Watch The Register

Problem Description

The full description can be found in this site's repository.

You are working with a very simple model of CPU that only has one register, `X`. This register, on startup, holds the value `1`. This register only supports two instructions:

- `addx n` takes *two cycles* to complete. After two cycles, the `X` register is increased by the value of `n` (`n` is an integer).
- `noop` takes one cycle to complete. It has no other effect.

Upon further investigation, you found that the `X` register controls the horizontal position of a sprite. Specifically, the sprite is 3 pixels wide, and the `X` register sets the horizontal position of the middle of the sprite. The CPU's display is 40 pixels wide and 6 pixels tall. The CPU draws a single pixel during each cycle count, either a `#` or a `.`.

Each pixel is formatted as follows. If the sprite is positioned such that one of its three pixels is the pixel currently being drawn, the screen produces a lit pixel (`#`); otherwise, the screen leaves the pixel dark (`.`).

Render the image that the CPU would output given the input program in *input.txt*. Determine the 8 capital letters that appear on the display, and turn in those letters embraced in `flag{}` braces.

Walkthrough

The hardest challenge here is understanding the challenge. We're asked to monitor a register that contains an integer as well as the cycle count (an unsigned integer). This hints us to use a `uint32_t` for the cycle count and an `int` for the register.

There are two ways to track the lights that are printed to the console: printing them as the sprite iterates or storing them in a buffer. I chose to store them as a buffer. There are two ways to store the buffer:

- A 2D array of characters (6 rows of 40 characters)
- A 1D array of characters (240 characters)

The second one will prove to be much easier to work with, especially when we're at the end of each row. We'll handle the formatting of the rows when we print the final flag.

Now that we have our data structures chosen, we need to read and parse the data.

Reading the File

The format of the file is a series of lines that we need to read one at a time. Therefore, we can use `getline()` as our read function.

We'll use the same initialization as the previous challenge:

```
FILE* fp = fopen("input.txt", "r");
if (fp == NULL)
exit(EXIT_FAILURE);

size_t len;
ssize_t read;
char* line = NULL;
```

Next, we'll initialize the data structures we're using for the challenge.

```
size_t cycles = 1;          // cycle counter
int X = 1;                  // register value of X
char lights[240] = {0};     // array of lights
```

We will also use the same iterating logic as the previous challenges to read the file until we reach an EOF.

```
while ((read = getline(&line, &len, fp)) != -1) { ... }
```

Parsing the Data

For each line, we need to decide whether the instruction is a `noop` or an `addx`. There are multiple ways to do this, so we'll make sure that we do this in the most robust way possible.

> *One of the most common ways people do this is by comparing the first letter*. Although this solution works, it's not very robust and expandable. We'll look at another way that better uses the string operatins present in the `string.h` library.

The first thing we need is the sprite's position within the display. **This is the position within the row**. We can get this by converting the cycle count into its position on the display. We subtract one from the cycle count because the display uses zero-based indexing, and we used one-based indexing for the cycle count.

```
int position = (cycles - 1) % 40;
```

Next, we need to decide how to handle the instructions. We want to handle both cases in the same iteration. For `noop` and `addx`, we need to process one cycle count. For `addx`, we will process another cycle count.

For each cycle, we need to compare the `position` (where we are in the display) to the register `X`. If our register is somewhere inside the sprite, we will display a `#`. Otherwise, we will display a `.`. We have the center location of the sprite, and `X` can be within one position of the center; we can use `abs(X - position)` to check the difference in positioning.

```

```

```
if (abs(X - position) <= 1)
    lights[cycles - 1] = '#';
else
    lights[cycles - 1] = '.';
```

Each time we make this comparison, we need to update the cycle count. The catch is that *we also must update the position*.

> *Why?* When we update the cycle count, this moves the sprite. Therefore, we need to account for this.

We'll handle this right after the comparison:

```
++cycles;
position = (cycles - 1) % 40;
```

Now, we need to handle the `addx` instruction. If our instruction is a `noop`, we are finished processing this instruction and can move on. However, if the instruction is an `addx`, we'll need to process another cycle count, and then update the register `X`.

