Supply Stacks

Problem Description

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

You are going to write an algorithm which opens a file `input.txt` and follows a series of instructions to switch supply crates between stacks. You must track the crates that are on the top of each stack.

The top of the file contains the current arrangement of the crates. Following that is the rearrangement procedure.

The Marines just need to know which crates are going to be on top of the stack. Submit your flag as the order of what's on the top of each stack wrapped in flag braces, i.e. `flag{CMZ}`.

Walkthrough

This is a difficult challenge. This is a hard file to parse. The primary difference between this file and the challenges before this is that this challenge *comes with initial data* meaning we can't simply start parsing instructions. We need to take time to collect the initial data before parsing the instructions. We'll quickly learn that doing this in C isn't all that fun.

Before we start, let's pick a data structure. We'll logically choose a **stack** as a good choice for this problem. There are two ways to implement stacks in C:

- As a static array, using a pointer to track where the top of the stack is.
- As a linked list, using a pointer to the tail node to indicate the topo of the stack.

For this solution, I chose to use a static array. *This is a less space-efficient solution*. I challenge you to rework this solution to use a linked list instead.

We will complete this challenge in two parts. First, we will read the initial data and store it in our stack. Then, we will read the instructions and update our stack accordingly.

Part 1: Reading the Initial Data

We'll start with the same initializations as the previous challenges. We'll open the file, ensure it's opened, and then initalized our read buffer.

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

// prep the input
char* line = NULL;
ssize_t read;
size_t len = 0;
```

We will use the same `getline` loop to read the data. When we read the stack data, we are not sure which stacks have data on each line, so it's easier to parse it knowing the entire line.

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

The first thing we must do is determine the number of stacks. At the same time, we will track the last line of stack data so we know how much to parse when we initialize the stacks.

Based on the input, the line containing the stack numbers begins with a space. We will read until our first character is a space. Once we have this line, we will go to the end of the line and read off the last number. This will be the number of stacks.

```
// read until the newline
size_t des_line = 0;
int num_stacks;
while ((read = getline(&line, &len, fp)) != 1) {
    // our desired line guaranteed starts with a space
    if (line[0] == ' ') {
        // get the number of stacks
        size_t num_idx = 0;
        char* num = malloc(3 * sizeof(char));
        for (size_t idx = read-4; line[idx] != ' '; --idx)
            num[num_idx++] = line[idx];
        num_stacks = atoi(num);
        break;
    } else {
        ++des_line;
    }
}
```

Wen define `des_line` as the number of lines containing stack data. We do this so that when we return to read the lines of data, we know how many lines to read.

Since we aren't sure the size of the number, we have to simply read from the last number to the space. This isn't the best solution, but it works. A plausible second solution would be to use `strtok()` to get each line number. Here is that solution:

```
size_t des_line = 0;
int num_stacks;
while ((read = getline(&line, &len, fp)) != 1) {
    // our desired line guaranteed starts with a space
    if (line[0] == ' ') {
        char* token = strtok(line, " ");
        while ((token = strtok(NULL, " ")) != NULL && token[0] != 0xd)
            num_stacks = atoi(token);
        break;
    } else {
```

```
            ++des_line;
        }
    }
```

> *Let's talk this solution*. This solution uses `strtok()` to delimit the stack numbers. However, when we reach the end of the line, we'll notice that we hit a `0xd`. This is a **carriage return**. This is a special character used to delimit new lines. For most modern processing, newlines can be represented as `0xd0xa` or `\r\n`. This is because the original ASCII standard used `0xd` to represent newlines. This is why we have to check for `0xd` in our `strtok()` loop.
>
> This is not an intuitive solution. I originally tried checking `token[0] == \n` and it took me a minute to debug why this solution didn't work.

How that we have the number of stacks, we need to re-read the start of the file until we reach this line. First, we need to reset the file pointer to the start of the file.

```
fseek(fp, 0, SEEK_SET);
```

> *Note*: This can also be doing by simply reopening the file.
>
> ```
> fp = fopen("input.txt", "r");
> ```
>
> This is not a *wrong* solution; `fseek()` is more efficient.

Next, we should allocate the storage. We need `num_stacks` stacks. Since we chose to use a static array, we are going to define a size for each stack. I chose `100`.

