COMP1511 Week 7

char/string functions, command line arguments and struct pointers

Joanna Lin

What we'll cover

more C functions

- char functions: **`getchar`** and **`putchar`**.
- string function: **`fgets`**
- using the **`man`** pages to jog our memories of how functions work.

command line arguments

- how to access them

struct pointers

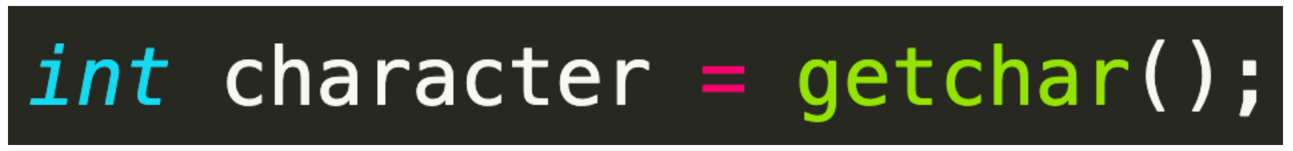
- **`->`** syntax

```
typedef struct s{
    struct s *finger;
} SpiderMan;
int main(void){
    SpiderMan* A = new SpiderMan;
    SpiderMan* B = new SpiderMan;
    A->finger = B;
    B->finger = A;
}
```

`getchar` and `putchar`

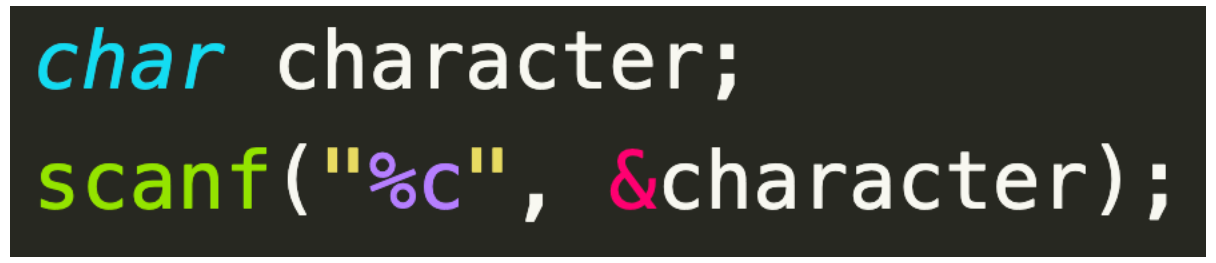
more `stdio.h` functions

We use an integer to store the return value of `getchar` because on some systems characters range from 0 to 255. However, getchar returns `EOF` (which is `-1`) to indicate the end of input.

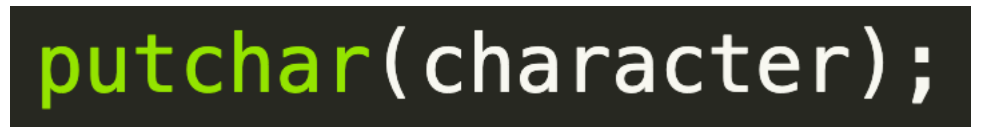


```
int character = getchar();
```

works like

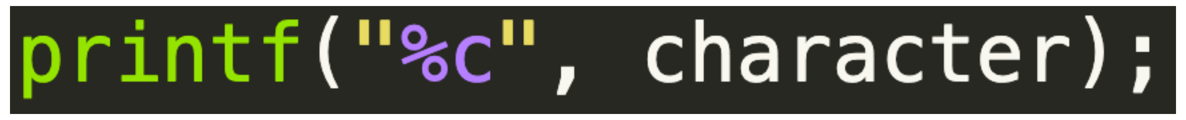


```
char character;
scanf("%c", &character);
```



```
putchar(character);
```

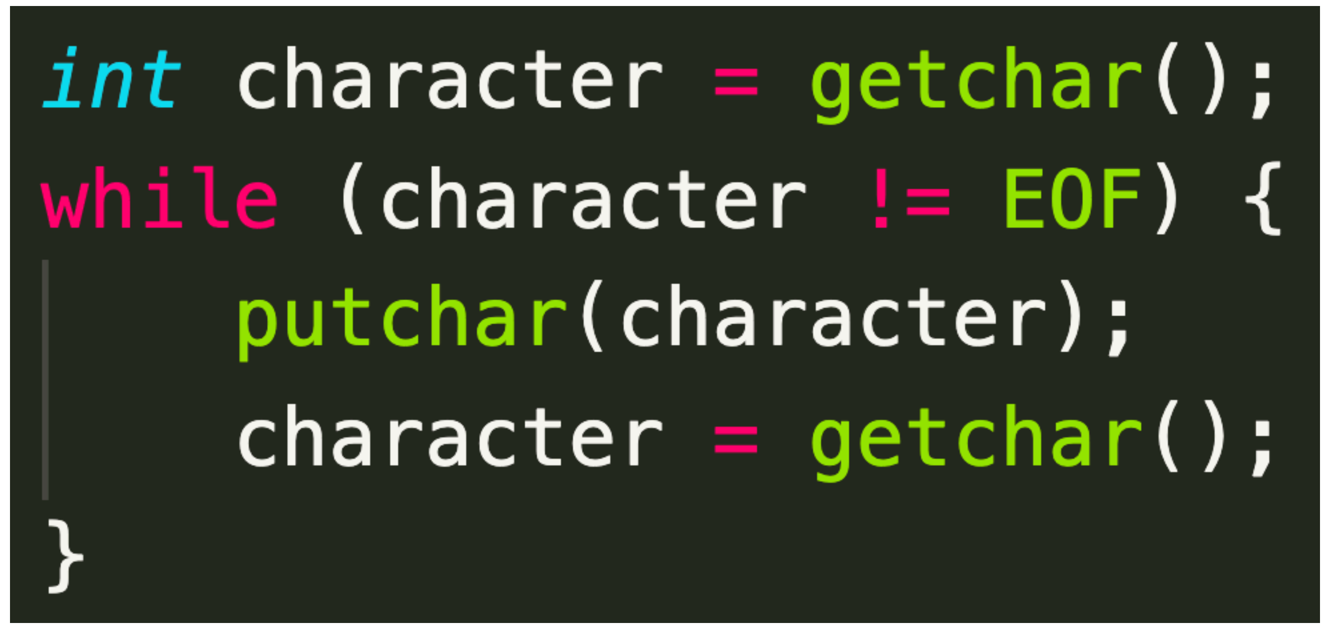
works like



```
printf("%c", character);
```

