



# UNIVERSIDADE DE ÉVORA

## RPN - Reverse Polish Notation Calculadora em C

André Rato nº45517  
José Alexandre nº45223

Abril e Maio  
Ano Letivo 2019/2020

Arquitetura de Sistemas e Computadores I

Prof. Miguel Barão

# Índice

1	Introdução	3
2	Pilha (Stack)	4
3	Funções de acesso direto à stack	4
4	Funções de acesso não direto à stack	4
5	Função Main	6

# 1 Introdução

A notação polaca inversa (**RPN - *Reverse Polish Notation***), também conhecida como notação pós-fixada, é uma notação onde os operandos são introduzidos primeiro, seguidos dos operadores. Enquanto que na notação infixa, o operador é colocado no meio dos operandos, como em  $3 + 4$ , na notação pósfixa o operador surge no final, como em  $3\ 4\ +$ . Uma calculadora em notação polaca funciona como uma pilha. Os operandos são colocados na pilha e os operadores são aplicados sobre os operandos que estão no topo da pilha, substituindo-os pelos resultado da operação.

Para esta trabalho é necessário que a calculadora criada suporte os seguintes operadores:

Operadores	Descrição
+	Adição (operador binário)
-	Subtracção (operador binário)
*	Multiplicação (operador binário)
/	Divisão (operador binário)
neg	Calcula o simétrico (operador unário)
swap	Troca a opsição dos dois operandos do topo da pilha
dup	Duplica (clone) o operando do topo da pilha
drop	Elimina o operando do topo da pilha
clear	Limpa toda a pilha (fica vazia)
off	Desliga a calculadora (termina o programa)

Esta calculadora foi desenvolvida na linguagem C. É também de notar que a calculadora deverá ler e executar operações de duas formas:

Stack: (empty) -> 2 Stack: 2 -> 3 Stack: 2 3 -> swap Stack: 3 2	Stack: (empty) -> 2 3 swap Stack: 3 2
---	--

Figura 1: Métodos de introdução da *input* do utilizador

Para que possam ser usados os dois métodos descritos acima, a leitura e análise da *input* do utilizador é feita caractere a caractere.

## 2 Pilha (Stack)

Tal como é sugerido no enunciação do trabalho, é usada uma estrutura de dados (**struct**) para representar a pilha. Deste modo, temos uma **struct stack** com dois membros:

- um **array** de inteiros de tamanho 10 que guarda os valores da pilha;
- um inteiro **sp** (*stack pointer*) que indica quantos elementos existem no **array** (se o *stack pointer* for 0 não existem elementos no **array**).

## 3 Funções de acesso direto à stack

- **void push(struct stack \*stack, int number)**: função responsável por colocar valores na **stack**. Recebe como argumentos a **stack** e **number** (valor a ser colocado na pilha). Se **sp**  $\geq$  10, a função apresenta a mensagem "Invalid input. Out of bounds. Type 'help' for available commands". Se **sp**  $<$  10 a função coloca o valor de **number** na posição **sp** do **array**, e incrementa **sp** de modo a deixar a pilha pronta para receber outro **number**.
- **int pop(struct stack \*stack)**: função responsável por retirar o valor do topo da **stack**. Recebe a **stack** como argumento e retorna o valor que se encontra na posição **sp** do **array** (**stack->array[stack->sp]**) após a decrementação de **sp**, decrementação esta que é efetuada de modo a obter a posição do valor presente no topo da pilha.
- **void print(struct stack \*stack)**: função responsável por dar *print* da **stack**. Recebe a **stack** como argumento e dá *print* dos elementos do **array** por ordem crescente de *index* (desde 0 até ao **number\_of\_elements** (**sp**)), utilizando um ciclo **while**. Se **number\_of\_elements** = 0, significa que ainda não foram adicionados elementos à pilha, ou seja, ela está vazia. Quando isto acontece, a função dá *print* da mensagem "(empty)".

## 4 Funções de acesso não direto à stack

- **void some(struct stack \*stack)**: função responsável pela soma dos dois operandos do topo da **stack** e que recebe a mesma como argumento. Para que a função realize a soma, é necessário que **sp**  $>$  1. Caso isto aconteça, a função retira o operando do topo da pilha com recurso à função **pop** e guarda-o em **arg1**. Depois, repete o processo e guarda-o em **arg2**. Por fim, e com auxílio da função **push**, coloca o valor de **arg2** + **arg1** no topo da pilha. Caso não aconteça, a função mostra a mensagem "Not enough values in the stack", sem realizar nenhuma modificação à pilha.
- **void subtraction(struct stack \*stack)**: função responsável pela diferença entre os dois operandos do topo da pilha e que recebe a **stack** como argumento. Verifica inicialmente se **sp**  $>$  1. Se a condição for verdadeira, a função retira os dois valores do topo da pilha utilizando o mesmo método da função acima descrita (a função **some**) e, utilizando a função **push**, coloca o valor de **arg2** - **arg1** no topo da

`stack`. Se a condição for falsa, a função mostra a mensagem "Not enough values in the stack", sem realizar nenhuma modificação à mesma.

