# A Calculator in C

Ying Pei Lin

Fall 2024

## Static stack

To implement a static stack in C, we can use an array to store the elements and an integer to record the current size of the stack. The following code shows how I initialize the stack.

```
stack *new_stack(int size) {
  int *array = (int*)malloc(size*sizeof(int));
  stack *stk = (stack*)malloc(sizeof(stack));
  stk->top = 0;
  stk->size = size;
  stk->array = array;
  return stk;
}
```

The top integer points to the next avaialble index in the stack. For expample, if the top is at index 3, this means that the stack has 3 elements stored at index 0, 1, and 2. When the stack is empty, the top equals to 0.

Whenever we need to push an element, we store the element at index equal to the top and then increase the top by 1. When we need to pop an elementm, we decrease the top first(because the top element is at index `top-1`) and then return the element at index `top-1`.

The following code shows how to push and pop elements from the stack.

```
void push(stack *stk, int val) {
  if(stk->top >= (stk->size)) {
    printf("Overflow ");
  } else {
    stk->array[stk->top] = val;
    stk->top++;
  }
}

int pop(stack *stk) {
```

```
  if(stk->top == 0) {
    printf("Underflow ");
    return INT_MIN;
  } else {
    stk->top--;
    return stk->array[stk->top];
  }
}
```

We check for overflow when pushing an element and check for underflow when popping an element. Since C doesn't support excpetion handling, we need to check the stack size before pushing and poping. If the stack is empty, we return `INT_MIN` to indicate that the stack is empty and if the stack is full, we just need to print out the error message.

## Dynamic stack

To implement a dynamic stack, we need to resize the stack when it's full. I made this modification in the push function. First, I check if the stack is full. If it is, then create a new array with the double size and copy the data from the old one to the new one. At the end, free the old array and set old array to the new one. By adding the following code to the push function, we can implement the dynamic stack.

```
if (stk->top == stk->size) {
  int size = stk->size * 2;
  int *copy = (int*)malloc(size*sizeof(int));
  stk->size = size;
  for (int i = 0; i < stk->top; i++) {
    copy[i] = stk->array[i];
  }
  free(stk->array);
  stk->array = copy;
}
```

Unless there is a huge amount of unused memory, I decide not to shrink the stack size because resizing and copying the data is a costly operation. For this case, I just let the stack grow.

If we want to shrink the stack size, we can modify the push function to check if there is a lot of unused memory. If the unused memory is more than a certain threshold (e.g. 75%), then we can shrink the stack size by half. By adding the following code to the push function, we can shrink the stack size, which is similar we did in the push function.

```
if (stk->top < stk->size / 4) {
  int size = stk->size / 2;
  int *copy = (int*)malloc(size*sizeof(int));
  stk->size = size;
  for (int i = 0; i < stk->top; i++) {
    copy[i] = stk->array[i];
  }
  free(stk->array);
  stk->array = copy;
}
```

## The Calculator

The HP35 calculator uses reverse Polish notation to perform the calculation. The user should input one number or a operator at a time following the sequence of reverse Polish notation. If the input is a number then push it to the stack. If it's an operator, then pop two numbers from the stack, and perform the operation. If the input is an empty line, then the program will stop. Following is a part of the code.

```
while(run) {
  printf(" > ");
  getline(&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, b+a);
  } else if(strcmp(buffer, "-\n") == 0) {
    int a = pop(stk);
    int b = pop(stk);
    push(stk, b-a);
  } else if(strcmp(buffer, "*\n") == 0) {
    int a = pop(stk);
    int b = pop(stk);
    push(stk, b*a);
  } else if(strcmp(buffer, "/\n") == 0) {
    int a = pop(stk);
    int b = pop(stk);
    push(stk, b/a);
  } else {
    int val = atoi(buffer);
    push(stk, val);
```

```
  }
}
```

For example, if the input is `4 2 3 * 4 + 4 * + 2 -`, then the stack will look like the following table.

| **Input** | **Stack (Top marked with *)** |
|---|---|
| 4 | 4* |
| 2 | 4 2* |
| 3 | 4 2 3* |
| * | 4 6* |
| 4 | 4 6 4* |
| + | 4 10* |
| 4 | 4 10 4* |
| * | 4 40* |
| + | 44* |
| 2 | 44 2* |
| - | 42* |

Table 1: HP-35 Calculator Steps (Top element marked)

## Conclusion

In this report, I tried to implement static, dynamic stack and a HP35 calculator in C. Because C doesn't support exception handling, so what we can do is to return a certain value to indicate the error, print out the error message or simply exit the program. The dynamic stack can grow but doesn't shrink unless we add the code to shrink the stack size. The HP35 calculator uses reverse Polish notation to perform the calculation. The advantage of using reverse Polish notation is that we don't need to worry about the order of the operation, stuff like processing the user input, also the stack make it easier to calculate the result. When the user input an operator, we just need to pop two numbers from the stack and perform the operation.