- Using `getchar` in a loop:

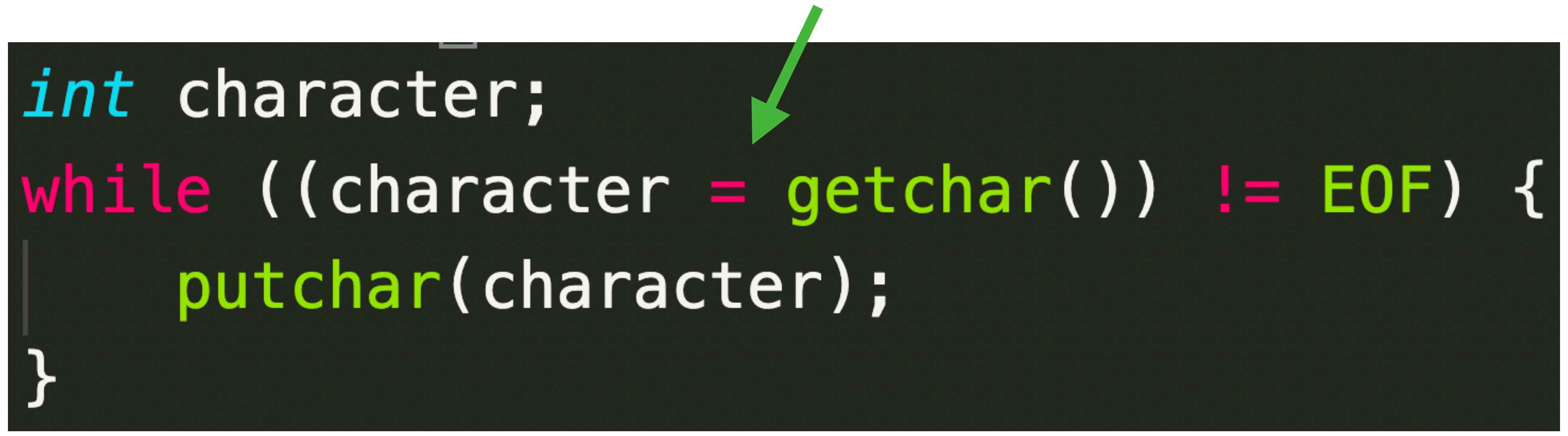
Program that 'echo' characters the user inputs



```
int character = getchar();
while (character != EOF) {
    putchar(character);
    character = getchar();
}
```

can be condensed to

Recall that an assignment statement evaluates to the right hand side!

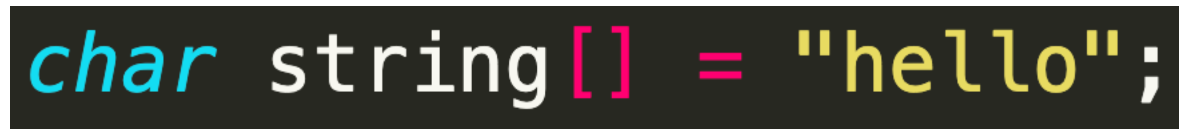


```
int character;
while ((character = getchar()) != EOF) {
    putchar(character);
}
```