To get the instruction, we will use `strtok()`. `strtok()` takes a string and a delimiter and will return the first token in the delimited string. In our case, our delimiter is `" "` (a space).

- For `noop`, there is no space, so `strtok()` will return the entire string.
- For `addx`, there is a space, so `strtok()` will return the first token, which is `addx`. To get the second token (the argument to `addx`), we'll use `strok()` again.

Let's get the instruction. If it's a `noop`, we can use `continue` to move to the next line iteration.

```
char* inst = strtok(line, " ");
if (strcmp(inst, "noop\n") == 0) continue;
```

> **There are two pitfalls to be made here.**
>
> - The first is that `getline()` leaves the newline character at the end of the line. `strtok()` does not remove this, meaning that we need to account for it in our comparison.
> - The second is that we must use `strcmp` to compare strings. The equality operator (`==`) compares *the addresses of the strings*, not the strings themselves.

If we pass these instructions, we know we have an `addx` instruction. In this case, we must process another cycle count.

```
// let another cycle go by
if (abs(X - position) <= 1)
    lights[cycles - 1] = '#';
else
```

```
    lights[cycles - 1] = '.';
++cycles;
```

Once the second instruction is over, we must update the cycle count. We can do this by using `strtok()` again to get the argument to `addx`. We'll use `atoi()` to convert the string to an integer.

```
// update the register
X += atoi(strtok(NULL, " "));
```

Why did we use `NULL`? `strtok()` keeps track of the string it's parsing. If we pass `NULL`, it will continue parsing the same string. If we pass a string, it will parse that string instead.

Printing the Flag

Once we finish parsing the file, we can print the flag. We need to print each row of the display, which is 40 characters long. Because we elected to use a 1-D array, we need to calculate in the array we are.

```
for (size_t i = 0; i < 6; ++i) {
    for (size_t j = 0; j < 40; ++j) {
        printf("%c", lights[40*i + j]);
    }
    printf("\n");
}
```

Once we're done with this, we need to free the data and close the file.

```
if (line) free(line);
fclose(fp);
```

Running this, we get the following output:

```
$ gcc -o solve solve.c && ./solve

####.#..#.####.####.####.#..#..##..####.
#....#..#....#.#.......#.#..#.#..#....#.
###..####...#..###....#..####.#......#..
#....#..#..#...#.....#...#..#.#.....#...
#....#..#.#....#....#....#..#.#..#.#....
####.#..#.####.#....####.#..#..##..####.
```

Reading the characters here, our flag is `flag{EHZFZHCZ}`.

Full Solution

The full solution is below:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
  FILE* fp = fopen("input.txt", "r");
  if (fp == NULL)
    exit(EXIT_FAILURE);

  size_t len;
  ssize_t read;
  char* line = NULL;

  size_t cycles = 1;          // cycle counter
  int X = 1;                  // register value of X
  char lights[240] = {0};     // array of lights
  while ((read = getline(&line, &len, fp)) != -1) {
    int position = (cycles - 1) % 40;

    // check cycles
    if (abs(X - position) <= 1)
      lights[cycles - 1] = '#';
    else
      lights[cycles - 1] = '.';
    ++cycles;
    position = (cycles - 1) % 40;

    char* inst = strtok(line, " ");
    if (strcmp(inst, "noop\n") == 0) continue;

    // let another cycle go by
    if (abs(X - position) <= 1)
      lights[cycles - 1] = '#';
    else
      lights[cycles - 1] = '.';
    ++cycles;

    // finally finish operation
    X += atoi(strtok(NULL, " "));
  }

  printf("\n");
  for (size_t i = 0; i < 6; ++i) {
    for (size_t j = 0; j < 40; ++j) {
      printf("%c", lights[40*i + j]);
    }
    printf("\n");
  }

  fclose(fp);
```

```
    if (line) free(line);
  }
```