Strings

COP 3223C – Introduction to Programming with C

Fall 2025

Yancy Vance Paredes, PhD

Discussion /1

- Our programs mostly processed **numbers**

- What if we want to write programs that can process **“words”**

Discussion /2

2

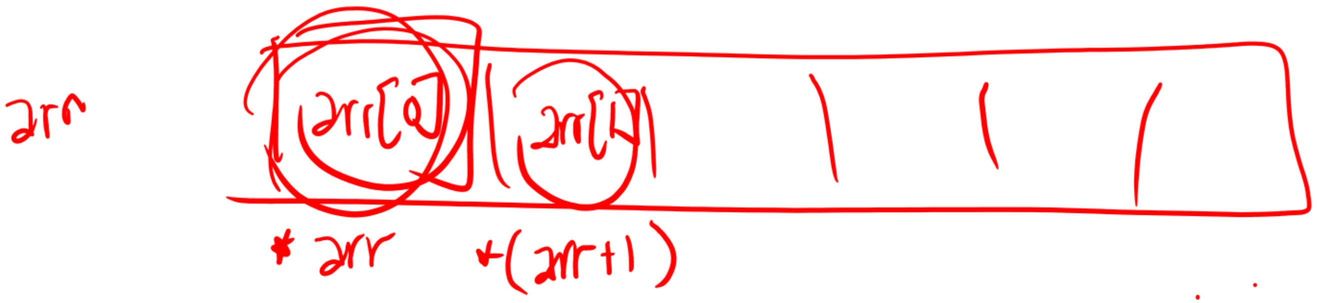
'a' 'A' '2' '!' ' '

'\n'

*arr+1

arr[0] +1

- What is a "word"?
- A more generic term is **string**; it is just a group of characters
- Since characters follow a particular order, it is a *list of characters*
- Formally, it is an **array of characters**



String /1

format string

"%d"

"hello"

3.0

- In C, a string is simply an array of **`char`** (i.e., characters)
- Unfortunately, it is **not a datatype**
- Typically enclosed in `"   "` (double quotes)

String /2

- For example, let’s visualize the **string literal `"Hello"`**

- But how?

- Recall, because it is an array, we need to **know the size**

```
char word[10] = "Hello";
```

'H'	'e'	'l'	'l'	'o'					

String /3

We assume that we want to store a string with at most 10 **`char`**s

char word[10] = "Hello";

i	0	1	2	3	4	5	6	7	8	9
`word[i]`	'H'	'e'	'l'	'l'	'o'	'\0'				

null character

String /4

How do we **store** the string literal in our array?

