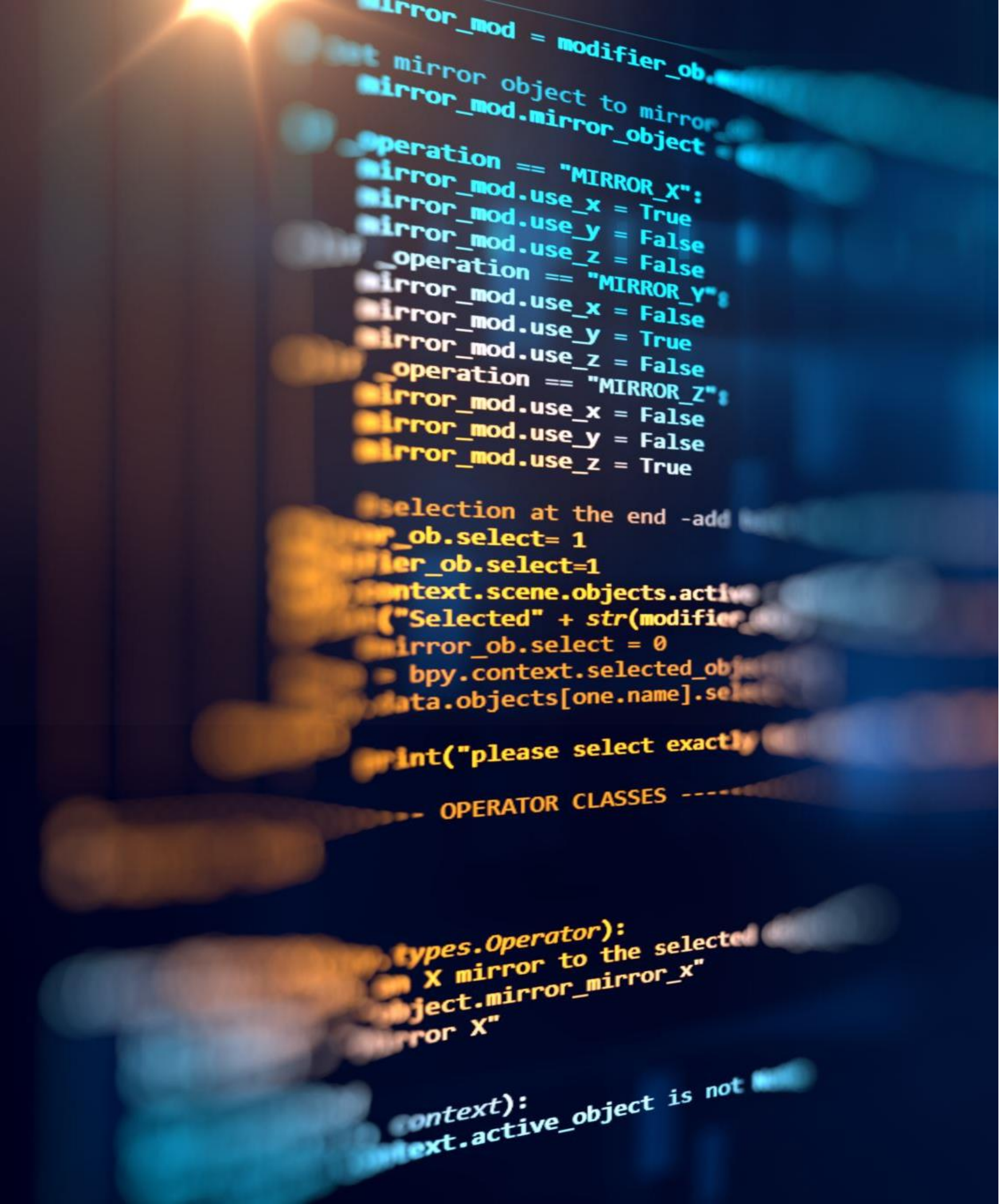
TEXT-BASED MATH CALCULATOR PT I.