strings

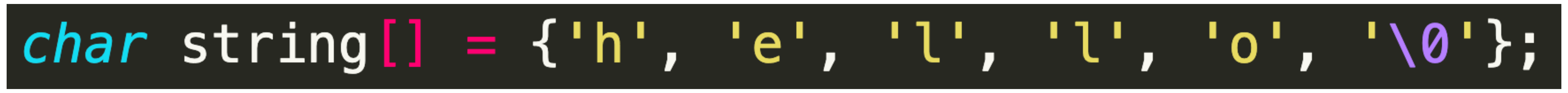
array of characters

- in C, a string is an array of characters



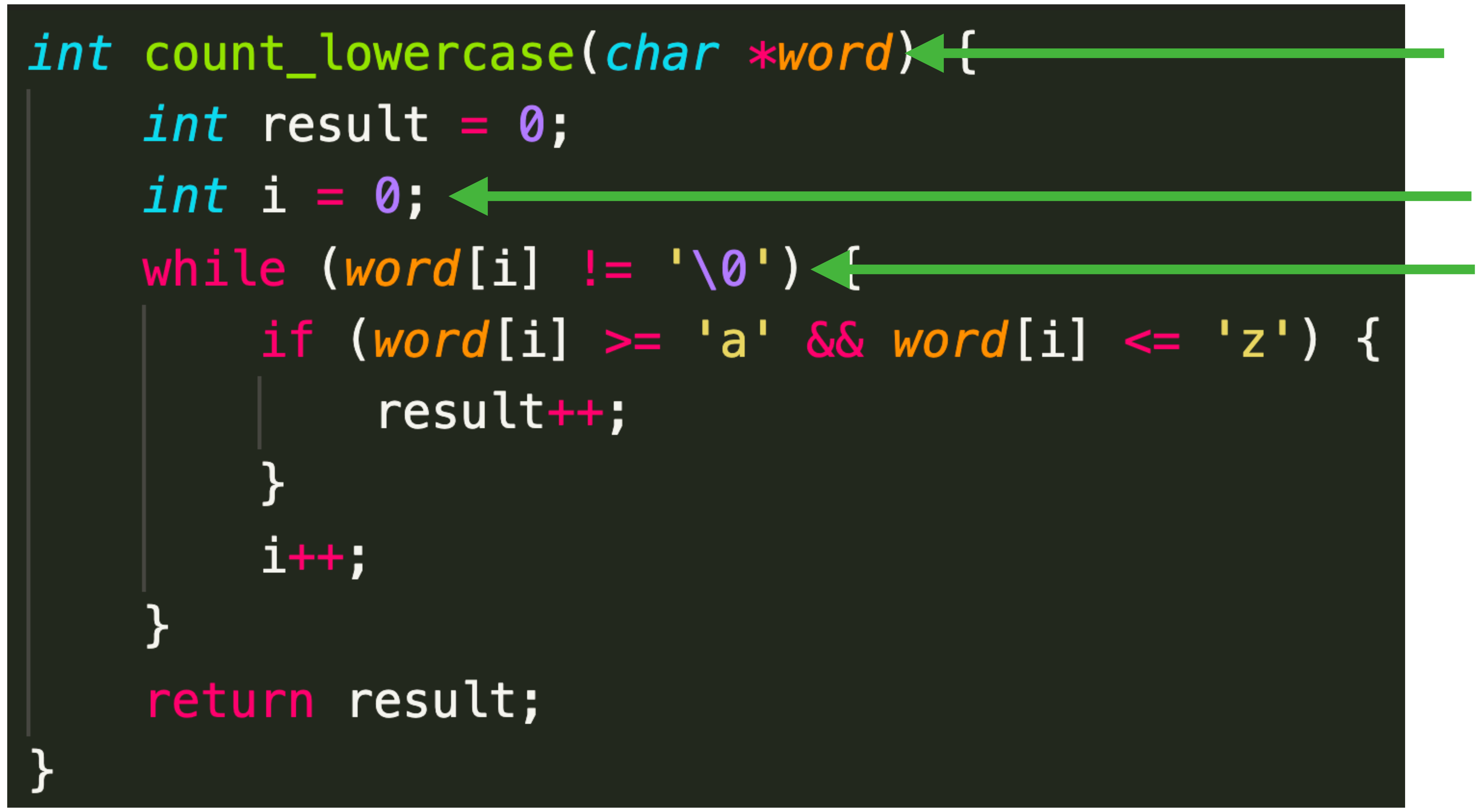
```c
char string[] = "hello";
```

is shorthand for



```c
char string[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

- the array stores all the expected characters, plus a null terminator `'\0'` at the end
  - the null terminator has ASCII value 0.
- the null terminator allows us to write the condition on a **`while`** loop when looping through a string.



