0x01. C - Stacks, Queues - LIFO, FIFO

System programming & Algorithm — Data structures and Algorithms

by Julien Barbier, co-founder at Holberton School

weight: 2

Ongoing project - started 06-05-2017, must end by 06-12-2017 (in 3 days) - you're 0% done.

QA review fully automated.

Readme

Read Google (https://www.google.com/webhp?sourceid=chrome-instant&rlz=1C1CHFX_enUS688US688&ion=1&espv=2&ie=UTF-8#q=stack%20and%20queue), How do I use extern to share variables between source files in C? (http://stackoverflow.com/questions/1433204/how-do-i-use-extern-to-share-variables-between-source-files-in-c), and Working with submodules (https://github.com/blog/2104-working-with-submodules).

What you should learn from this project

At the end of this project you are expected to be able to explain to anyone, without the help of Google:

- What is LIFO and FIFO
- What is a stack, and when to use it
- What is a queue, and when to use it
- What are the common implementations of stacks and queues
- What are the most common use cases of stacks and queues
- What is the proper way to use global variables
- How to work with git submodules

Requirements

- Allowed editors: `vi` , `vim` , `emacs`
- All your files will be compiled on Ubuntu 14.04 LTS
- Your programs and functions will be compiled with `gcc 4.8.4` (`C90`) using the flags `-Wall -Werror -Wextra and -pedantic`
- All your files should end with a new line
- A `README.md` file, at the root of the folder of the project is mandatory
- Your code should use the `Betty` style. It will be checked using betty-style.pl (https://github.com/holbertonschool/Betty/blob/master/betty-style.pl) and betty-doc.pl (https://github.com/holbertonschool/Betty/blob/master/betty-doc.pl)
- You allowed to use a maximum of one global variable
- No more than 5 functions per file
- You are allowed to use the C standard library
- The prototypes of all your functions should be included in your header file called `monty.h`
- Don't forget to push your header file
- All your header files should be include guarded
- You are expected to do those tasks in the order shown in the project
- Please use those data structures for this project:

```
/**
 * struct stack_s - doubly linked list representation of a stack (or que
ue)
 * @n: integer
 * @prev: points to the previous element of the stack (or queue)
 * @next: points to the next element of the stack (or queue)
 *
 * Description: doubly linked list node structure
 * for stack, queues, LIFO, FIFO Holberton project
 */
typedef struct stack_s
{
        int n;
        struct stack_s *prev;
        struct stack_s *next;
} stack_t;
```

```
/**
 * struct instruction_s - opcoode and its function
 * @opcode: the opcode
 * @f: function to handle the opcode
 *
 * Description: opcode and its function
 * for stack, queues, LIFO, FIFO Holberton project
 */
typedef struct instruction_s
{
        char *opcode;
        void (*f)(stack_t **stack, unsigned int line_number);
} instruction_t;
```

Compilation

- Your code will be compiled this way:

```
$ gcc -Wall -Werror -Wextra -pedantic *.c -o monty
```

Tests

- We strongly encourage you to work all together on a set of tests

The Monty language

Monty 0.98 is a scripting language that is first compiled into Monty byte codes (Just like Python). It relies on a unique stack, with specific instructions to manipulate it. The goal of this project is to create an interpreter for Monty ByteCodes files.

Tasks

0. push, pall #advanced

☐ Done?

Help!

Implement the `push` and `pall` opcodes.

Monty byte code files

Files containing Monty byte codes usually have the `.m` extension. Most of the industry uses this standard but it is not required by the specification of the language. There is not more than one instruction per line. There can be any number of spaces before or after the opcode and its argument:

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat -e bytecodes/000.m
push 0$
push 1$
push 2$
  push 3$
                   pall    $
push 4$
    push 5    $
      push    6        $
pall$
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Monty byte code files can contain blank lines (empty or made of spaces only, and any additional text after the opcode or its required argument is not taken into account:

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat -e bytecodes/001.m
push 0 Push 0 onto the stack$
push 1 Push 1 onto the stack$
$
push 2$
  push 3$
                   pall    $
$
$
                           $
push 4$
$
    push 5    $
      push    6        $
$
pall This is the end of our program. Monty is awesome!$
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

The monty program

- Usage: `monty file`
- where `file` is the path to the file containing Monty byte code
- If the user does not give any file or more than one argument to your program, print `USAGE: monty file`, followed by a new line, and exit with the status `EXIT_FAILURE`
- If, for any reason, it's not possible to use read the file, print `Error: Can't open file <file>`, followed by a new line, and exit with the status `EXIT_FAILURE`
    - where `<file>` is the name of the file
- If the file contains an invalid instruction, print `L<line_number>: unknown instruction <opcode>`, followed by a new line, and exit with the status `EXIT_FAILURE`
    - where is the line number where the instruction appears. Line numbers always start at 1
- The monty program runs the bytecodes line by line and stop if:
    - it executed properly every line of the file
    - or it finds an error in the file
    - or an error occured
- If you can't malloc anymore, print `Error: malloc failed`, followed by a new line, and exit with status `EXIT_FAILURE`. You have to use `malloc` and `free` and are not allowed to use any other function from `man malloc`

The push opcode

The opcode `push` pushes an element to the stack.