WHAT ARE THE GOALS?

- There is an ASU course called: ***CSE340 [Principles of Programming Languages]***
- In ***CSE340***, the course covers how a compiler and/or interpreter takes the first steps in processing a programming language.
  - ***NOT an easy class***: The course is heavy in terms of theory as well as requiring a good grasp of data structures that are required to properly process code and why they are used.
  - Much of the basis of the course is putting *that* theory you're assaulted with into working code.
- If you're a **Comp-Sci** major, you are **required** to take & pass ***CSE340*** with a minimum grade of C.
- The goal of this very workshop is to help, as much as possible, prepare you for this course when you reach ASU.

1ST PART - LEXICAL ANALYSIS AKA LEXING

- The point of lexical analysis is to break up meaningful pieces of text, differentiate them according to what each part means, and assign a symbolic meaning to the differentiated parts.
- Example: this piece of code in Java/C: `int a = f(x) * numbers[3] + 100;`
  - The code would enter the compiler/interpreter typically from a text file or from a command-line input.
  - Typically, the text, whether from a file or command-line, is read as a string type.
  - Spaces, tabs, and newlines are usually called ***whitespace*** and are usually skipped.
    - can be used for other intentions.
    - Python uses spaces and tabs to control scoping of constructs.
- Long story short: Lexical Analysis is processing a piece of input [usually text] into a way that's programmatically easy to analyze by assigning meaning to the symbols in the input stream.
  - Example:
    - "int" = keyword
    - "a" = identifier/name
    - "=" = assignment symbol
  - The symbols with their associated meaning and other data are called **Tokens**.
    - Tokens usually store the symbol's meaning, its location, and its string value.

2ND PART – SYNTAX ANALYSIS AKA PARSING

- Once the input is broken up and processed into a stream of tokens [usually done as an array or one-at-a-time], the next step in the process is applying grammar to the stream of tokens.
- Before parsing, it's best to first design how the grammar of something should work.
  - How do you design grammar?
  - Special mathematical notation used to design a formalized specification of grammars.
  - A few exist, **Backus-Naur Form** is one example.
- Why? Because formalized grammars are needed to describe what set of tokens are valid for the syntax.
- For our case, we will have a very simple formal grammar to help us create our calculator.
  - Just enough to properly
- Long story short, Parsing is taking the token stream as an input and producing a type of output, usually a type of data structure or other data.

FIRST THINGS FIRST: READING INPUT

- Two ways we can read input:
  - Prompting the user for input.
  - Reading directly from a file.
- For our purposes, we will prompt the user to input an equation to calculate.