```c
int count_lowercase(char *word) {
    int result = 0;
    int i = 0;
    while (word[i] != '\0') {
        if (word[i] >= 'a' && word[i] <= 'z') {
            result++;
        }
        i++;
    }
    return result;
}
```

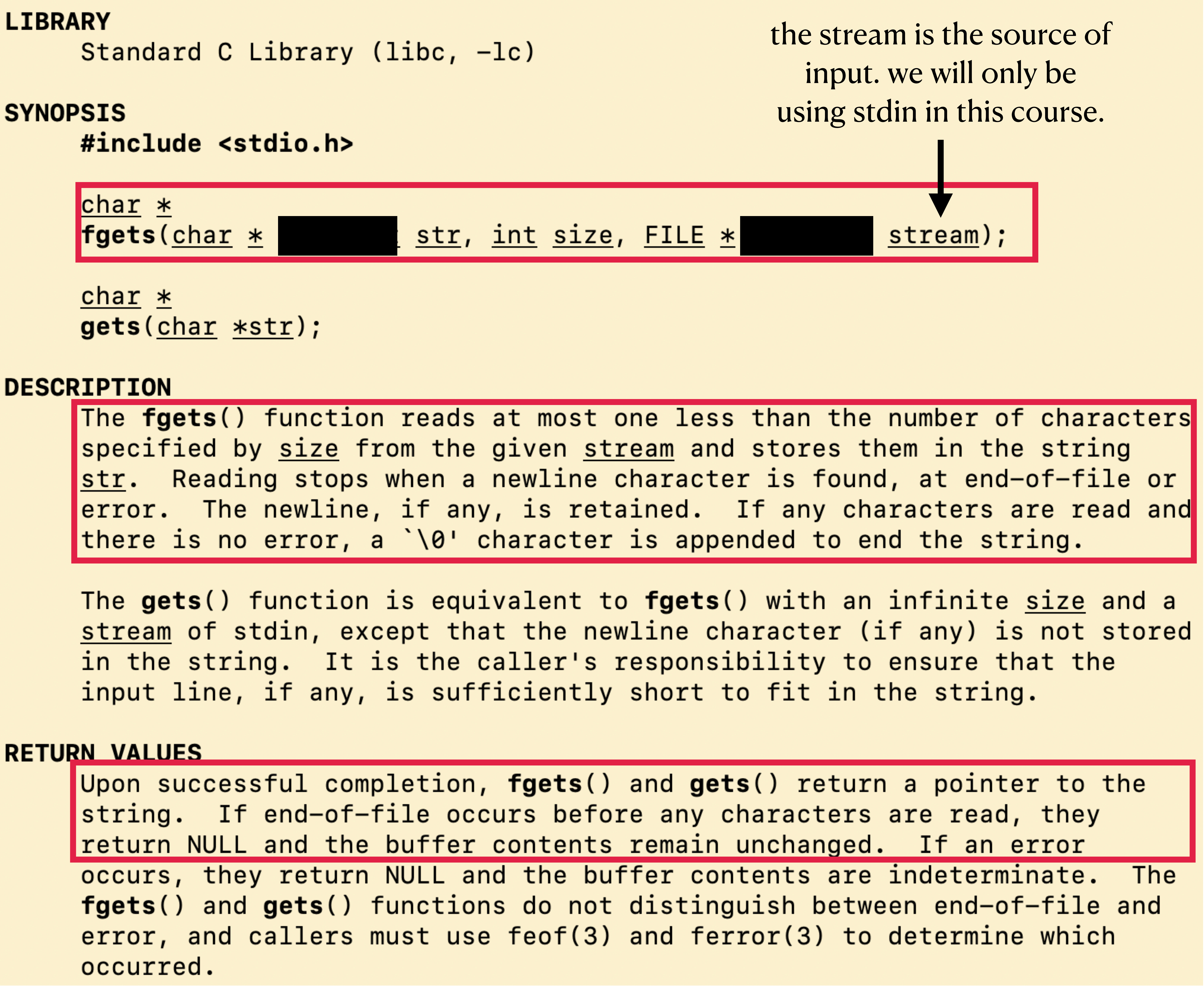
like arrays, we can pass in a char array into a char pointer parameter

we still use our regular loop counter variable to index the array....

... but instead of the condition `i < ...`, we check whether the character at the current index is `'\0'`

fgets

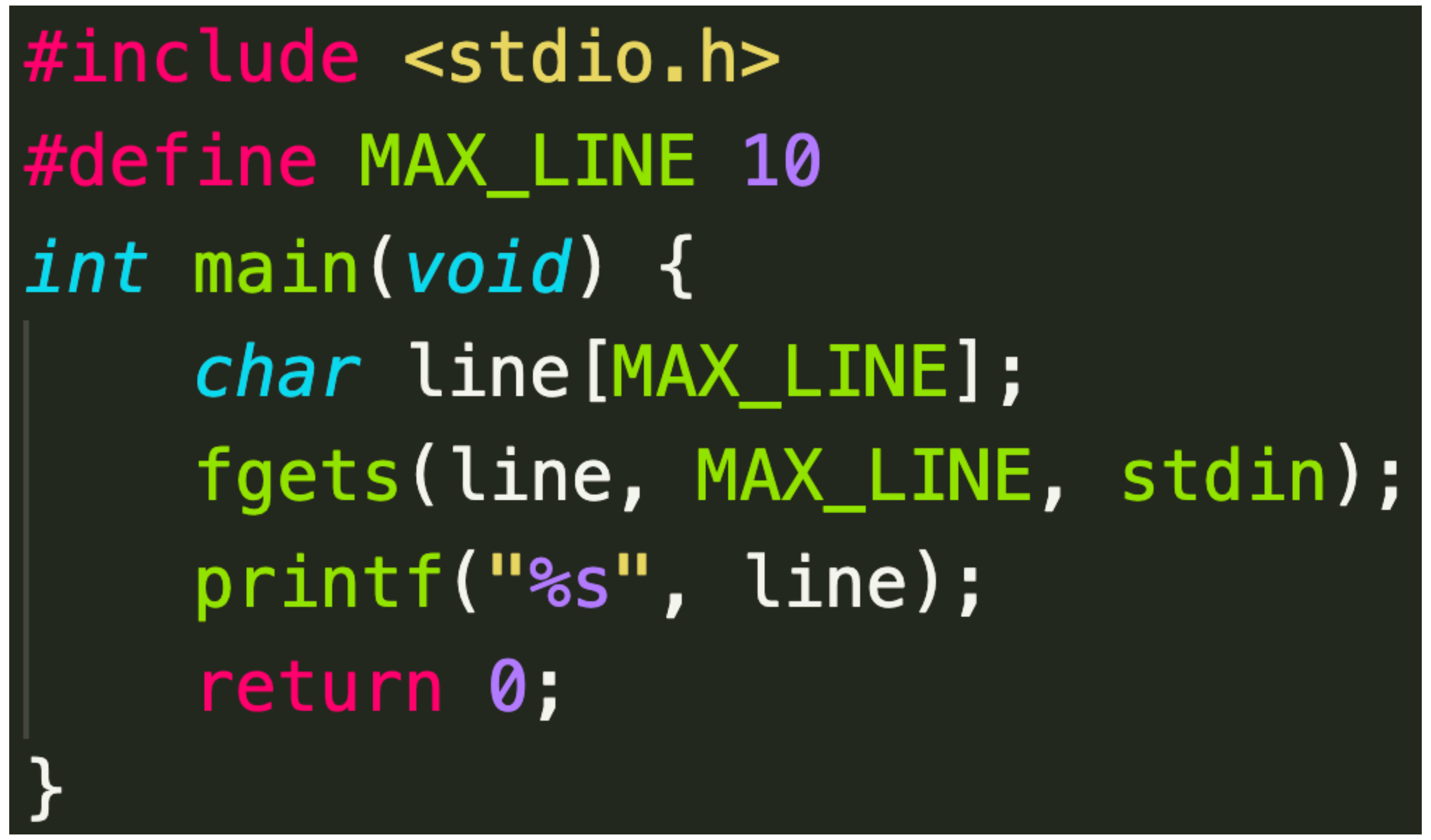
inspecting the man pages



