CS2002 Week 7 Practical

C Programming: Reverse Polish Calculator

9th March 2021

Matriculation Number: 200002548

Tutor: Marco Caminati

Contents

1 Introduction

The aim of this practical was to gain experience with programming C by creating and manipulating data structures. In this practical we should use our knowledge of C to implement a stack data structure using structs, and then create a calculator to evaluate mathematical instructions in reverse polish notation (RPN). Both the stack data structure and the calculator were then to be tested after implementation.

2 Design and Implementation

2.1 Stack Data Structure

First the stack data structure was implemented. The stack was chosen to be implemented using dynamic memory allocation. The stack struct itself was defined within `stack.h` and stores three variables. There is an `array` pointer that references an float array that holds all the data within the struct, as well as two integers `capacity` and `top` that store the total capacity of the stack (size of `array`) and the position of the topmost element of the stack in the `array`. A `MAX_CAPACITY` is also defined which sets the maximum size of the array and therefore the stack.

The stacks functions are defined within `Stack.c`. `newStack()` creates a dynamically allocated `Stack` using `malloc` and initalises its variables. The `capacity` is set to the `MAX_CAPACITY`, the `array` is dynamically allocated to a space of size `MAX_CAPACITY * sizeof(float)` bytes. The `top` variable is then initialised to `-1` since the stack is empty so there is no top value in the underlying `array` to point to. The function then returns a pointer to the position of the dynamically allocated stack within memory. This pointer is passed into every function of the stack, so that it can read and modify the stack by reference. `Stack_push()` takes in this reference to the stack and an element to be pushed onto it. If the stack is full, the element cannot be pushed on and so the function returns `false`, otherwise the element is pushed onto the top of the stack by adding it to the smallest unused index in the array and incrementing the `top` variable. The function then returns `true`, as the operation has been performed successfully. `Stack_pop()` also takes in a pointer to the stack, as well as another pointer to a float which will store the value popped from the stack. If the stack is empty it is impossible to pop off another element and so the function returns `false`. Otherwise the value is popped off the top of the stack and placed in the location of the `retval` pointer. The `top` variable is then decremented to reduce the size of the stack. `Stack_size()` and `Stack_isEmpty()` are descriptive from their implementation. `Stack_clear()` clears the stack by removing all elements from it to reduce it back to its original state. This is done by resetting the `top` variable to `-1`, this does not remove any of the data and instead simply resets the size of the stack. The last function `Stack_destroy()` works similarly to a constructor, it calls `Stack_clear()` to clear the stack, then frees the `array` and thereafter the `Stack` struct itself from memory. The `array` must be freed before the `Stack` to remove the possibility of memory leaks whereby the `array` pointer is deleted before the `array` itself has been freed from memory.

2.2 RPN Calculator

The calculator interface is defined within `Calculator.h`. In this file the struct `Calculator` is declared and holds the attributes of an RPN calculator. The `Stack` pointer holds the address of the stack whereby numerical data used when evaluating expressions are stored. It also includes a `char` pointer to point to the expression to be evaluated and a `float` to hold the `result` of the calculation. Within the header file also contains a definition for the maximum expression length (`MAX_EXPR_SIZE`) used for allocating a certain size of buffer to hold the expression.

The calculator is then implemented within `Calculator.c`. The file contains four functions that perform the `addition()`, `subtraction()`, `multiplication()` and `division()` binary operations. Within the specification there is no mention of error handling with division by zero, so I have included this within the `division()` function. If the second operand i.e. the denominator `op2` is equal to zero. Then the program sends `Bad expression` to `stdout` and the program is halted. The `performOperation()` function is where these operations are performed based on the current symbol being evaluated. This function pops two operands from the stack, performs and operation based on the current operand being evaluated, and pushes the result to the top of the stack. `isNum()` checks if a number is valid, it does this by checking if the first character is a digit or if the first character is a minus sign followed by a digit. Then `checkNotEnoughOperands` and `checkNotEnoughOperators` are used to check the stack size to validate if the expression is valid. If not then the program outputs `Bad expression` and exits. This is crucial for evaluating expressions since if there is still an operator to evaluate but less than two operators left in the stack then two few operands have been supplied. Moreover if there is more than one operand left in the stack and no operators left then two few operators have been supplied. In both these conditions the expression is invalid. Finally we reach the implementation of the Calculator functions declared in `Calculator.h`. `newCalculator()` takes in a `char` pointer to the expression string. Firstly a new calculator is created. Then the expression `expr` is tokenised between every space. Then the `reslut` is initialised to 0 and the `stack` created using `newStack()`. `destroyCalculator()` destroys the Calculator by clearing any stack data (and freeing the memory if you are using dynamic memory allocation). Finally the `evaluateExpression()` is where most of the work is done evaluating the expression. In this function the tokenised expression is iterated over. If the current token is a number then it is converted to a `float` and pushed onto the stack. Otherwise the token must be an operator so we first check if there is not enough operands, if this is false we perform the operation. Once each token in the expression has been iterated over we check if there is more than one operand left in the stack. If this is the case not enough operators where supplied in the expression. Otherwise, the one operand left in the stack is the result, therefore this is popped from the stack and returned.

