

School of Computer Science and Software Engineering
Clayton Campus, Monash University

CSE1303 Part A
Semester II, 2001

Practical Session 2: Stacks



Aims of this Practical Session

1. To familiarise yourself with stacks and how to implement them.
2. To introduce Reverse Polish notation.

Project: To write a small calculator program which calculates integer arithmetic expressions involving the operators $+$, $*$, $-$, and $/$.

Background:

We will first assume that the expression will be written in Reverse Polish notation. Reverse Polish notation is useful as it does not require us to use brackets to solve the precedence problems normal "infix" notation has.

The following table illustrates how normal "infix" expressions can be written in Reverse Polish notation:

<i>Infix</i>	<i>Reverse Polish</i>
$a + b * c$	$a b c * +$
$a + (b * c)$	$a b c * +$
$(a + b) * c$	$a b + c *$
$a * b + c$	$a b * c +$
$a * (b + c)$	$a b c + *$
$(a + b) * (c - d)$	$a b + c d - *$
$(a + b) * (c - d) / (e + f)$	$a b + c d - * e f + /$

Note that the variables appear in the same order, and the operators ($+$, $*$, $-$, $/$) come after the operands (variables) they are operating on, and that no parentheses are needed for expressions in Reverse Polish notation.

In this prac we will use a stack to evaluate the Reverse Polish expressions. The idea is as follows: as each number is read it is pushed onto the stack; when the characters $+$, $-$, $*$ or $/$ are read, two numbers are popped off the stack, the operator is applied to them, and the result is pushed back onto the stack; when the character $=$ is read the number on top of stack is popped out and printed; otherwise an error message is printed.

For example the infix expression

$$(1 - 7) * (4 + 5)$$

is entered as

$$1 \ 7 \ - \ 4 \ 5 \ + \ * \ =$$

When **1** and **7** are read they are pushed onto the stack. Then after the character **-** is read, **1** and **7** are popped off and their difference **-6** is pushed onto the stack. Next **4** and **5** are pushed onto the stack, then after the character **+** is read, they are popped off and replaced by their sum **9**. Then after the character ***** is read **9** and **-6** are popped off the stack, and their product **-54** is pushed onto the stack. Finally, after the character **=** is read **-54** is popped off the stack and printed.

Preparation (3 marks if completed before class)

Before this prac:

1. Write what output the program should produce when given the following input.

- $3 \ 8 \ * \ 5 \ 4 \ * \ - \ =$

- $10 \ 3 \ / \ 3 \ 2 \ - \ 12 \ 5 \ / \ * \ + \ =$

⊕ **Remember to do integer division.**

2. Assume an integer is a string of digits that may begin with a negative sign. Write a C program which reads in character strings that do not contain blanks, newlines, or tabs, and prints out the character string together with a message indicating whether it is or is not an integer.

⊕ **For example for the following input:**

$$12 \ 3 \ + \ -8ha \ -98$$

the output would be:

```
12 Number
3 Number
+ Not a number
-8ha Not a number
-98 Number
```

⊕ **You may assume all strings have at most 4 characters.**

3. Modify the code given in lectures for a stack, so that instead of holding numbers of type **float** it holds numbers of type **int**.

Question 1: (1 mark)

Before you can write a program to evaluate expression you need to be able to interpret the input.

Extend the C program you wrote in the second part of the preparation so that the program prints out the character string together with a message, either indicating it is an integer, it is an operator, or it is an invalid string.



For example for the following input:

12 3 + -8ha -98

the output would be:

12 Number
3 Number
+ Operator
-8ha Invalid string
-98 Number

Question 2: (3 marks)

To check the code you wrote in the third part of the preparation write a C program which has a menu with the following options to manipulate a Stack:

- **Push**: which asks the user for an integer, and pushes the integer onto the stack.
- **Pop**: which removes the integer at the top of the stack, and prints it.
- **Print**: which prints all the integers in the stack and doesn't change the stack.
- **Size**: which prints the number of integers in the stack and doesn't change the stack.
- **Quit**: which allows the user to quit the menu and exit the program.

Question 3: (3 marks)

Using the program you wrote for question 1 and the code you wrote in the third part of the preparation, write a new C program which reads integer expression in Reverse Polish notation and evaluates them.

Test this program on the example input given in the first part of the preparation.

Advanced Questions: (2 marks)

So far our simple calculator reads expressions in Reverse Polish notation. In this question you will modify your calculator so that it can read normal “infix” notation.

Dijkstra’s shunting algorithm is a method of generating Reverse Polish notation from normal “infix” notation.

We say that infix multiplication and division have a higher precedence than addition and subtraction. The former bind more tightly to their operands than the latter.

The shunting algorithm uses a stack of characters and works as follows. When a **NUMBER** is read in, it is immediately printed. When an **OPERATOR** is read in, (let’s call it the **current Operator**), then either:

- the stack is empty and **current Operator** is pushed onto the stack,
- the **current Operator** has higher precedence than the **OPERATOR** on top of the stack, and the **current Operator** is pushed onto the stack, or
- the **current Operator** has lower, or equal, precedence than the **OPERATOR** on top of the stack, and in this case those **OPERATORS** on the stack with greater, or equal, precedence than the **current Operator** are popped off the stack (and printed) until either the stack is empty, or the **OPERATOR** on top of the stack has lower precedence than the **current Operator**. Then the **current Operator** is pushed onto the stack.

Finally, when all the input is exhausted all the **OPERATORS** remaining on the stack are popped off and printed.

Parentheses have still to be considered. They isolate subexpressions. The algorithm therefore treats the subexpression as a piece of sub-input. When an opening (is encountered, it is pushed onto the stack and hides everything beneath. The next **OPERATOR** which is encountered is pushed onto the stack. When a closing) is encountered, it is treated as the end of a subexpression and any operators of the subexpression are popped off the stack (and printed) - up to the opening (. Both parentheses are then discarded. The algorithm then continues as before.

Questions:

- Write a C program that reads integer infix expressions and prints out Reverse Polish expressions.
- Implement a calculator which accepts infix notation.