early computer scientist Charles L. Hamblin proposed a scheme in which the operators follow the operands (postfix operators), resulting in the Reverse Polish Notation. This has the advantage that the operators appear in the order required for computation. RPN was first used in the instruction language used by English Electric computers of the early 1960's. Engineers at the Hewlett-Packard company realised that RPN could be used to simplify the electronics of their calculators at the expense of a little learning by the user. The first "calculator" to use RPN was the HP9100A, which was introduced in 1968, although this machine is now regarded by many as the first desktop computer.

Once mastered, RPN allows complex expressions to be entered with relative ease and with a minimum of special symbols. In the 1960's that initial effort would have been regarded as a reasonable trade-off. For most calculator users of the time, the alternative was the error-prone practice of writing down intermediate results. Using RPN, it is possible to express an arbitrarily complex calculation without using brackets at all.

Many financial courses still recommend the use of RPN calculators. In fact, the HP12C is recommended in the Wits Actuarial Science courses!

Example RPN Notations:

Infix Notation	Postfix Notation
$(1+2)\times 3$	$3\,2\,1\,+\,\times$
$5\times(3+4)$	$5\,3\,4\,+\,\times$
$(3+5)\times(7-2)$	$5\,3\,+\,7\,2\,-$
$5+((1+2)\times 4)-3$	$5\,1\,2\,+\,4\,\times\,+\,3\,-$

Table 1: Reverse Polish Notation Examples

Here is an example from finance:

$$\frac{P\times(1+\frac{i}{12})^n}{\frac{i}{12}} \qquad \text{becomes} \qquad P\;1\;i\;12\div\times n\,\hat{}\,\times i\;12\div\div$$

2 Evaluation

RPN calculators evaluate the expressions using a stack. The following rules are used:

1. If the next symbol is an operand (a number), then put it on top of the stack (push).
2. If the next symbol is an operator, remove the correct number of operands from the top of the stack, perform the operation and place the result back on the top.

Suppose we have the following expression:

$$5 + ((1 + 2) \times 4) - 3$$

In RPN it is:

$$5\ 1\ 2 + 4 \times + 3 -$$

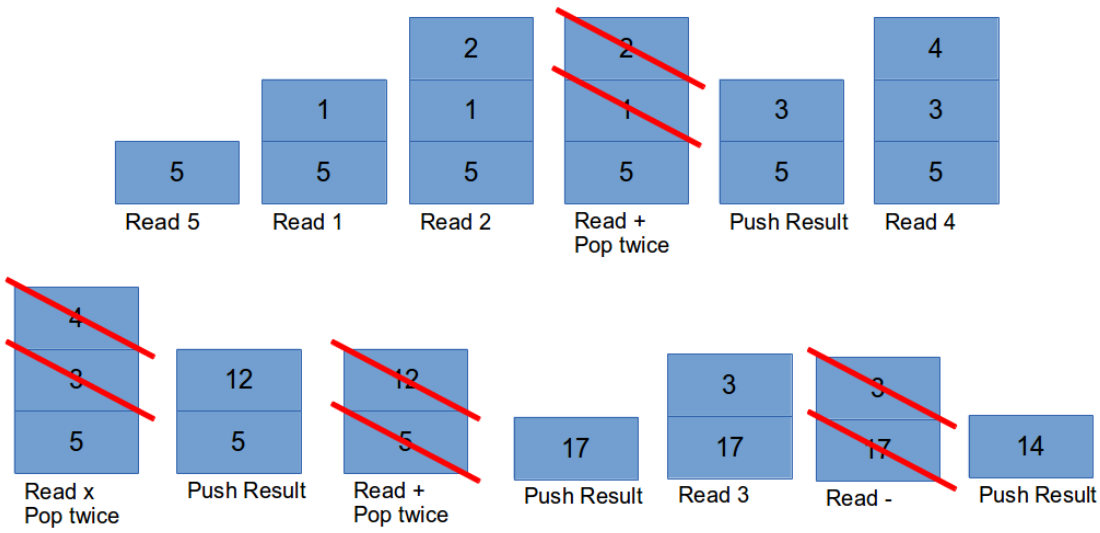

Figure 1: Evalutation of RPN using a Stack

3 Conversion from Infix to Postfix

The following algorithm converts from Infix to Postfix notation.