```
LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdio.h>

     char *
     fgets(char * [redacted] str, int size, FILE * [redacted] stream);

     char *
     gets(char *str);

DESCRIPTION
     The fgets() function reads at most one less than the number of characters
     specified by size from the given stream and stores them in the string
     str.  Reading stops when a newline character is found, at end-of-file or
     error.  The newline, if any, is retained.  If any characters are read and
     there is no error, a `\0' character is appended to end the string.

     The gets() function is equivalent to fgets() with an infinite size and a
     stream of stdin, except that the newline character (if any) is not stored
     in the string.  It is the caller's responsibility to ensure that the
     input line, if any, is sufficiently short to fit in the string.

RETURN VALUES
     Upon successful completion, fgets() and gets() return a pointer to the
     string.  If end-of-file occurs before any characters are read, they
     return NULL and the buffer contents remain unchanged.  If an error
     occurs, they return NULL and the buffer contents are indeterminate.  The
     fgets() and gets() functions do not distinguish between end-of-file and
     error, and callers must use feof(3) and ferror(3) to determine which
     occurred.
```

the stream is the source of input. we will only be using stdin in this course.

program that echoes the string a user inputs



```c
#include <stdio.h>
#define MAX_LINE 10
int main(void) {
    char line[MAX_LINE];
    fgets(line, MAX_LINE, stdin);
    printf("%s", line);
    return 0;
}
```

case 1: user enters “**hello**” into terminal, and presses enter.

line will store **{'h','e','l','l','o','\n','\0'}***

case 2: user enters “**hello**” into the terminal, and presses ctrl+d

line will store **{'h','e','l','l','o','\0'}***

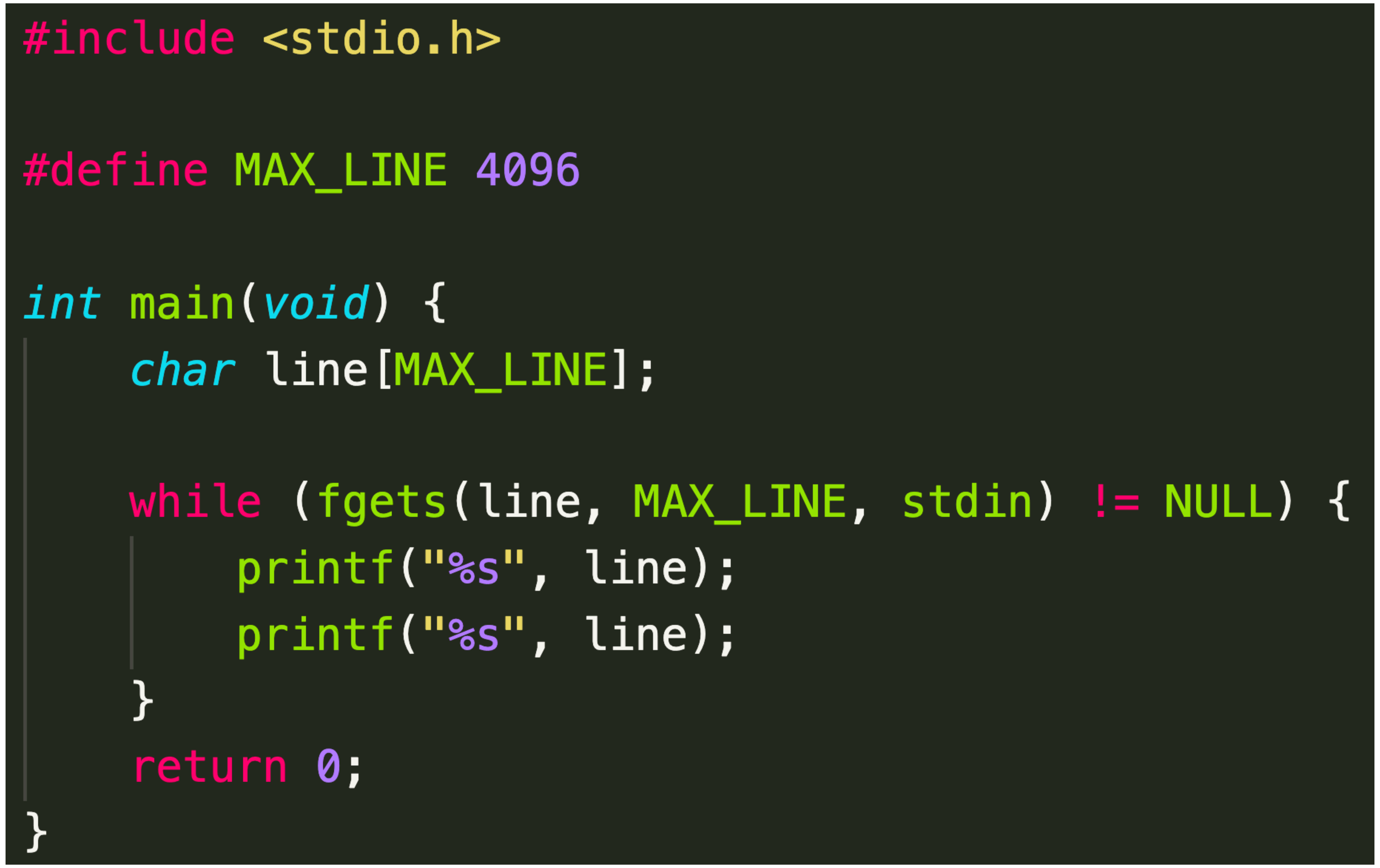
case 3: user enters “**hello world**” into the terminal, then presses either enter or ctrl+D

line will store
{'h','e','l','l','o',' ','w','o','r','\0'}

fgets

usage in a loop

program that echoes the string a user inputs twice, in a loop



```c
#include <stdio.h>

#define MAX_LINE 4096

int main(void) {
    char line[MAX_LINE];

    while (fgets(line, MAX_LINE, stdin) != NULL) {
        printf("%s", line);
        printf("%s", line);
    }
    return 0;
}
```

every time the while loop condition is evaluated, the **`fgets`** function is executed

recall from the **`man`** pages that **`fgets`** returns **`NULL`** if an empty line is read — this will signal the end of the user's input, and so we use it as our loop condition

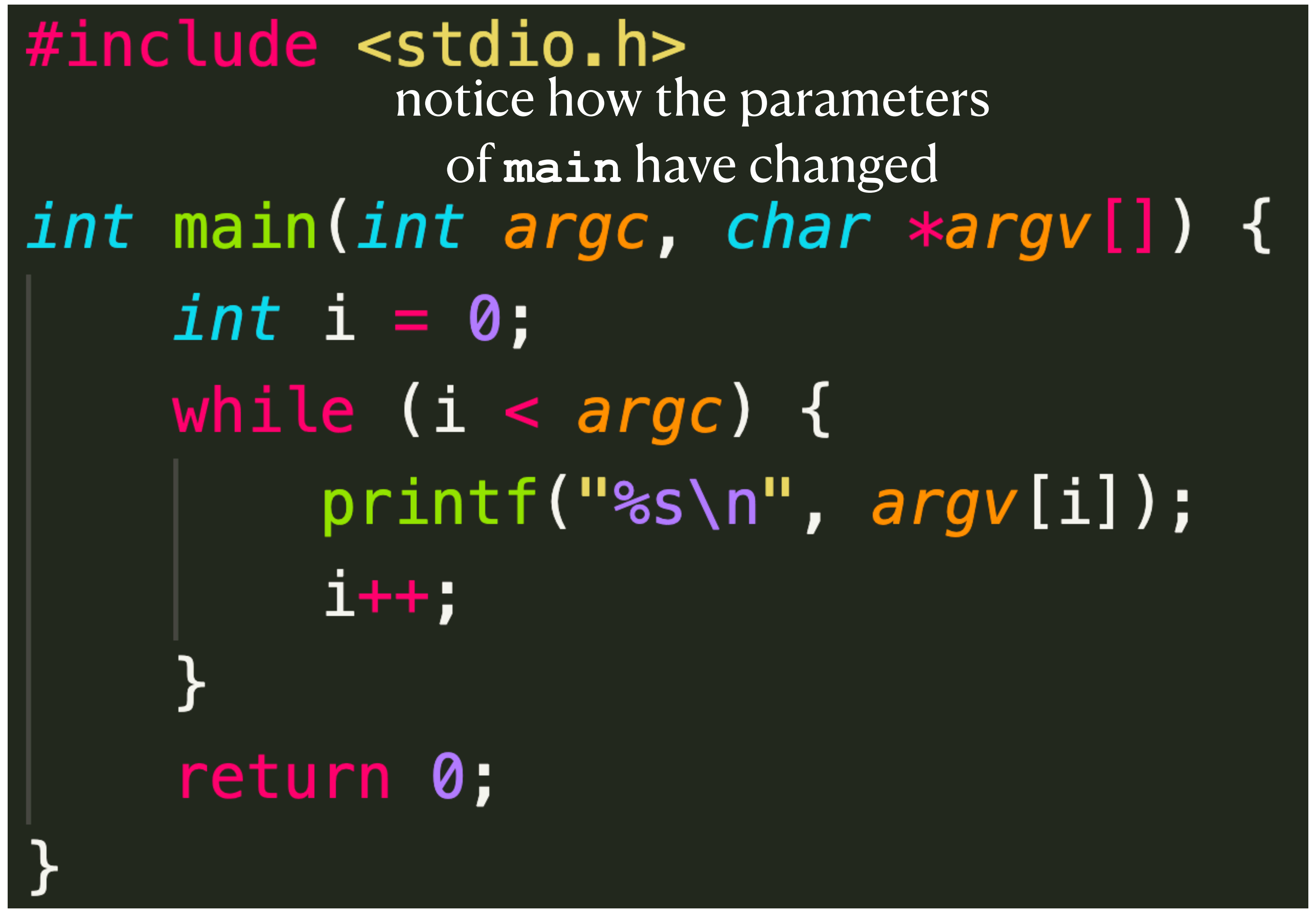
Command Line Arguments

- the user can supply some arguments when running the program.
- for example: **`./program hello world!`**
