**A GUIDE TO THE MONTY PROJECT**

The knowledge of the data structures stacks and queues in C language, gives a base to build a C program that acts as a Monty bytecode interpreter.

**So what is a Monty bytecode interpreter ?**

This is a C program that will take a file as an argument, open the file, and read the instructions in the file, line by line and then go ahead and execute these instructions.
Monty bytecode is made up of **special instructions** and each line contains either just one instruction or no instruction at all.
Files containing Monty byte codes usually have **.m** extension.
These special instructions are in the form of opcodes that are native to the Monty scripting language and each opcode performs a particular task or rather executes a particular functionality.
Below is a table detailing each opcode and its subsequent operation.

**To successfully do this project you could essentially follow the steps below:**

Create a C program that accepts a file as an argument. Why ? Because To run our Monty byte code interpreter you will type the following command :
**./monty bytecode_file.m**
This means your C program has to first check whether the user provided a monty byte code file as an argument.
Your interpreter should therefore be expecting 2 arguments (the name of the program itself followed by the name or path to the file with the Monty bytecodes).
This will be checked within your main function, whether argc=2. If not then,throw an error message

From there the next thing would be to check if the provided file is accessible or not and if it is then go ahead and open the file.

This done within your **main()** function.

From this point you can go on and define a few helper functions that will help your interpreter to run successfully.

1. You need a function to check if the first character in a line is a newline
2. You need a function to tokenize the command and value in the line.
3. You need a function that uses the first token (command) to find what opcode(instruction) to execute.
4. You will also need a function to free everything in your stack.

After creating these functions you then go ahead and define functions that will execute each of the following opcodes below. Including checking for comments.
A call to any of these functions means an execution of the opcode. This is what your interpreter should ‘interpret’ .

All these functions can be in separate **.c** files or you could group them together in individual files but following the betty coding style of no more than 5 functions per file.

Don’t forget to create your monty.h header file to include the prototypes of all your functions

Once done you will compile all your **.c** files using the command provided in the project guidelines
```
gcc -Wall -Werror -Wextra -pedantic -std=c89 *.c -o monty
```

At this point you can go ahead and create a folder that will have a couple of monty bytecode files with the **.m** extension. These files will serve as tests to check whether your interpreter works.

| Opcode | Description |
|---|---|
| push | Pushes an element to the stack. e.g (push 1 # pushes 1 into the stack) |
| pall | Prints all the values on the stack, starting from the top of the stack. |
| pint | Prints the value at the top of the stack. |
| pop | Removes the to element out of the stack. |
| swap | Swaps the top to elements of the stack. |
| add | Adds the top two elements of the stack. The result is then stored in the second node, and the first node is removed. |
| nop | This opcode does not do anything. |
| sub | Subtracts the top two elements of the stack from the second top element. The result is then stored in the second node, and the first node is removed. |
| div | Divides the top two elements of the stack from the second top element. The result is then stored in the second node, and the first node is removed. |
| mul | Multiplies the top two elements of the stack from the second top element. The result is then stored in the second node, and the first node is removed. |
| mod | Computes the remainder of the top two elements of the stack from the second top element. The result is then stored in the second node, and the first node is removed. |
| # | When the first non-space of a line is a # the line will be trated as a comment. |
| pchar | Prints the integer stored in the top of the stack as its ascii value representation. |
| pstr | Prints the integers stored in the stack as their ascii value representation. It stops printing when the value is 0, when the stack is over and when the value of the element is a non-ascii value. |
| rotl | Rotates the top of the stack to the bottom of the stack. |
| rotr | Rotates the bottom of the stack to the top of the stack. |
| stack | This is the default behavior. Sets the format of the data into a stack (LIFO). |
| queue | Sets the format of the data into a queue (FIFO). |