```
// allocate the storage
char** stacks = malloc(num_stacks * sizeof(char*));
for (size_t i = 0; i < num_stacks; ++i) {
    stacks[i] = malloc(100 * sizeof(char));
}
```

At the same time, we also must track the number of crates on each stack. We will use a second array to track this.

```
size_t* size = malloc(num_stacks * sizeof(size_t));
```

Once we do this, we need to read the data. We defined `des_line` earlier, which we tracked as the number of lines containing stack data. We will read lines until we reach this number of lines

```

```

```
size_t line_num = 0;
while (line_num != des_line) {
    // get the line
    read = getline(&line, &len, fp);

    /* process the line here... */

    ++line_num;
}
```

For each line, we need to parse the data. We can actually take advantage of the data format to figure out what column we're in.

- Based on the character number we're at, we will be on stack `1 + char_num / 4`. This is because each stack item is four characters long (two brackets, one letter, and one space).
- We can check each character to see if it's alphanumeric. If it is, `char_num / 4` will tell us what column we're in. *Note: I removed the `1 +` because our array uses zero-based indexing, while the stack numbers are one-based.*
- We can simply increment the size of the stack as we increment using the post-increment operator.

Let's make this happen.

```
size_t line_num = 0;
while (line_num != des_line) {
    // get the line
    read = getline(&line, &len, fp);

    // read across the line looking for chars
    for (size_t i = 0; i < read; ++i) {
        if (line[i] >= 'A' && line[i] <= 'Z') {
            if (i/4 < num_stacks) {
                stacks[i/4][size[i/4]++] = line[i];
            } else {
                printf("Out of bounds error on line %zu\n", line_num);
                exit(EXIT_FAILURE);
            }
        }
    }

    ++line_num;
}
```

We now have the initial data stored. **However**, it's not quite right. We read the stacks from top to bottom, meaning that the top of the stacks are at the bottom. We need to reverse each stack to fix this.

```
// we put them in backwards, reverse the lists
for (size_t i = 0; i < num_stacks; ++i) {
    char* rev_stack = malloc(100 * sizeof(char));
```

```
    for (size_t j = 0; j <= size[i]; ++j)
        rev_stack[j] = stacks[i][size[i]-j-1];
    free(stacks[i]);
    stacks[i] = rev_stack;
}
```

We're almost done! All we need to do is move the `FILE*` pointer to the start of the instructions. We know this is two lines away (the stack numbers line plus the newline).

```
// push the line twice to get to the right spot
getline(&line, &len, fp);
getline(&line, &len, fp);
```

Now, our data is initialized. We can move on to parsing the instructions.

Part 2: Parsing the Instructions

Thankfully, this is the easy part. We can easily read the rest of the lines and get the data we need.

We need three unsigned integers: the number of moving items, the source stack, and the destination stack. We can use `strtok()` to get these values. We can ignore the strings in between, but can't forget to read them.

```
 size_t moving, origin, destination;
while (getline(&line, &len, fp) != -1) {
    strtok(line, " "); // Moving

    size_t moving = atoi(strtok(NULL, " "));
    strtok(NULL, " "); // from

    size_t origin = atoi(strtok(NULL, " ")) - 1;
    strtok(NULL, " "); // to

    size_t destination = atoi(strtok(NULL, " ")) - 1;

    /* Move the items between stacks */
}
```

How do we move the items between stacks? We need to move them *in reverse order*. We can use a `for` loop to count from `0` to `moving` and move them from the old stack to the new stack. At the same time, we can post-decrement `size[destination]` and `size[origin]`.

We can do this in a nice ~~ugly~~ one-liner.