- Usage: `push <int>`
- where `<int>` is an integer
- if `<int>` is not an integer or if there is no argument to `push`, print the message `L<line_number>: usage: push integer`, followed by a new line, and exit with the status

    EXIT_FAILURE

    - where is the line number in the file
- You don't have to deal with overflows. Use the `atoi` function

The pall opcode

The opcode `pall` prints all the values on the stack, starting from the top of the stack.

- Usage `pall`
- Format: see example
- If the stack is empty, don't print anything

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat -e bytecodes/00.m
push 1$
push 2$
push 3$
pall$
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/00.m
3
2
1
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

1. pint #advanced

☐ Done?

Help!

Implement the `pint` opcode.

The pint opcode

The opcode `pint` prints the value at the top of the stack, followed by a new line.

- Usage: `pint`
- If the stack is empty, print `L<line_number>: can't pint, stack empty`, followed by a new line, and exit with the status `EXIT_FAILURE`

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/06.m
push 1
pint
push 2
pint
push 3
pint
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/06.m
1
2
3
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

2. pop

#advanced

☐ Done?

Help!

Implement the `pop` opcode.

The pop opcode

The opcode `pop` removes the top element of the stack.

- Usage: `pop`
- if the stack is empty, print `L<line_number>: can't pop an empty stack`, followed by a new line, and exit with the status `EXIT_FAILURE`

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/07.m
push 1
push 2
push 3
pall
pop
pall
pop
pall
pop
pall
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/07.m
3
2
1
2
1
1
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

3. swap #advanced

☐ Done?

Help!

Implement the `swap` opcode.

The swap opcode

The opcode `swap` swaps the top two elements of the stack.

- Usage: `swap`
- If the stack is less than two element long, print `L<line_number>: can't swap, stack too short`, followed by a new line, and exit with the status `EXIT_FAILURE`

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/09.m
push 1
push 2
push 3
pall
swap
pall
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/09.m
3
2
1
2
3
1
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

4. add #advanced

☐ Done?

Help!

Implement the `add` opcode.

The add opcode

The opcode `add` adds the top two elements of the stack.

- Usage: `add`
- If the stack is less than two element long, print `L<line_number>: can't add, stack too short` , followed by a new line, and exit with the status `EXIT_FAILURE`
- The result is stored in the second top element of the stack, and the top element is removed, so that at the end:
    - the top element of the stack contains the result
    - the stack is one element shorter

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/12.m
push 1
push 2
push 3
pall
add
pall

julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/12.m
3
2
1
5
1
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

5. nop

#advanced

☐ Done?

Help!

Implement the `nop` opcode.

The nop opcode

The opcode `nop` doesn't do anything.

- Usage: `nop`

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

6. sub #advanced

☐ Done?

Help!

Implement the `sub` opcode.

The sub opcode

The opcode `sub` subtracts the top element of the stack from the second top element of the stack.

- Usage: `sub`
- If the stack is less than two element long, print `L<line_number>: can't sub, stack too short`, followed by a new line, and exit with the status `EXIT_FAILURE`
- The result is stored in the second top element of the stack, and the top element is removed, so that at the end:
    - the top element of the stack contains the result
    - the stack is one element shorter

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/19.m
push 1
push 2
push 10
push 3
sub
pall
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/19.m
7
2
1
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

7. div #advanced

☐ Done?

Help!

Implement the `div` opcode.

The div opcode

The opcode `div` divides the second top element of the stack by the top element of the stack.

- Usage: `div`
- If the stack is less than two element long, print `L<line_number>: can't div, stack too short`, followed by a new line, and exit with the status `EXIT_FAILURE`
- The result is stored in the second top element of the stack, and the top element is removed, so that at the end:
    - the top element of the stack contains the result
    - the stack is one element shorter
- If the top element of the stack is `0`, print `L<line_number>: division by zero`, followed by a new line, and exit with the status `EXIT_FAILURE`

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

8. mul #advanced

☐ Done?

Help!

Implement the `mul` opcode.

The mul opcode

The opcode `mul` multiplies the second top element of the stack with the top element of the stack.

- Usage: `mul`
- If the stack is less than two element long, print `L<line_number>: can't mul, stack too short`, followed by a new line, and exit with the status `EXIT_FAILURE`
- The result is stored in the second top element of the stack, and the top element is removed, so that at the end:
    - the top element of the stack contains the result
    - the stack is one element shorter

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

9. mod

#advanced

☐ Done?

Help!

Implement the `mod` opcode.

The mod opcode

The opcode `mod` computes the rest of the division of the second top element of the stack by the top element of the stack.

- Usage: `mod`
- If the stack is less than two element long, print `L<line_number>: can't mod, stack too short`, followed by a new line, and exit with the status `EXIT_FAILURE`
- The result is stored in the second top element of the stack, and the top element is removed, so that at the end:
    - the top element of the stack contains the result
    - the stack is one element shorter
- If the top element of the stack is `0`, print `L<line_number>: division by zero`, followed by a new line, and exit with the status `EXIT_FAILURE`

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

10. comments

#advanced

☐ Done?