- `void product(struct stack *stack)`: função responsável pelo produto dos dois operandos do topo da pilha. Esta função recebe a `stack` como argumento e, se `sp > 1`, retira os dois valores do topo da mesma para as variáveis locais `arg1` e `arg2`, respetivamente, utilizando a função `pop`. Após isto, e através da função `push`, a função coloca na `stack` o valor de `arg2 * arg1`. Se não se verificar a condição, é apresentada a mensagem "Not enough values in the stack".
- `void quotient(struct stack *stack)`: função responsável pela divisão dos dois operandos do topo da pilha. Esta função recebe a `stack` como argumento e, caso `sp > 1`, retira os dois valores do topo da mesma para as variáveis locais `arg1` e `arg2`, respetivamente, recorrendo à função `pop`. Depois, verifica se `arg1 = 0`. Se a condição for verdadeira apresenta uma mensagem de erro e volta a colocar na pilha os valores guardados nas variáveis locais pela ordem com que estavam na pilha; se for falsa, coloca o valor de `arg2 / arg1` no topo da pilha. Se a condição inicial (`sp > 1`) não for verdadeira, a função dá *print* da mensagem "Not enough values in the stack".
- `void neg(struct stack *stack)`: função responsável pela alteração do sinal do operando do topo da pilha, recebendo a mesma como argumento (`struct stack *stack`). Se `sp > 0`, esta função retira o último operando da pilha e guarda-o na variável inteira `arg` utilizando a função `pop`. Depois disso, e com recurso à função `push`, coloca o valor simétrico de `arg` (`-arg`) no topo da pilha. Caso não se verifique a condição, a função mostra "Not enough values in the stack".
- `void swap(struct stack *stack)`: função responsável pela troca dos dois operandos do topo da `stack` e que recebe a mesma como argumento. Esta função retira os dois operandos do topo da pilha do mesmo modo que a função `some` (utilizando a função `push`), caso `sp > 1`. Depois volta a introduzi-los na `stack` de modo a que o primeiro valor retirado seja o primeiro a ser colocado, sendo assim invertida a ordem dos mesmos. Caso a condição seja falsa, é apresentada a mensagem "Not enough values in the stack".
- `void clear(struct stack *stack)`: função responsável pela limpeza da pilha recebendo a `stack` como argumento. Para executar a limpeza da `stack`, a função coloca o *stack pointer* (`sp`) a 0 (zero elementos presentes na pilha). Deste modo, a `stack` não fica literalmente vazia, mas deixa-a pronta a receber novos valores sem serem utilizados os valores que já lá estavam guardados.
- `void dup(struct stack *stack)`: função responsável por clonar o operando do topo da `stack`. Esta função recebe a `stack` como argumento e, se `sp > 0`, retira o operando do topo da pilha e guarda-o na variável local `arg`, utilizando a função `pop`. Depois, recorrendo à função `push`, coloca o valor de `arg` no topo da pilha duas vezes, clonando assim o valor. Se `sp < 0`, a função dá *print* de "Not enough values in the stack".
- `void drop(struct stack *stack)`: função que recebe a `stack` como argumento e é responsável pela remoção do operando do topo da pilha, mas apenas se `sp > 0`.

Para isso retira da pilha o operando do topo e guarda-o numa variável que não será usada (`arg`).

- `int process_input(char input[32], struct stack *stack)`: função que processa a *input* do utilizador. Esta função recebe como argumentos a `input[32]` e a `struct stack *stack` e tem como variáveis locais a variável `int number` (número a ser guardado na *stack*), `int run` (controla a continuação da leitura do *input* do utilizador) e `int aux` (controla quando é colocado um novo número na *stack*). Esta função corre, principalmente, com um ciclo `for` (`for (int i = 0; run; i++)`) que encerra assim que o caractere nulo for lido (código ASCII - 0). Para cada caractere lido é avaliado se é um número ou uma expressão válida (que represente uma operação ou função). Se o caractere lido for um número, este é colocado na *stack* e atualiza a variável `aux` de modo a permitir a chamada da função `push` após encontrado o caractere `' '` (código ASCII - 32). Se encontrado o caractere nulo, é dado *"push"* do `number` na *stack* se `aux==1` e coloca a variável `run` a 0 para terminar o ciclo `for`. A função avalia ainda se o caractere/ conjunto de caracteres correspondem aos caracteres definidos para as operações/funções desempenhadas pela calculadora, executando a função correspondente. Caso não se verifique igualdade de caracteres, a função apresenta a mensagem `"Unkown command. Type 'help' for available commands"`. Se a *input* do utilizador for `"off"`, a função retorna 1, se não retorna 0. Por fim, e após a execução do ciclo `for`, a função mostra a pilha de operandos recorrendo à função `print`.

## 5 Função Main

A função `void main()` é a função principal da calculadora. É nela que as funções são chamadas e é feita a interação com o utilizador. É feita a criação e inicialização da *stack* com 0 elementos e alocada na memória. Depois desta inicialização é feita a leitura do *input* do utilizador e guardado no *array* de caracteres `input[32]`. Neste ponto o programa entra no ciclo infinito (`while(1)`) até que a função `process_input` retorne 1 (`off = process_input(input, stack)`). Neste ciclo, são apresentadas as operações introduzidas pelo utilizador e é pedida novamente uma nova *input* ao mesmo. No final do programa a mensagem `"Bye!"` é apresentada no ecrã e é libertada a memória que foi usada para a *stack* (`free(stack)`).