```
for (size_t action = 0; action < moving; ++action) {
    stacks[destination][size[destination]++] = stacks[origin]
```

```
[(size[origin]--)-1];
}
```

Printing the Flag

Now that the data has been processed, we can print the flag. The flag is simply the tops of each stacks. We can use a nice `for` loop to handle this:

```
// get the top stacks
printf("flag{");
for (size_t i = 0; i < num_stacks; ++i)
    printf("%c", stacks[i][size[i]-1]);
printf("}\n");
```

Don't forget to free the memory! We allocated memory for the stacks, the sizes, and the line buffer. We need to free all of these.

```
// free the memory
for (size_t i = 0; i < num_stacks; ++i)
    free(stacks[i]);
free(stacks);
free(size);
if (line) free(line);
```

Running this prints our flag.

> *A note on this solution*: You'll notice that if you attempt to print the stacks to watch the program run, it will not work as expected. This is because I chose to use a static array and a pointer to watch the top of the stack. In doing so, I neglect to clear the data that I no longer need. This means that the data is still there, but it's not being tracked. This is why the flag is correct, but the output is not.
>
> *A linked list solution mitigates this problem.*

Full Solution

The full solution is below:

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

int main(void)
{
    // open the file
    FILE* fp = fopen("input.txt", "r");
    if (fp == NULL)
```

```
        exit(EXIT_FAILURE);

    // prep the input
    char* line = NULL;
    ssize_t read;
    size_t len = 0;

    // read until the newline
    size_t des_line = 0;
    int num_stacks;
    while ((read = getline(&line, &len, fp)) != 1) {
        // our desired line guaranteed starts with a space
        if (line[0] == ' ') {
            char* token = strtok(line, " ");
            while ((token = strtok(NULL, " ")) != NULL && token[0] != 0xd)
               num_stacks = atoi(token);
            break;
        } else {
            ++des_line;
        }
    }

    // this resets the file to start reading from the beginning
    fseek(fp, 0, 0);

    // allocate the storage
    char** stacks = malloc(num_stacks * sizeof(char*));
    for (size_t i = 0; i < num_stacks; ++i) {
        stacks[i] = malloc(100 * sizeof(char));
    }
    size_t* size = malloc(num_stacks * sizeof(size_t));

    // now read the first 8 lines and get the data
    size_t line_num = 0;
    while (line_num != des_line) {
        // get the line
        read = getline(&line, &len, fp);

        // read across the line looking for chars
        for (size_t i = 0; i < read; ++i) {
            if (line[i] >= 'A' && line[i] <= 'Z') {
                if (i/4 < num_stacks) {
                    stacks[i/4][size[i/4]++] = line[i];
                } else {
                    printf("Out of bounds error on line %zu\n", line_num);
                    exit(EXIT_FAILURE);
                }
            }
        }

        ++line_num;
    }

    // we put them in backwards, reverse the lists
```

```
    for (size_t i = 0; i < num_stacks; ++i) {
        char* rev_stack = malloc(100 * sizeof(char));
        for (size_t j = 0; j <= size[i]; ++j)
            rev_stack[j] = stacks[i][size[i]-j-1];
        free(stacks[i]);
        stacks[i] = rev_stack;
    }

    // push the line twice to get to the right spot
    getline(&line, &len, fp);
    getline(&line, &len, fp);

    // now we need to parse the rest of the lines
    size_t moving, origin, destination;
    while (getline(&line, &len, fp) != -1) {
        strtok(line, " "); // Moving

        size_t moving = atoi(strtok(NULL, " "));
        strtok(NULL, " "); // from

        size_t origin = atoi(strtok(NULL, " ")) - 1;
        strtok(NULL, " "); // to

        size_t destination = atoi(strtok(NULL, " ")) - 1;

        // perform the action
        for (size_t action = 0; action < moving; ++action) {
            stacks[destination][size[destination]++] = stacks[origin]
[(size[origin]--)-1];
        }
    }

    // get the top stacks
    printf("flag{");
    for (size_t i = 0; i < num_stacks; ++i)
        printf("%c", stacks[i][size[i]-1]);
    printf("}\n");

    // free the memory
    for (size_t i = 0; i < num_stacks; ++i)
        free(stacks[i]);
    free(stacks);
    free(size);
    if (line) free(line);

    return 0;
}
```