Help!

Every good language comes with the capability of commenting. When the first non-space character of a line is `#`, treat this line as a comment (don't do anything).

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

11. pchar #advanced

☐ Done?

Help!

Implement the `pchar` opcode.

The pchar opcode

The opcode `pchar` prints the char at the top of the stack, followed by a new line.

- Usage: `pchar`
- The integer stored at the top of the stack is treated as the ascii value of the character to be printed
- If the value is not in the ascii table (man ascii) print `L<line_number>: can't pchar, value out of range`, followed by a new line, and exit with the status `EXIT_FAILURE`
- If the stack is empty, print `L<line_number>: can't pchar, stack empty`, followed by a new line, and exit with the status `EXIT_FAILURE`

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/28.m
push 72
pchar
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/28.m
H
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

12. pstr #advanced

☐ Done?

Help!

Implement the `pstr` opcode.

The pstr opcode

The opcode `pstr` prints the string starting at the top of the stack, followed by a new line.

- Usage: `pstr`
- The integer stored in each element of the stack is treated as the ascii value of the character to be printed
- The string stops where:
    - the stack is over

    - the value of the element is 0
    - the value of the element is not in the ascii table
- If the stack is empty, print only a new line

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/31.m
push 1
push 2
push 3
push 4
push 0
push 110
push 0
push 110
push 111
push 116
push 114
push 101
push 98
push 108
push 111
push 72
pstr
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/31.m
Holberton
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

13. rotl #advanced

Done?

Help!

Implement the `rotl` opcode.

The rotl opcode

The opcode `rotl` rotates the stack to the top.

- Usage: `rotl`
- The top element of the stack becomes the last one, and the second top element of the stack becomes the first one
- `rotl` never fails

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/35.m
push 1
push 2
push 3
push 4
push 5
push 6
push 7
push 8
push 9
push 0
pall
rotl
pall
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/35.m
0
9
8
7
6
5
4
3
2
1
9
8
7
6
5
4
3
2
1
0
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

14. rotr

#advanced

☐ Done?

Help!

Implement the `rotr` opcode.

The rotr opcode

The opcode `rotr` rotates the stack to the bottom.

- Usage: `rotr`
- The last element of the stack becomes the top element of the stack
- `rotr` never fails

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

15. stack, queue

#advanced

☐ Done?

Help!

Implement the `stack` and `queue` opcodes.

The stack opcode

The opcode `stack` sets the format of the data to a stack (LIFO). This is the default behavior of the program.

- Usage: `stack`

The queue opcode

The opcode `queue` sets the format of the data to a queue (FIFO).

- Usage: `queue`

When switching mode:

- the top of the stack becomes the front of the queue
- the front of the queue becomes the top of the stack

```
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ cat bytecodes/47.m
queue
push 1
push 2
push 3
pall
stack
push 4
push 5
push 6
pall
add
pall
queue
push 11111
add
pall
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$ ./monty bytecodes/47.m
1
2
3
6
5
4
1
2
3
11
4
1
2
3
15
1
2
3
11111
julien@ubuntu:~/0x18. Stack (LIFO) & queue (FIFO)$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`

Check your code?

16. Holberton #advanced

☐ Done?

Help!

Write a Brainf*ck script that prints `Holberton` , followed by a new line.

- All your Brainf*ck files should be stored inside the `brainfuck` sub directory
- You can install the `bf` interpreter to test your code: `sudo apt-get install bf`
- Read: Brainf*ck (https://en.wikipedia.org/wiki/Brainfuck)

```
julien@ubuntu:~/brainfuck$ bf 1000-holberton.bf
Holberton
julien@ubuntu:~/brainfuck$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`
- File: `1000-holberton.bf`

Check your code?

17. Add two digits #advanced

☐ Done?

Help!

Add two digits given by the user.

- Read the two digits from stdin, add them, and print the result
- The total of the two digits with be one digit-long (<10)

```
julien@ubuntu:~/brainfuck$ bf ./1001-add.bf
81
9julien@ubuntu:~/brainfuck$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`
- File: `1001-add.bf`

Check your code?

18. Multiplication

#advanced

Multiply two digits given by the user.

- Read the two digits from stdin, multiply them, and print the result
- The result of the multiplication will be one digit-long (<10)

```
julien@ubuntu:~/braifuck$ bf 1002-mul.bf
24
8julien@ubuntu:~/braifuck$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`
- File: `1002-mul.bf`

☐ Done?

Help!

Check your code?

19. Multiplication level up

#advanced

☐ Done?

Help!

Multiply two digits given by the user.

- Read the two digits from stdin, multiply them, and print the result, followed by a new line

```
julien@ubuntu:~/brainfuck$ bf 1003-mul.bf
77
49
julien@ubuntu:~/brainfuck$
```

Repo:

- GitHub repository: `holbertonschool-datastructures_algorithms`
- Directory: `0x01-c_stacks_queues_lifo_fifo`
- File: `1003-mul.bf`

Check your code?