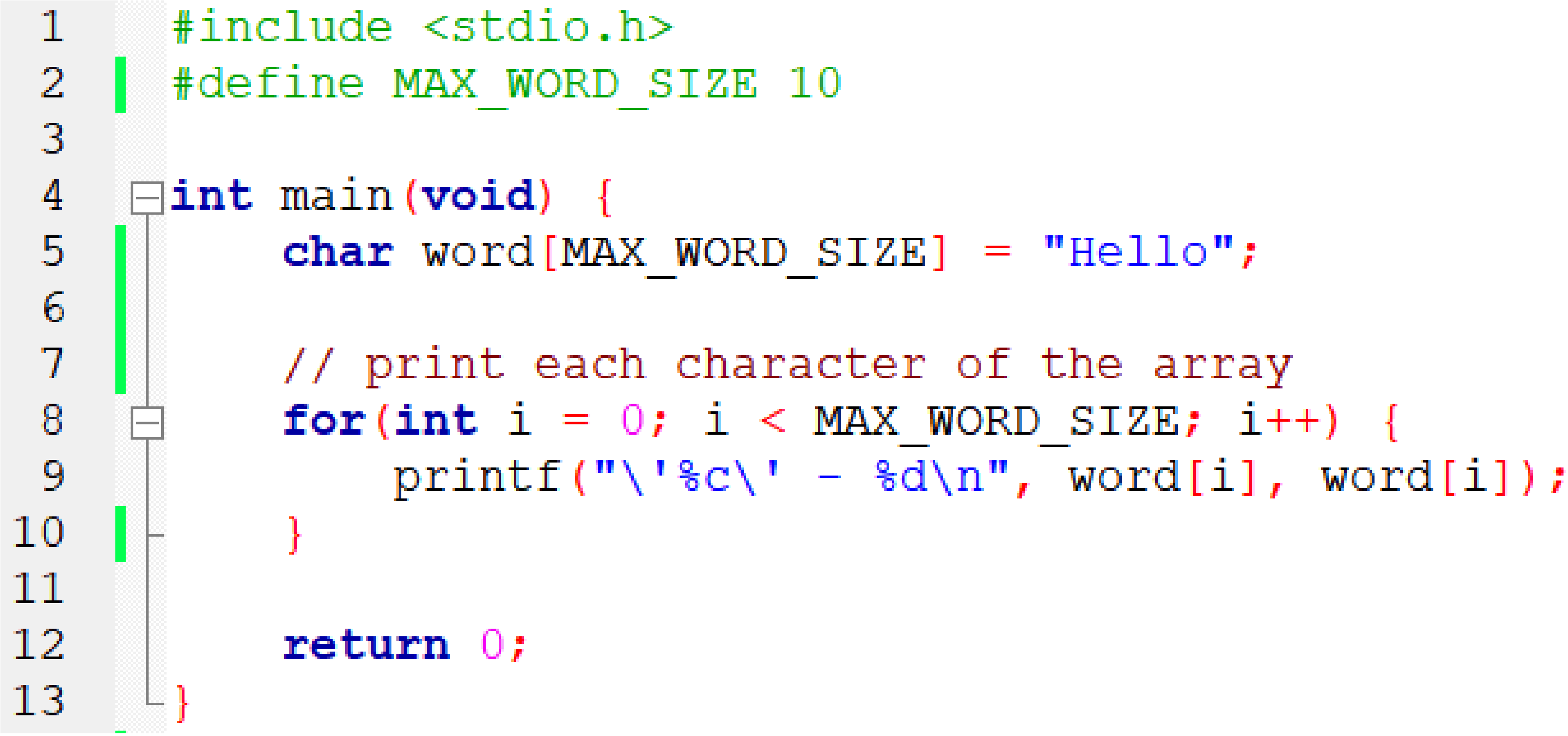
null-terminated

char word[10] = "Hello";

i	0	1	2	3	4	5	6	7	8	9
word[i]	'H'	'e'	'l'	'l'	'o'	'\0'	'\0'	'\0'	'\0'	'\0'

Practice

- Declare and initialize an array of character (both literal and list)
- Try using assignment operator to assign a string literal
- Print all the characters, one line at a time
- Understand what the characters are



```c
#include <stdio.h>
#define MAX_WORD_SIZE 10

int main(void) {
    char word[MAX_WORD_SIZE] = "Hello";

    // print each character of the array
    for(int i = 0; i < MAX_WORD_SIZE; i++) {
        printf("\'%c\' - %d\n", word[i], word[i]);
    }

    return 0;
}
```

Notes

- You can only perform the assignment of a string literal during **initialization**; otherwise, it won't work!

~~char word[10];~~

~~word = "Hello";~~

- What if we really have to?*

- We can also use an **initializer list** like when we were dealing with numbers (but…)

The `'\0'` or Null Character

- In C, string literals must be terminated by `'\0'` (**null-terminated**)
- What is its implication?
- Because it is also a valid character, it will take up one space!
- Therefore, our array earlier could only actually hold **9 characters**

Visualizing a String

```
char word[10] = "Hello";
```

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
word[i]	'H'	'e'	'l'	'l'	'o'	'\0'	'\0'	'\0'	'\0'	'\0'

Discussion

- Knowing that a string will always have the null character, what happens if we want to store `"basketball"`?
- It will cause an undefined behavior due to **buffer overflow**
- Avoid this situation!

Visualizing a String (Not Null-Terminated)

char word[10] = "basketball";

i	0	1	2	3	4	5	6	7	8	9
word[i]	'b'	'a'	's'	'k'	'e'	't'	'b'	'a'	'l'	'l'

