A Calculator in C using stack

Mo Wang

Spring 2026

Introduction

This report delves into the implementation of a calculator in C using reversed Polish notation with a stack. The purpose of the report is to learn practical application of a stack data structure as well as to compare differences between static and dynamic stack, by implementing a simple terminal-based calculator supporting reversed Polish notation.

Different stack implementations

Because a stack is an array that supports only push and pop operations, it can be implemented with either with a fixed capacity size or a dynamically growing capacity size. A stack with a fixed capacity size is called a static stack, while the other one that can grow and shrink is called a dynamic stack. Both types of stacks are discussed below.

Static stacks

A static stack has a fixed maximum capacity because heap allocation of the stack array requires a pre-defined size to fit within the available memory blocks. The stack has a stack pointer (or index) named `top` that points to the element immediately after the topmost element, effectively pointing to the next free slot, which follows the stack pointer convention. This convention becomes especially clear in an empty stack, where the `top` index points to the first element at index zero.
<MINTED>

A static stack has two different operations, push and pop. The push operation overlays the given element above the topmost element of the stack, which means that the element is written using stack index, while incrementing it. On the other hand, pop operation removes the topmost element, revealing the element beneath it, by decrementing the stack index and returning the element at its position. The operations are shown below in functions `static_stack_push` and `static_stack_push`.

<MINTED>

However, attempting to push an element beyond its capacity size or pop an element when empty results in accessing out-of-bounds memory and causes undefined behavior. To mitigate this issue, the stack pointer `top` will be checked before performing any operation. The push operation fails when stack pointer reaches the array capacity while pop operation fails when stack pointer is zero.

For reporting operation success, the push function can return `true` if successful, and the pop function can return a `Result` struct containing a boolean success flag and an integer value for the popped element. By catching the success flag, the caller function can determine the status of the operation.

<MINTED>

Dynamic stacks

Implementing a dynamic stack that allow growth and shrink is similar compared to a static stack, especially in terms of struct variables and creating as well as destroying the stack. The key difference relies in array resizing by reallocating memory data from one size to another one, as well as keeping the current array capacity in variable `size`, which is kept fixed in static stacks.

<MINTED>

During push operation in a full array, the stack allocates another twice as large array and copies all the existing elements into the new one using the helper function `array_resize_copy`. This resizing approach ensures constant amortized time complexity $O(1)$ since each operation involves on average one write and one copy operation.

<MINTED>

Shrinking the array, on the other hand, is also important to free memory when the number of elements decreases. However, reducing array size whenever it falls below half its capacity can lead to array thrashing during repeatedly growth and shrinks around the same size. To prevent this, the array is only shrunk when the number of elements drops strictly below one fourth of the current capacity. Additionally, the array will shrink below certain minimum capacity size, in this case 4, in order to prevent frequent grow-shrink oscillation of tiny stacks.

<MINTED>

Calculator implementation

With the stack implementations, building a reverse Polish notation terminal calculator becomes straightforward. Each token is passed into the CLI interface delimited by newlines and will be checked for number or operator.

If the token is a number, it will be pushed onto the stack, otherwise if it's an operator it's used to pop 2 operands and push the calculated result if the given token is an operator. After processing the current token, the process is repeated over again, until an empty token is passed and the final calculated result will be produced. The structural code can be visualized as below (A dynamic stack is used in the example below):
<MINTED>

Terminal user input are read from standard input `stdin` and written into a char buffer `buffer` using function `fgets`. When enter key is pressed, a newline character is fed into standard input and `fgets` stop reading while storing the user input line including newline and null character to the buffer.
<MINTED>

Numbers can be distinguished from operators using `strcmp` and `strchr` from `<string.h>` library. An empty line is detected using `strcmp(buffer, "n") == 0`. Single-character operators are detected using `strchr("+-*/", buffer[0])` together with the condition `buffer[1] == 'n'` to ensure that the token contains only the operator and a newline.
<MINTED>

When an operator is detected in a buffer line, two values should be popped out from the stack and calculated as well as pushing the value back to stack. The calculating operation is accomplished by `apply_op` that takes the stack and an operator and produce the correct calculation. Operation success or failure is reported by the output boolean flag, `true` for successed operation, while `false` for failed operation. The operation can be failed if the stack state is invalid or division by zero attempt is detected.
<MINTED>

The caller function function recieves the status flag of the operation in the application loop and terminate the application program when invalid state error or division by zero attempt occurs. The main application loop is exited while heap resources are deallocated in the process. Using this approach, any invalid user input or invalid program state invariant can be caught and resolved by terminating the program.
<MINTED>

Hints: - How tokens are read from stdin. - How you distinguish numbers from operators. - How each operator works (pop two values, compute, push result). - How division by zero is handled. - How the calculator behaves when the stack is in an invalid state.

Experiments and testing

Hints: (MOST IMPORTANT) This is where you show that you "experimented with both versions of the stack." Include: Example expressions you

tested. Differences in behavior between static and dynamic stacks. What happens when the static stack overflows. What happens when you pop from an empty stack. The evaluation of the example expression: