A Calculator

in C

Algorithms and data structures ID1021

Johan Montelius

Spring 2026

Introduction

In this assignment you should implement a calculator that can calculate mathematical expressions described using *reverse Polish notation*. We do this in order to see how a *stack* can be used. In all your programming courses so far you have been working using a stack but you might not have used one explicitly like we shall do now. You have probably never heard of the reverse Polish notation but everything will be clear in a minute (or hour).

The HP-35 calculator and reverse Polish notation and the stack

Reverse Polish notation is simply writing mathematical expressions with the operand last. Instead of writing $5 + 3$ we write $5\ 3\ +$. This sounds weird but it has its advantages and the first programmable computer, Z3, used reveres Polish notation when expressing mathematical formulas (start googling now). If you took this class in 1972 rather than 2022 you would probably be the proud owner of a HP-35 pocket calculator that also used this form of entering expressions.

The good thing with reversed Polish notation is that you can do a away with parenthesis. The expression $(4 + 5) * 6$ is simply written $4\ 5\ +\ 6\ *$. The biggest advantage is that we have a very simple way of calculating the result, all we need is a *stack*.

A stack is data structure that allows two basic operations: *push* and *pop*. An item can be pushed on the stack and is then at top of the stack. A pop operation will remove, and return, the item at the top of the stack, if the stack is not empty. The item below the removed item is then at the top of the stack. We can have other operations: check if the stack is empty, peek

at a value some steps below the top of the stack etc but the two operations that changes the state is *push* and *pop*.

It turns out that the reversed notation and the stack are made for each other. If we have the expression 3 4 + then we simply push 3 on the stack, push 4 on the stack and then do the addition by popping two items from the stack and push the result back.

Take a pen and paper and go through the steps to calculate 3 4 + 2 4 + ∗, if you end up with 42 on the stack you got the point.

The calculator itself is very simple in its design. It will keep a stack that is used to store the numbers we enter and the results computed. It will read input from the terminal and either push the values on the stack or pop the required arguments, perform the calculation and push the result.

The tricky part is implementing the stack - don't worry, you will have it up and running in about an hour.

Implementing the stack

You will implement the stack in two versions, one static and one dynamic.

a static stack

The first implementation will be a fixed sized stack where the size is given when the stack is created. The stack should allocate an array of this size and keep track of a *stack pointer* (an index). For simplicity we implement a stack that can only hold integers.

The two methods `push` and `pop` are fairly simple to implement and the only thing you need to keep track of is the stack pointer and make sure that you do not push items outside of the array.

Questions you need to consider:

- Does the pointer point to the location above the top of the stack or does it point to the top of the stack?
- What is the value of the pointer when the stack is empty?
- What should you do when a program tries to push a value on a full stack (stack overflow)?
- What should happen when someone pops an item from an empty stack?

Some skeleton code to get you starting:

```
#include <stdlib.h>
#include <stdio.h>
```

```
typedef struct stack {
  int top;
  int size;
  int *array;
} stack;

stack *new_stack(int size) {
  int *array = (int*)malloc(size*sizeof(int));
  stack *stk = (stack*)malloc(sizeof(stack));
  :
  return stk;
}

void push(stack *stk, int val) {
  :
}

int pop(stack *stk) {
  :
}

int main() {

  stack *stk = new_stack(4);

  push(stk, 32);
  push(stk, 33);
  push(stk, 34);

  printf("pop : %d\n", pop(stk));
  printf("pop : %d\n", pop(stk));
  printf("pop : %d\n", pop(stk));
}
```

Note that we have to provide the size of the array when we create a stack i.e. wee need to know beforehand how many items we might want to push. Try to give a too small value and see what happens.

a dynamic stack

Slightly more complex is to handle a stack that can grow as we add more items. In a push operation, that would generate a stack overflow using the static stack, we simply extend the size of the stack by allocating a new

larger array and copy the items from the original array to the new array. One question is how much larger the new stack should be, should we increase by only one item (no), a fixed amount (maybe) or something else?

```
void push(stack *stk, int val) {
  if (stk->top == stk->size) {
    int size =
    stk->size =
    int *copy =
    for (int i = 0; i < ...; i++) {
      copy[i] =
    }
    free(stk->array);
    stk->array = copy;
  }
  :
}
```

Creating a larger stack should be only a few lines of code but how do you do if you should also be able to shrink the stack? Assume that you extend the stack from size 8 to 16 but then pop a few items, you then decrease the size to 8 again. Yet you do not want extend to 16 and then immediately decrease to 8 and then maybe immediately increase to 16. There should be some mechanism in the system that only decreases the size of the stack after a while.

You might want to keep a smallest possible size of your stack. There might not be any point in having stacks smaller than for example four items.

an empty stack

There is always the question what should be done if someone tries to perform an operation that can not be done; what should we do if we try to pop an item from an empty stack? One solution is to be happy but return a value that somehow signals that the stack is empty. This often zero, a negative value or a *null reference*. If we choose this strategy it is of course important that the user does not add these items to the stack since it would then be impossible to determine if the stack is empty or it just happened to be a value on the stack.

Another approach is to raise an error or exception. How this is done is very language dependent; some languages like Java has good support for this while other languages, like C, lack support. In this course we can take the simple way out and either return a known value or simply crash.

does it work

You experiment using the two versions of the stack in order to see that things work.

```
int main() {
  stack *stk = stack(4);

  int n = 10;
  for(int i = 0; i < n; i++) {
    push(stk, i+30);
  }

  for(int i = 0; i < stk->top; i++) {
    printf("stack[%d] : %d\n", i, stk->array[i]);
  }

  int val = pop(stk);
  while(val != 0) {  // assuming 0 is returned when the stack is empty
    printf("pop : %d\n", val);
    val = pop(stk);
  }
}
```

The Calculator

Once we have a stack we can implement a calculator. We will not implement a fancy graphical interface, we will simply interact using the terminal. This is some skeleton code that should get you started:

```
int main() {

  stack *stk = stack();

  printf("HP-35 pocket calculator\n");

  int n = 10;
  char *buffer = malloc(n);

  bool run = true;

  while(run) {
    printf(" > ");
    fgets(buffer, n, stdin);
```

```
    if (strcmp(buffer, "\n") == 0) {
      run = false;
    } else if (strcmp(buffer, "+\n") == 0) {
      int a = pop(stk);
      int b = pop(stk);
      push(stk, a+b);
    } else
      :
    } else {
      int val = atoi(buffer);
      push(stk, val);
    }
  }
  printf("the result is: %d\n\n", pop(stk));
  printf("I love reversed polish notation, don't you?\n");
}
```

What is the result of:

```
4 2 3 * 4 + 4 * + 2 -
```

... so you know the answer, do you know the question?