- to access what the user enters into the command line, we change the **`main`** function signature from **`int main(void)`** to **`int main(int argc, char *argv[])`**
  - **`argc`** stores the count of the command line arguments (the length of **`argv`**)
  - **`argv`** stores the command line arguments as an array of strings
  - **`argv[0]`** is always the name of the program.
- in the above example, **`argc`** would be 3 and **`argv`** would store the strings "**`./program`**", "**`hello`**" and "**`world!`**" in that order.

Command Line Arguments

array of strings... or 2D array?

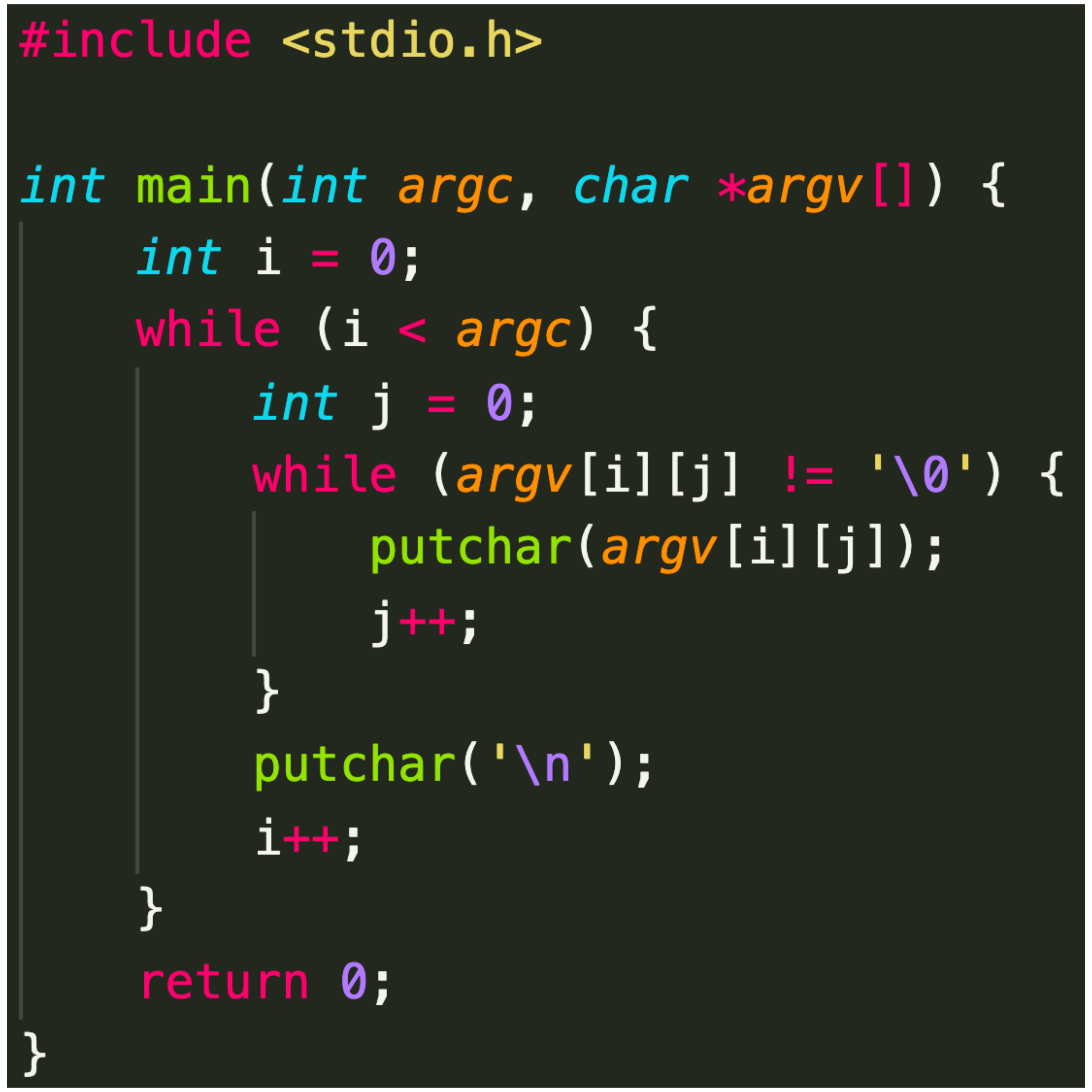
use **`argc`** to write the condition on the loop when looping through **`argv`**



