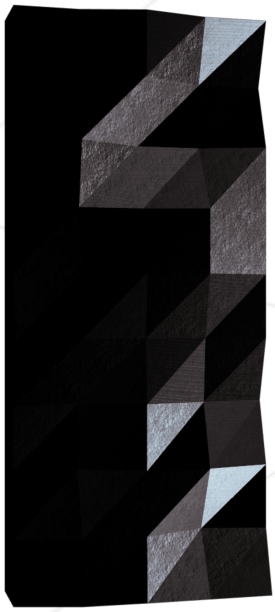

Summer 2023 Review Campaign

Code Refactoring

Summary: Time to clean up some messy code!

Contents

Chapter I

Foreword

Eminem - Lose Yourself
His palms are sweaty, knees weak, arms are heavy.
There's vomit on his sweater already, mom's spaghetti.

What makes good code? Bad code? Spaghetti code?
In this project, you will learn about the principles of clean code.

Chapter II

Instructions

- Chapters 3-4 are the base excercise for bsq.
- You will be primarily evaluated on chapter 5: Refactor.
- However, in order to be evaluated on chapter 5, the program must be functional!
- If your program does not pass the criteria from chapter 3 you will receive a 0.
- Some aspects of your evaluation will be subjective. Be prepared to defend your code!
- You are not required to abide by the norminette for this project.

Chapter III

Subject

Program name	`bsq`
Turn in files	`Makefile and all the necessary files`
Makefile	`Yes`
Arguments	`File(s) in which to read the square`
External functs.	`open, close, read, write, malloc, free, exit`
Libft authorized	`Not applicable`
Description	`Write a program that draws and print the biggest possible square in the given area`

- The biggest square :
  - The aim of this project is to find the biggest square on a map, avoiding obstacles.
  - A file containing the map will be provided. It'll have to be passed as an argument for your program.
  - The first line of the map contains information on how to read the map :
    - The number of lines on the map;
    - The "empty" character;
    - The "obstacle" character;
    - The "full" character.
  - The map is made up of `'"empty" characters'`, lines and `'"obstacle" characters'`.
  - The aim of the program is to replace `'"empty" characters'` by `'"full" characters'` in order to represent the biggest square possible.
  - In the case that more than one solution exists, we'll choose to represent the square that's closest to the top of the map, then the one that's most to the left.
  - Your program must handle 1 to n files as parameters.

- When your program receives more than one map in argument, each solution or `map error` must be followed by a line break.
- Should there be no passed arguments, your program must be able to read on the standard input.
- You should have a valid Makefile that'll compile your project. Your Makefile mustn't relink.

- Definition of a valid map :
  - All lines must have the same length.
  - There's at least one line of at least one box.
  - At each end of line, there's a line break.
  - The characters on the map can only be those introduced in the first line.
  - The map is invalid if a character is missing from the first line, or if two characters (of empty, full and obstacle) are identical.
  - The characters can be any printable character, even numbers.
  - In case of an invalid map, your program should display `map error` on the error output followed by a line break. Your program will then move on to the next map.
- Here's an example of how it should work :

```
%>cat example_file
9.ox
.........................
....o....................
...........o.............
.........................
....o....................
.............o...........
.........................
......o.............o.....
..o.......o..............
%>./bsq example_file
.....xxxxxxx.............
....oxxxxxxx.............
.....xxxxxxxo............
.....xxxxxxx.............
....oxxxxxxx.............
.....xxxxxxx...o.........
.....xxxxxxx.............
......o.............o.....
..o.......o..............
%>
```

It is a square indeed. Even though it might not look like it visually.

Chapter IV

Annex

- Perl map generator

```
#!/usr/bin/perl

use warnings;
use strict;

die "program x y density" unless (scalar(@ARGV) == 3);

my ($x, $y, $density) = @ARGV;

print "$y.ox\n";
for (my $i = 0; $i < $y; $i++) {
    for (my $j = 0; $j < $x; $j++) {
        if (int(rand($y) * 2) < $density) {
            print "o";
        }
        else {
            print ".";
        }
    }
    print "\n";
}
```

Chapter V

Refactor

You will be provided a starting code base from which to improve.
Watch the following video, and refactor the code base to reflect the principles covered.
https://www.youtube.com/watch?v=BVwxan6WGpI&ab_channel=CodamCodingCollege

- Readability
  - Clear intent
  - Expressive and meaningful names
  - Simple and straighforward logic
  - Helpful comments
- Redundancy
  - No code duplication
- Scalability/modularity
  - Easily understood and modified
  - Easy to maintain
  - Easy to extend
- Organization
  - Intuitive file structure
  - Reasonable function size
  - Doesn't surprise the reader

To be evaluated on this chapter, your code must work as defined in chapter 3.

The above principles will be evaluated looking at primarily looking at the following:

- Naming
- Functions
- Comments
- Implementation patterns

If your code doesn't require a particular concept, don't add code to fulfil it. You will be reviewed on how often your code violates these principles, not how often it complies with them.