Printing Strings

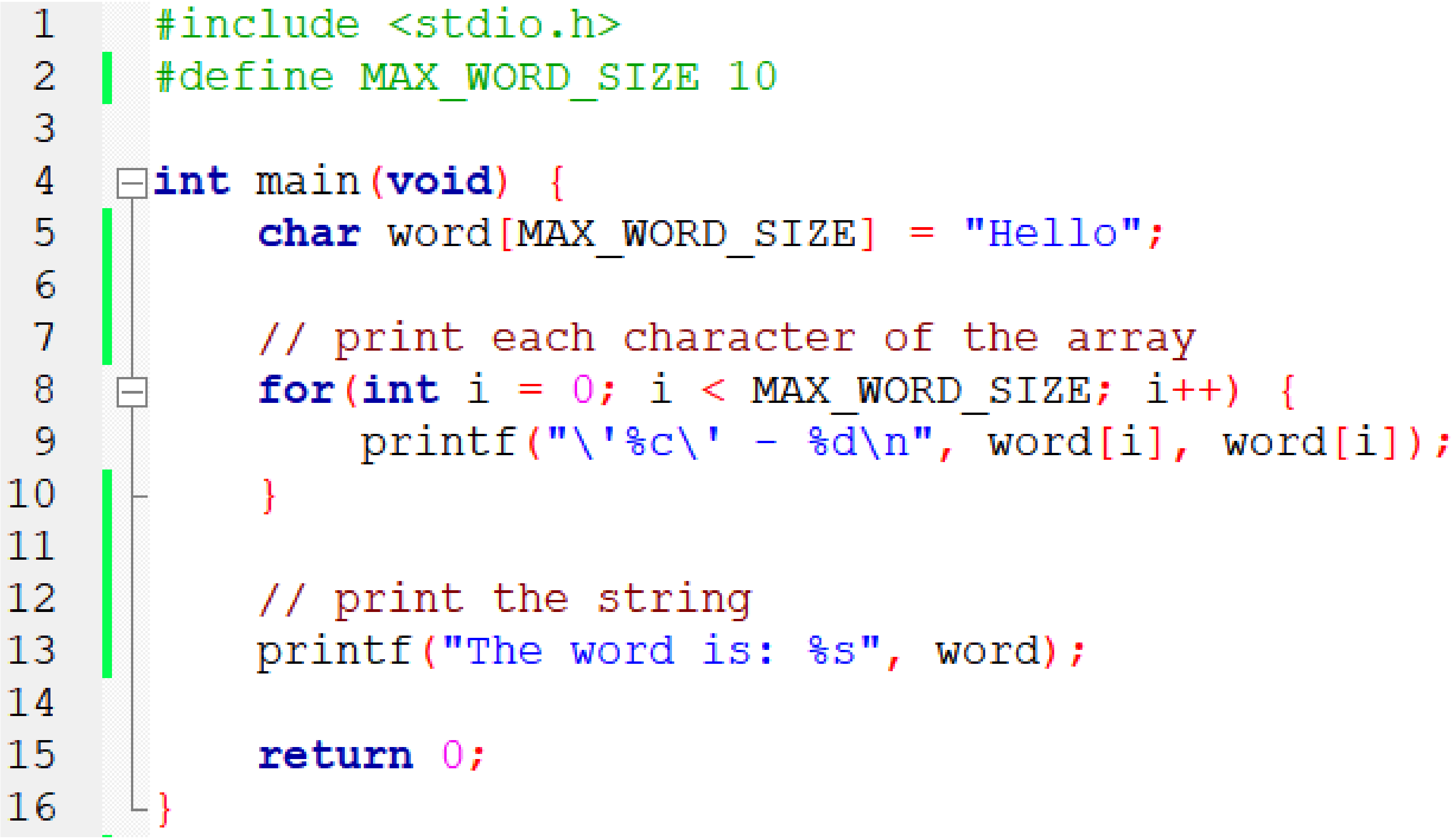
- Since it is an array, we can do something similar to an **`int`** array
- We can also simply call **`printf()`** right away and pass the array
- Strings have their own format specifier, which is **`%s`**

```
char word[10] = "hi";
printf("%s", word);
```

```
hi
␣
```

Practice

Print the string **`word`** that we have from earlier



```c
#include <stdio.h>
#define MAX_WORD_SIZE 10

int main(void) {
    char word[MAX_WORD_SIZE] = "Hello";

    // print each character of the array
    for(int i = 0; i < MAX_WORD_SIZE; i++) {
        printf("\'%c\' - %d\n", word[i], word[i]);
    }

    // print the string
    printf("The word is: %s", word);

    return 0;
}
```

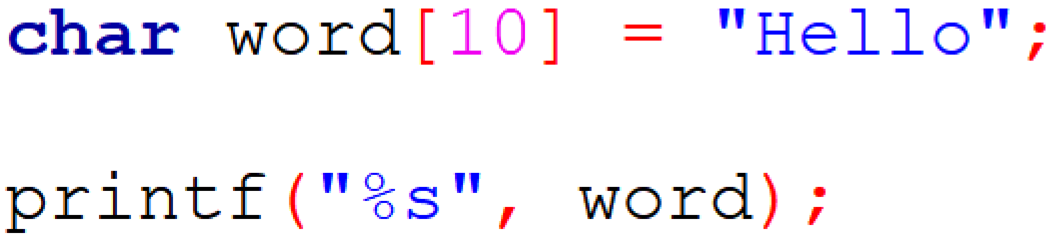
Notes

- We know that the array has 10 elements (or characters), however, why are we only seeing 5 characters?