Algorithm 1 Convert from Infix to Postfix Notation

```
1:  function CONVERT(infix[])
2:      Initialise a stack
3:      for each character/token in the string do
4:          if it is a number then
5:              Append it to output string.
6:          else if it is left brace then
7:              Push it onto stack.
8:          else if it is an operator then
9:              if the stack is empty then
10:                 Push it onto the stack
11:             else
12:                 while the top of the stack has higher precedence do
13:                     Pop and append to output string
14:                 end while
15:                 Push the token/character to the stack
16:             end if
17:         else if it is right brace then
18:             while the stack is not empty and the top item isn't a left brace do
19:                 Pop from stack and append to output string.
20:             end while
21:             Finally pop out and disgard the left brace.
22:         end if
23:     end for
24:     If there is any input in the stack, pop and append each item to the output string.
25:     return The output string.
26: end function
```

4 Instructions

For this project you will need to implement an RPN calculator. You may assume that the length of the expressions will never be more than 255 characters. Use an array to implement your stack. You may hard code this number as a `#define`. You will need to implement a number of functions. We will provide a main method that will call your functions. You must implement each required function. Do not print to the terminal in your final output. Do not change the main function.

You may assume that all numbers in the input will be single digit numbers. Eg. `12+` is postfix notation for `1+2`.

In C the following functions are required:

```
IntStack* stack_i_init();
void stack_i_push(IntStack *s, int c);
int stack_i_pop(IntStack *s);
int stack_i_peek(IntStack *s);

CharStack *s stack_c_init();
void stack_c_push(CharStack *s, char c);
char stack_c_pop(CharStack *s);
char stack_c_peek(CharStack *s);
int calculate(char *postfix);
char* convert(char *infix);
```

In C++ the following functions are required:

```
void Stack<T>::Stack();
void stack<T>::push(Stack *s, T c);
T stack<T>::pop(Stack *s);
T stack<T>::peek(Stack *s);
int calculate(char *postfix);
char* convert(char *infix);
```

`calculate` should take in a character array in postfix format and evaluate the expression. `convert` should take in a character array in infix format and return the expression in postfix format.

4.1 C Files

We will provide

`main.c`	Contains our main function. You may not change this file.
`postfix.h`	Contains the function prototypes that are needed for the other .c files. You may not change this file.
`postfix.c`	Contains the function implementations. This is the only file you may change. You will submit this file.
`makefile`	Contains necessary make commands.

Both `main.c` and `postfix.c` should `#include "postfix.h"`

4.2 C++ Files

We will provide

`main.cpp`	Contains our main function. You may not change this file.
`postfix.h`	Contains the function and template prototypes. This is the only file you may change. You will submit this file.
`makefile`	Contains necessary make commands.

4.3 Compiling and Marking

We will copy your `main.c` or `postfix.h` file over a fresh copy with the other files. We will then compile the program by running: `make`
To run your program we will run: `./main`
Submissions that do not compile will get 0. You can submit to moodle multiple times until your submission is correct. Note that we will be doing plagiarism detection.

4.4 Error Detection

- You should always check that there are items on the stack to pop, before doing so.
- When a valid postfix expression has been evaluated, there should only be 1 item left on the stack. If there are more items left, it means that the expression malformed/invalid. In this case, return -1000.
- When a valid postix expression is evaluated, there should always be enough operands on the stack to perform the operation. If at any stage there aren't enough operands on the stack, then expression is malformed. In this case, return -2000.

4.5 Extra Credit (10%)

Implement the `calculate_extra` function, to handle numbers that contain an arbitrary number of digits. The input format changes so that there is always a space between operands and operators. You are only eligible for this extra credit if your `calculate` function works correctly.
Original format:

```
123*+
```

New Format:

```
12 3 *
```

4.6 Grading

Description	Marks
Correct Stack Functions	30
Evaluate Valid Postfix Expressions	30
Catch errors when there are too few operands	10
Catch errors when there are too few operators	10
Infix to Postfix conversion	20
Extra Credit	10
Total	110