```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 0;
    while (i < argc) {
        printf("%s\n", argv[i]);
        i++;
    }
    return 0;
}
```

notice how the parameters of **`main`** have changed

does the same thing as



```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 0;
    while (i < argc) {
        int j = 0;
        while (argv[i][j] != '\0') {
            putchar(argv[i][j]);
            j++;
        }
        putchar('\n');
        i++;
    }
    return 0;
}
```

loop through each character in each string if you want to have finer-grain control.

Struct Pointers

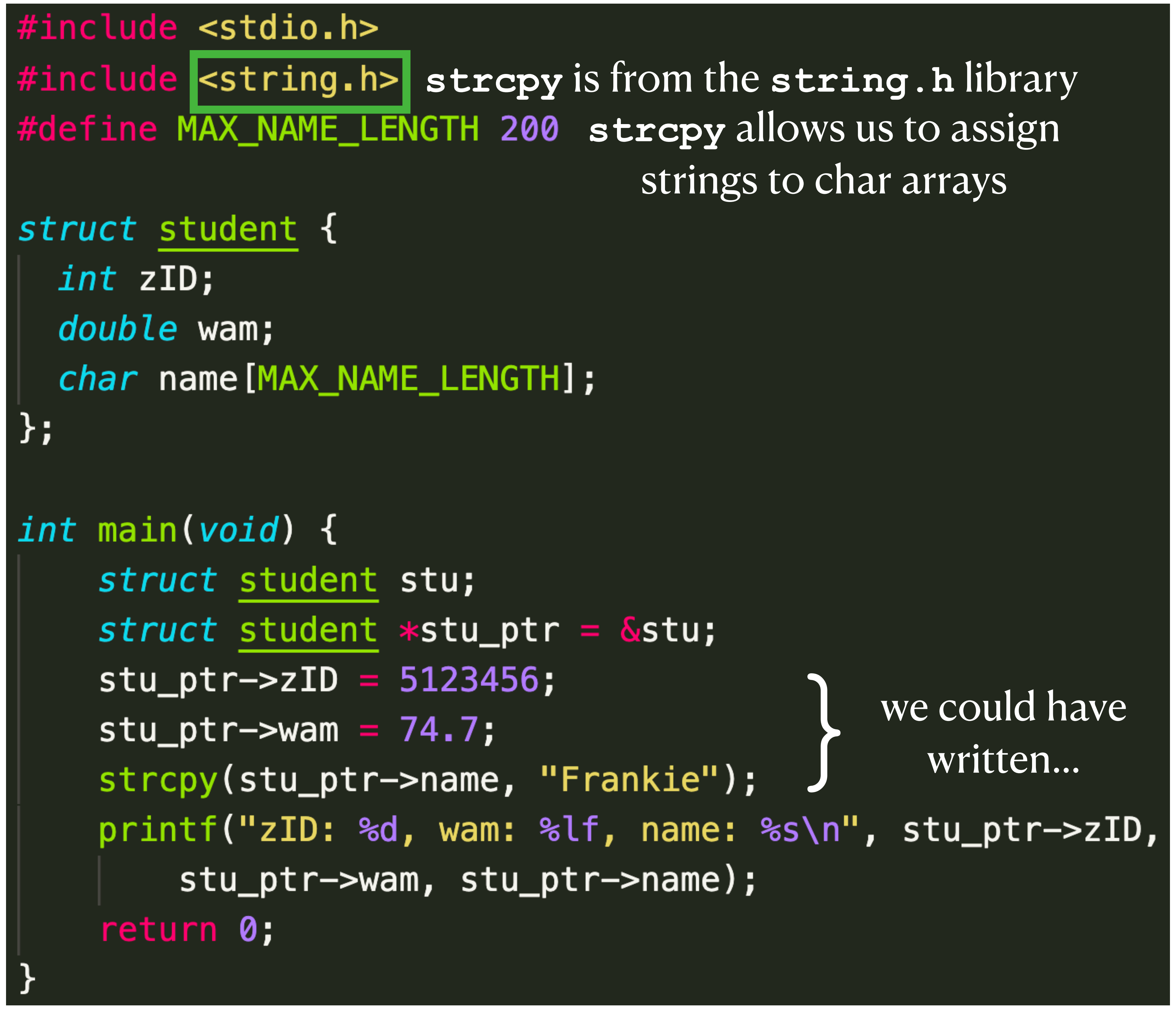
First Look

- They work like any other pointer.
  - we declare it by adding a **`*`** at the end of the struct type
  - e.g. **`struct student *student_p;`**
  - struct pointers store memory addresses of structs.
  - we can dereference the pointer (change the struct at the memory address).
- C has syntactic sugar for accessing fields of the struct at the stored memory address.
  - Instead of **`(*student_p).field_name`**, we can write **`student_p->field_name`**.

Struct Pointers

syntax

remember: whatever goes on the left of a '**->**' must be a struct pointer!

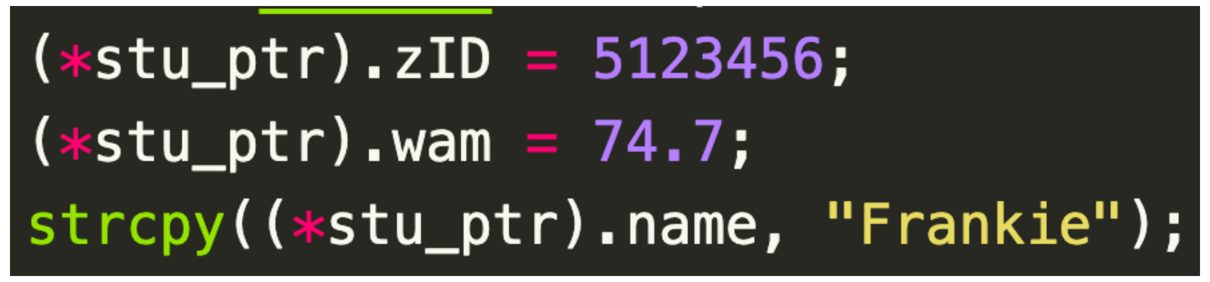


```
#include <stdio.h>
#include <string.h>
#define MAX_NAME_LENGTH 200

struct student {
  int zID;
  double wam;
  char name[MAX_NAME_LENGTH];
};

int main(void) {
    struct student stu;
    struct student *stu_ptr = &stu;
    stu_ptr->zID = 5123456;
    stu_ptr->wam = 74.7;
    strcpy(stu_ptr->name, "Frankie");
    printf("zID: %d, wam: %lf, name: %s\n", stu_ptr->zID,
        stu_ptr->wam, stu_ptr->name);
    return 0;
}
```

`strcpy` is from the **`string.h`** library

`strcpy` allows us to assign strings to char arrays

we could have written...



```
(*stu_ptr).zID = 5123456;
(*stu_ptr).wam = 74.7;
strcpy((*stu_ptr).name, "Frankie");
```