The program itself started as a switch case structure with a variable you could hardcode a single opcode, while far from optimal, a functional temporary solution took no work and allowed for some fundamental testing. A struct was also made to allow carrying all the data to the functions that needed it. Work also began on some of the opcodes, of which many were made quickly, but because of general lacking in debugging potential, the tests were not the best, but as these opcodes were quite simple, only a few had fundamental shortcomings.

The next problem to solve was the messed nature of and nested switch case structure; I had heard about function pointers as an alternative and started to research. At first, the examples I found looked clunky needing to assign a variable to each function, because I did not want to do that. I went to the internet to see if I could instead assign a lot of them to a single array at the time, and after diving in into the web, I found this (<http://www.multigesture.net/wp-content/uploads/mirror/zenogais/FunctionPointers.htm> ) which seemed like the perfect solution. After messing about with it a bit I got the hang of it and the new decoding structure was complete.

With that problem out of the way, I could work on a better way of getting opcodes. While precoding a single opcode could work for a few of the simple opcodes, it was useless for any and every opcode that read to/from memory or another register. It was time for a proper way to do this, reading them from memory. While quite simple, it requires a twostep solution, as the opcodes are 2 bytes long, while each memory cell is one.   
While in ELE112, I would normally just write the first to the top half and the second to the lower, this did not seems possible in C. Instead this is achieved by loading one, shifting it 8 spaces to the left and then loading the other with an OR statement, which acts the same as normal assignment when writing to zeroes, but without affecting any bit outside its range.

Now opcodes are read from memory and nice as that is, it is not doing much when memory is empty meaning it is time to make a rom loader. After reading around a bit I decided that it should also be able to stop if the file it was trying to read was too big, but with a lack of knowhow for doing this is searched for inspiration, only to find that the source of one of the first learning pages we found had just this. (<http://www.multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/> )  
While wanting to write my own, this one just seemed functional, and any change I could have made would probably just make it worse, so I ended up just repurposing it for our own code.

Next problem came with the FX-series and its naming scheme, while there is only nine of them, several have the same character as each other in the last or second to last spot. This meant they need a two level identification, in addition to the F, as opposed to every other opcode, including the 8-series where they all have different last characters and making it perfect for function pointers.   
FX on the other hand, seems a better fit for a switch case. But that would look messy again so I made a 2d array of pointers, mostly filled with a function that says "this is not an opcode" and ends the program, while it seems a bit excessive, i still find that is looks better than a switch case would and it lets the code keep it uniform shape of function pointers.

Along the way, we also had a bunch of problems with SDL, our chosen UI choice. After stagnating the project for a while trying to get SDL to work, I decided waiting for the others to learn SDL and get a usable display up and running was not worth it. Instead, I repurposed some code I already had made on the side in a few hours, so we went for a SDL/openGL hybrid and could write shapes to the screen.

Next, after some push from our teacher I made some shitty debugging, letting us see opcodes and registers and following the flow of a program. It turns out this was really hard and unnecessary when there were better ways of doing it, but it made us find a few problems, and fixing errors like the DXYN not always setting the collision flag properly causing Tetris pieces to fall through the ground.

While testing the different games, we also found that some just straight up crashed at certain points. After looking into this, we found that this had root in something about the graphics and narrowed it down to happening when trying to draw multi height sprites at the lowest row, leading to the conclusion that it was somehow trying to draw outside the screen.   
This caused some confusion because the renderer was set not to draw outside the given range of pixels, but this is where the error occurred. Eventually after some frustration hours, we found that the cause was DXYN trying to write outside the GFX array. While it at first seemed like this was already impossible it was because it was only the first row of the sprite was impossible to draw lower than the lowest pixel row, nothing stopped it from trying to do that with the next rows. This sometimes caused it to change other registers and other times caused SDL to crash, probably due to the array being "corrupted" this was fixed by making the DXYN stop writing when the given x and y in addition to the current row and column was over 32\*64 (the resolution of the chip 8)

At this point, we had all the opcodes working to some degree, a display, input, and rom loading and could run some games, with errors mind you, but we had proof that we were getting somewhere. And after fixing some big errors in some opcodes the games ran, but not perfectly. Some of the games seemed incredibly slow, while others seemed fine and a few of them seemed impossible as like invaders ( a space invaders clone) where the aliens were moving as fast as you, making it almost impossible to beat. Not knowing this was not the way it was supposed to work I accepted it, thinking the game was just some proof of concept using the limited abilities of the chip8 to its best ability.  
However, I was wrong, and after rereading about the timers, I realised the obvious; the timers are not supposed to be controlled by the number of cycles, but rather real time. After fixing this by adding a condition, checking time difference from last time the timers decreased everything ran much smoother, and fixing the issues with timing and the flow of the games we had encountered earlier.

After this I have just done a bit restructuring, some clean-up, made proper tests, revised the debugger and a few bug fixes and now the core emulator is looking highly stable and is working great, being pretty much done (there was an issue where one of the buttons wasn’t working at all though)