The calculator itself is controlled from `CalculatorMain.c` the main entry point of the C program. In this file the expression is read in from standard input (`stdin`) using `scanf` whereby only the characters `.e0-9+-*/` are accepted. The expression is then sent into the `newCalculator()` function which returns a `Calculator` struct. The calculator can then be ran by executing `evaluateExpression()` and passing in the `Calculator` struct. The result is then sent to standard output (`stdout`) using `printf` and displayed to 2 d.p. as per the specification. Finally, the function `destroyCalculator` is called to free up the underlying `Stack` within the calculator.

3 Testing

3.1 Stack Data Structure

The stack was tested using the provided `TestStack.c` file. The tests were written during implementation to test each of the individual stack functions. The tests and their purpose are described below:

- `newStackIsNotNull()` - Checks that the Stack constructor returns a non-`NULL` pointer.
- `newStackSizeZero()` - Checks that the size of an empty stack is 0.
- `pushOneElement()` - Checks if an element has been pushed onto the stack.
- `pushMultipleElements()` - Checks if multiple elements can be pushed onto the stack, up to the max capacity.
- `popOneElement()` - Checks if an element can be popped from the stack and that the popped value is correct.
- `popMultipleElements()` - Checks if multiple elements can be popped from the stack and that all popped values are correct.
- `pushFullStack()` - Checks if you are unable to push onto a stack at maximum capacity.
- `popEmptyStack()` - Checks if you are unable to pop from an empty stack.
- `isStackEmpty()` - Checks if the stack is empty after a push and a pop.
- `hasStackCleared()` - Checks if the stack is empty after it has been cleared.

The results of these tests are shown below:

```
Stack Tests complete: 10 / 10 tests successful.
----------------
Process finished with exit code 0
```

Figure 1: TestStack.c output

3.2 RPN Calculator

The calculator was tested using the provided Stackscheck tests. A further twenty-two comprehensive tests were ran to validate the capabilities of the calculator for different inputs such as:

- floats in exponential notation
- floats in exponential notation with a negative exponent
- adding a number to 0 and vice versa.
- dividing by zero

- dividing by a negative
- dividing by a small decimal number
- infix expressions
- postfix expressions
- prefix expressions
- to many operands
- to many operators
- recurring numbers

The results of these provided and extra comprehensive tests are shown below:

```
* BUILD TEST - build-clean : pass
* BUILD TEST - calculator/comprehensive/build : pass
* COMPARISON TEST - calculator/comprehensive/prog-add-exp-non-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-add-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-add-neg-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-add-num-zero.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-add-zero-num.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-add-zero-zero.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-div-by-negative.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-div-by-small-num.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-div-by-zero.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-div-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-div-neg-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-infix-expr.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-multi-exp-non-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-multi-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-multi-neg-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-multiple-operands.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-multiple-operators.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-postfixed-expr.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-prefixed-expr.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-recurring-rounded.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-sub-exp.out : pass
* COMPARISON TEST - calculator/comprehensive/prog-sub-neg-exp.out : pass
```

Figure 2: Stacscheck comprehensive tests output

```
* BUILD TEST - calculator/provided/build : pass
* COMPARISON TEST - calculator/provided/prog-0.out : pass
* COMPARISON TEST - calculator/provided/prog-1.out : pass
* COMPARISON TEST - calculator/provided/prog-2.out : pass
* COMPARISON TEST - calculator/provided/prog-3.out : pass
* COMPARISON TEST - calculator/provided/prog-bad-1.out : pass
* COMPARISON TEST - calculator/provided/prog-bad-2.out : pass
* COMPARISON TEST - calculator/provided/prog-bad-3.out : pass
* COMPARISON TEST - calculator/provided/prog-bad-4.out : pass
* COMPARISON TEST - calculator/provided/prog-float-1.out : pass
* COMPARISON TEST - calculator/provided/prog-float-2.out : pass
* COMPARISON TEST - calculator/provided/prog-longer-1.out : pass
* COMPARISON TEST - calculator/provided/prog-negative-1.out : pass
* COMPARISON TEST - calculator/provided/prog-negative-2.out : pass
38 out of 38 tests passed
```

Figure 3: Stacscheck provided tests output

From testing it can be seen that both the stack and the reverse polish notation (RPN) calculator perform correctly under a wide range of inputs.

4 Conclusion

During this practical I have greatly increased my understanding of C programming, especially how to program in C with an object-oriented style. I have also gained valuable experience into working with dynamic memory (using `malloc()` and `free()`) as well as dealing with various issues related to pointers. Furthermore the practical allowed me to explore the use of stacks and how you can use them to evaluate RPN expressions.

References

[1] Kelly Gendron, *How do you allow spaces to be entered using scanf?*, 08-08-2009, accessed 09-03-2021,
`https://stackoverflow.com/a/1247993/7711225`

[2] Mysticial, *How to convert string to float?*, 31-10-2011, accessed 09-03-2021,
`https://stackoverflow.com/a/7951034/7711225`