- It only considers anything before the `'\0'`

'\0' != '_' != '\n'

Code Tracing /1

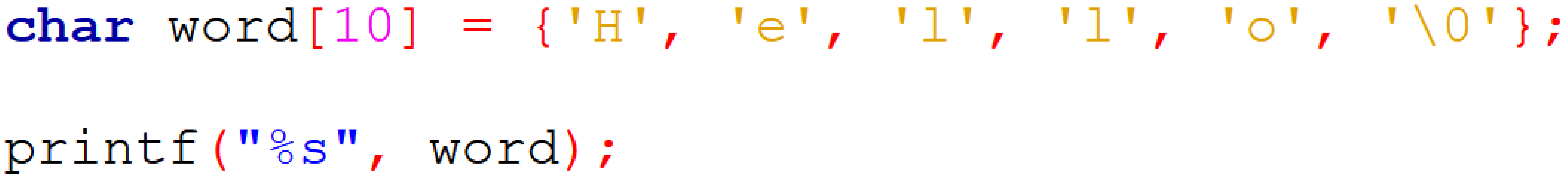


```
char word[10] = "Hello";

printf("%s", word);
```

Code Tracing /2

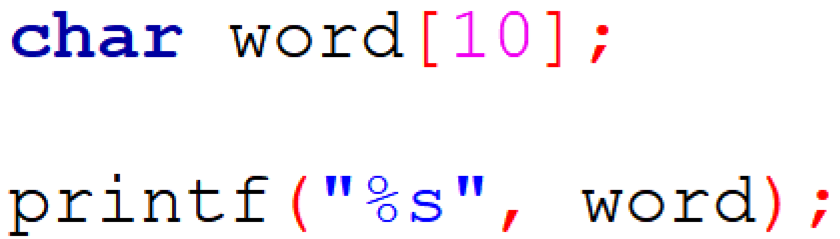
```c
char word[10] = {'H', 'e', 'l', 'l', 'o'};

printf("%s", word);
```

Code Tracing /3



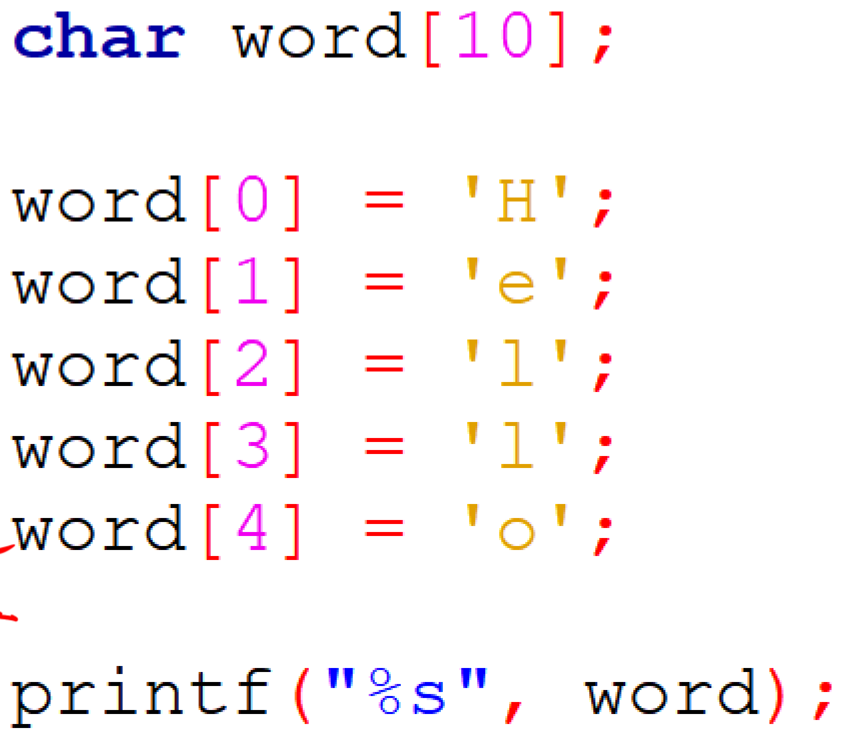
```
char word[10] = {'H', 'e', 'l', 'l', 'o', '\0'};

printf("%s", word);
```