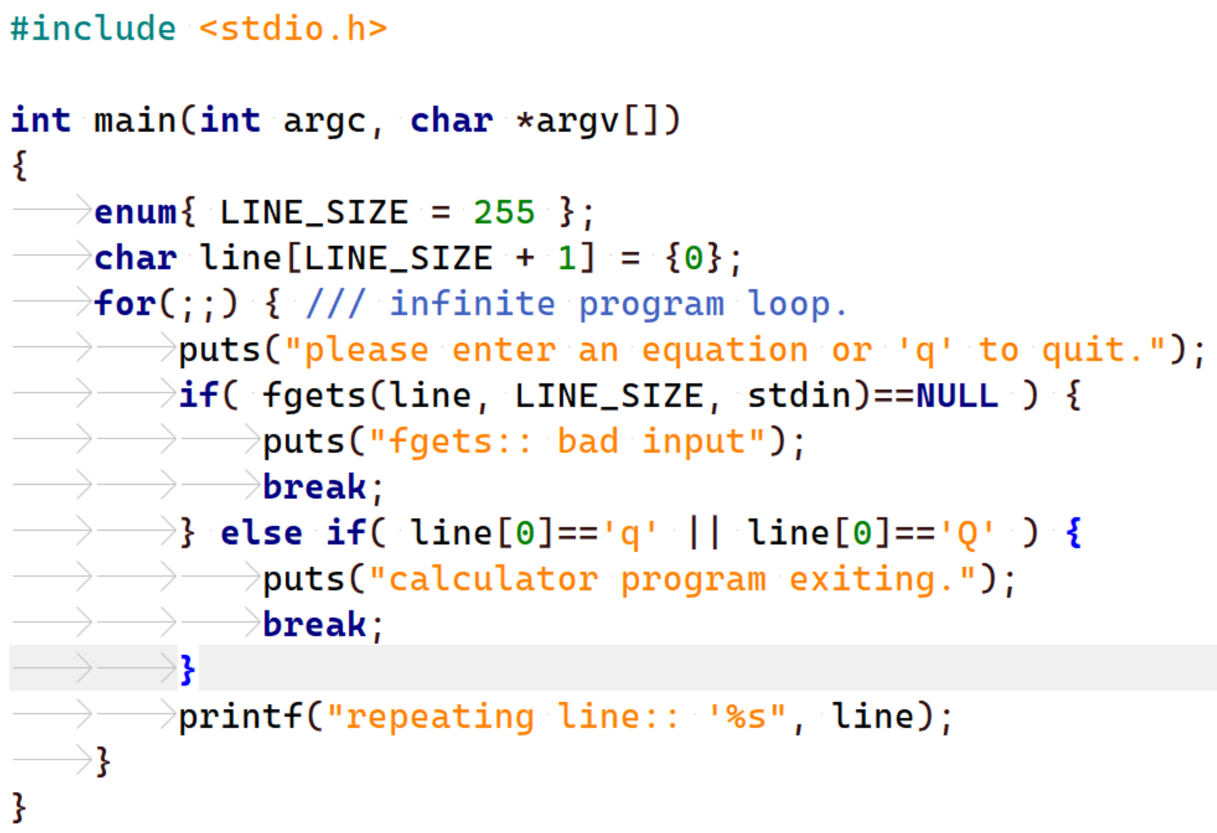
```
char *fgets(char *str, int count, FILE *stream);
```

- "fgets" ("file get string") function in C's standard library will be our friend here.
- It'll prompt the user for input and grab it as a line of text.
- How will we use it? Like so:

```
fgets(line, line_length, stdin);
```

- **Note**: "fgets" usually adds a newline character at the end of the array.

INTERACTIVE COMMAND-LINE PROGRAM



```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    enum{ LINE_SIZE = 255 };
    char line[LINE_SIZE + 1] = {0};
    for(;;) { /// infinite program loop.
        puts("please enter an equation or 'q' to quit.");
        if( fgets(line, LINE_SIZE, stdin)==NULL ) {
            puts("fgets:: bad input");
            break;
        } else if( line[0]=='q' || line[0]=='Q' ) {
            puts("calculator program exiting.");
            break;
        }
        printf("repeating line:: '%s", line);
    }
}
```

- We also want to make our text-input calculator be more interactive.
    - We don't want it to end the program as soon as it finishes calculating.
- All that's required is a **program loop** and a **condition** that will terminate said program loop.
    - We'll prompt the user that, if they want to quit, they must type '*q*'.
    - Even if the user fat-fingers it, as long as the first letter is 'q', it'll quit.

TEXT-INPUT CALCULATOR WORKSHOP SCHEDULE

- **Intro (talking about Lexical Analysis & Parsing) & Reading file input in C** (*February 2nd, 2024*) [**Today, Done**]
- **Lexically analyzing numbers, names, and operators/symbols** (*Projected February 9th, 2024*) [***Up Next***]
- **Intro to Parsing theory [Backus-Naur form, Extended Backus-Naur form, Parsing Expression Grammar]** (*Projected February 16th, 2024*)
- **Recursive Descent Parsing and testing** (*Projected February 23rd, 2024*)
- **Adding functions like *ln* and *sin*** (*Projected March 1st, 2024*)