HP35 Calculator Assignment

Serban Alin-Cristian

September 2, 2023

1 Introduction

In this assignment I recreated the functionality of a HP35 calculator , using Reverse Polish Notation, but most importantly implemented a stack with both static and dynamic sizes. The language I used for this was C because I like it and needed an excuse to get better at it. I then proceeded to test them, then run several benchmarks to get accurate(ish) time readings to show the difference in performance between static and dynamic stacks.

2 Code

2.1 Structure of the project

I spread out everything in 5 files,

- staticstack.cs (which holds the code for, well, a static stack)
- dynamicstack.cs (same as above almost)
- calculator.cs (holds all the code regarding the HP35 functionality and also calls the push/pop functions from the stacks)
- misc.cs (code for the item object and nothing else that fit)
- main.cs (benchmarking and actually running the calculator)

2.2 Implementing the stack

As in the assignment instructions, I've made both a static and dynamic stack. They are both basically just an int array that only gets modified with a .push() and .pop() function. I used a pointer integer to keep track of where the stack's top was. As expected, in a static stack, I declare the size before running the calculator and if it exceeds the maximum size it throws a stackoverflow error. This is where the dynamic stack comes in.

```
public void push(int x)
{
    pointer++;
    thestack[pointer] = x;
}
public int pop()
{
    int x = thestack[pointer];
    thestack[pointer] = 0;
    pointer--;
    return x;
}
...
```

For the dynamic stack, I added extra functionality to .push(), where if the pointer was equal to the stack's size, I'd create a new array with double the size, and copy everything on there. This poses a slight issue - what if we have 1024 elements and an array of size 1024, then push an element, raising the stack size to 2048, but we immediately decrease and never use over 1024 elements, therefore wasting all that memory?

As such, I've created a "counter" integer that starts at 10, and counts down for each push/pop performed under half of the current stack size. So, if we had 33 elements (and a stack size of 64), and stayed over 33 elements, it'd never decrease, HOWEVER, if we decreased back to 32 elements and under, and pushed and popped 10 times without going over 32 elements, the stack size would decrease back to 32 using the same method to increase it - making a new array of the desired size and copying everything over.

```
int counter = 10;
...
...
if (counter != 0 && pointer+1 < thestack.Length/2)
            counter--;
        else
            counter = 10;
        if (counter <= 0)
        {   int[] newstack = new int[thestack.Length/2];
            for (int i = 0; i < thestack.Length/2; i++)
                newstack[i] = thestack[i];
            thestack = newstack;
        }
        ...
```

2.3 Implementing the calculator

This was by far the easiest part. In misc.cs I've basically taken the Item class verbatim, holding a value and itemType type. Using this I programmed the calculator.cs file, which reads the value of a Item array (the expression fed to the calculator), and runs a foreach loop to go through each expression command, adding values to the stack, and for operators popping twice and pushing the result. Simple enough.

```
foreach (Item item in expr)
        {
            switch (item.type)
            {
                case itemType.VAL:
                {
                    stack.push(item.value);
                    break;
                }
                case itemType.ADD:
                {
                    int b = stack.pop();
                    int a = stack.pop();
                    stack.push(a+b);
                    break;
                }
                ...
```

Simple enough in its implementation, because I encountered various difficulties coding it, because of two reasons: stupidity and laziness.

The one caused by stupidity was, in main.cs, calling the instance of Calculator....Calculator. Like, Calculator Calculator = new Calculator(expression). I will not elaborate further on this.

The one caused by laziness was and still is in the code. To switch between dynamic and static stack you need to commend and uncomment 2 lines, declaring the property "stack" inside Calculator to be either of DynamicStack or StaticStack type, and in the constructor too. This could have been solved in various ways including interfaces or an abstract "stack" class, but for the purposes of this assignment I didn't deem it important enough. The target was to implement the stacks, and see the differences in performance, not to implement the calculator efficiently or conveniently.

2.4 Testing

After finishing the main code, I ran some tests using both types of stack. The main difficulties I encountered were finding out that C requires you to have every element of an array to be NOT null when equaling it to another array (like in a constructor, i.e. this.expr = expr), and the relative weirdness of accessing variables from inside an object. I needed to access the stack length from my Calculator object, but couldn't figure out at the time how to do it, so I just had a function inside both stacks that printed out it's size. It worked well enough. This is bad coding practice but it sufficed for what I needed it to do, and for the scope of this assignment.

In the end, I managed to get the right result for "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 + * + * + * + * + * + * + * +", pushing and popping 1000 items, and for all the operation types. Also I checked that the stack size did indeed modify with dynamic stacks and that it properly reduced when the counter ticked low enough.

2.5 Benchmarking

As C does not have a convenient command to automatically benchmark stuff, I made my own. I feed it two integers, repeat and bench, and it runs the calculator for repeat amount of times, by filling the stack with half values of 1, and the latter half with + signs (minus 1 of course to get a pretty looking result on the calculator).

Alongside this, the Stopwatch class was very useful to actually time the speed of the program. Basically starting a stopwatch before running the calculator, then ending it, printing the result, and resetting. A cleaner solution would've been getting the average of the last 50 results out of 100 repeats, but they were close enough that it wasn't neccesary.

```
...
stopwatch.Start();
Calculator calc = new Calculator(expr, bench*2);
calc.run();
stopwatch.Stop();
Console.WriteLine(stopwatch.Elapsed);
stopwatch.Reset();
...
```

3 Results

3.1 Benchmark Numbers

For the StaticStack and DynamicStack, running a benchmark of 1000 repeats and 5000 elements pushed, then popped...

Type	Time
Static	0.0001800s
Dynamic	0.0003100s

Table 1: A table of average stack speeds.

We can see that the DynamicStack is significantly slower than the StaticStack. This is expected, as DynamicStack has to waste time moving data from one array into the other, which adds up the more times it is performed. StaticStack has no such issues and can simply use the same array all over, provided it has enough space for everything you need to do with it. The downside of using a static stack is that, even knowing how much space you need to use to do all you need, it will waste memory as it empties out (if you used 2048 elements, had a max size of 2048, but then decreased back to 1 element, all that space would be wasted!)

Meanwhile, a dynamic stack can both increase in size as needed, and also decrease if it's not needed anymore! This makes it less time efficient than a static array, but arguably much more memory efficient.

3.2 Conclusions and takeaways

In the end, I've achieved every goal this assignment asked; programmed a HP35 calculator in reverse polish notation, implemented a stack in both dynamic and static flavors, tested them both and showed the differences between them, realized the advantages and disadvantages, and personally, learned a good bit about C along the way here.