Code Tracing /4



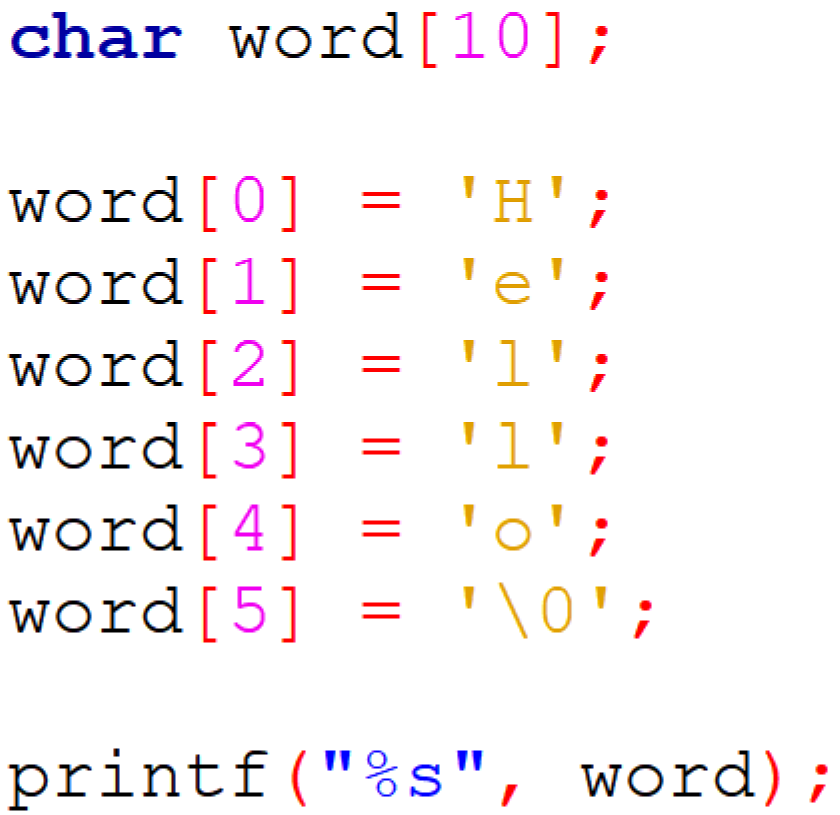
```
char word[10];

printf("%s", word);
```

Code Tracing /5



```
char word[10];

word[0] = 'H';
word[1] = 'e';
word[2] = 'l';
word[3] = 'l';
word[4] = 'o';

printf("%s", word);
```

Code Tracing /6

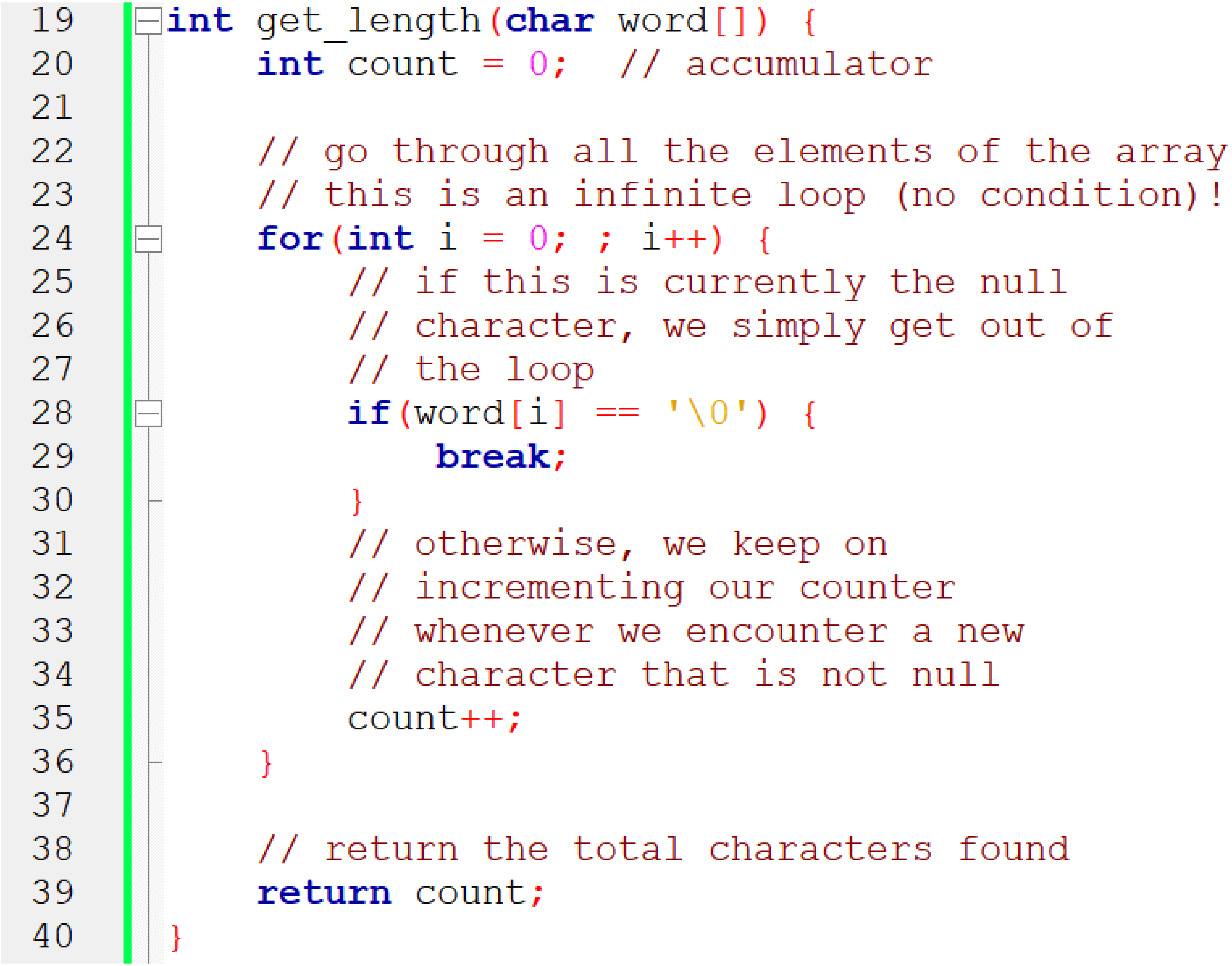


```
char word[10];

word[0] = 'H';
word[1] = 'e';
word[2] = 'l';
word[3] = 'l';
word[4] = 'o';
word[5] = '\0';

printf("%s", word);
```

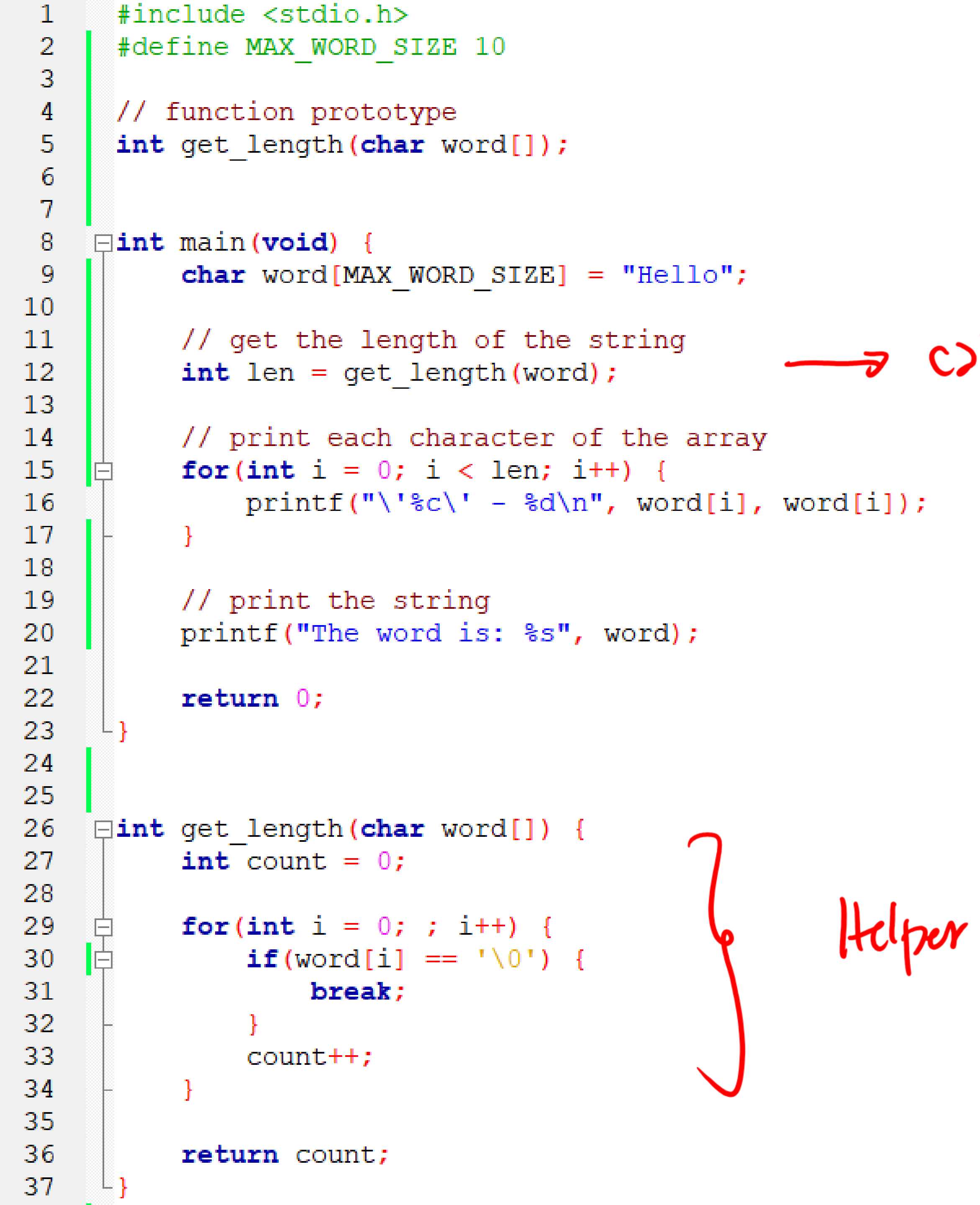
Practice: Function Completion

Write a function named **`get_length()`** that takes a string as its only parameter and returns the number of characters before the first `'\0'`. Assume the string is null-terminated.

```
int get_length(char *word) {





}
```



```
int get_length(char word[]) {
    int count = 0;  // accumulator

    // go through all the elements of the array
    // this is an infinite loop (no condition)!
    for(int i = 0; ; i++) {
        // if this is currently the null
        // character, we simply get out of
        // the loop
        if(word[i] == '\0') {
            break;
        }
        // otherwise, we keep on
        // incrementing our counter
        // whenever we encounter a new
        // character that is not null
        count++;
    }

    // return the total characters found
    return count;
}
```



```c
#include <stdio.h>
#define MAX_WORD_SIZE 10

// function prototype
int get_length(char word[]);


int main(void) {
    char word[MAX_WORD_SIZE] = "Hello";

    // get the length of the string
    int len = get_length(word);

    // print each character of the array
    for(int i = 0; i < len; i++) {
        printf("\'%c\' - %d\n", word[i], word[i]);
    }

    // print the string
    printf("The word is: %s", word);

    return 0;
}


int get_length(char word[]) {
    int count = 0;

    for(int i = 0; ; i++) {
        if(word[i] == '\0') {
            break;
        }
        count++;
    }

    return count;
}
```

→ call this function once only

} this will only print the characters H e l l o only .

} Helper function

Notes

- There are various strategies for solving the problem
- We don’t know the size of the array
- Because it is a string, we assume that it is null-terminated*
- The function fails if we don’t have a null-terminated string!

Reading Strings

- We can use **`scanf()`** and use the same format specifier

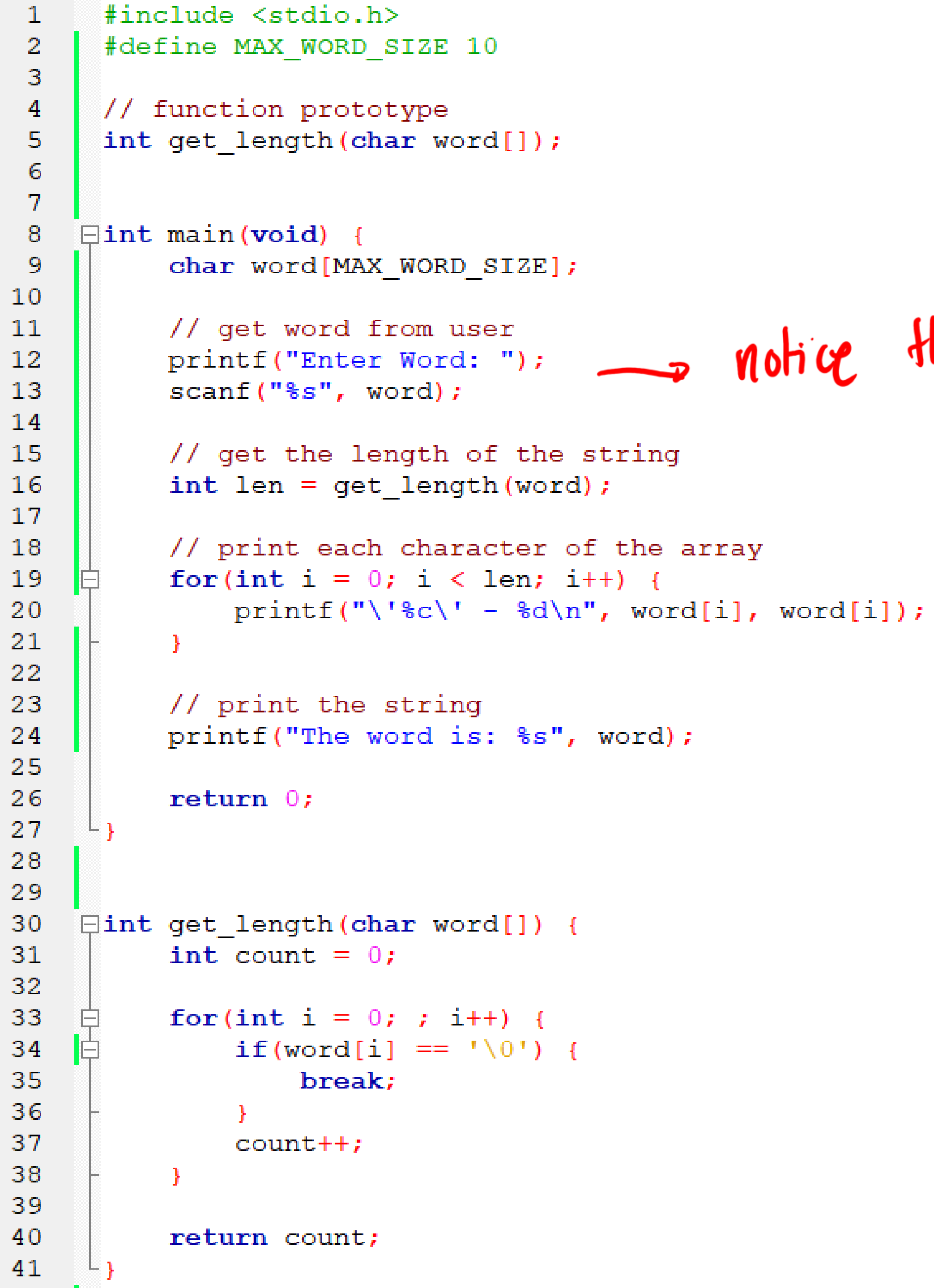
- Recall that when an array is passed to a function, the array's name decays into a pointer to its first element, there is no need to use the **`&`** (address of operator)

```
char word[101];
scanf("%s", word);
```

Practice

- Ask the user to enter a word then display it afterward

- Try typing a word that has a space in it



```c
#include <stdio.h>
#define MAX_WORD_SIZE 10

// function prototype
int get_length(char word[]);


int main(void) {
    char word[MAX_WORD_SIZE];

    // get word from user
    printf("Enter Word: ");
    scanf("%s", word);

    // get the length of the string
    int len = get_length(word);

    // print each character of the array
    for(int i = 0; i < len; i++) {
        printf("\'%c\' - %d\n", word[i], word[i]);
    }

    // print the string
    printf("The word is: %s", word);

    return 0;
}


int get_length(char word[]) {
    int count = 0;

    for(int i = 0; ; i++) {
        if(word[i] == '\0') {
            break;
        }
        count++;
    }

    return count;
}
```

→ notice that we don't use the & address operator here! It is because the array name decays to an address to its first element

Notes

- Some of the strings we entered were cut-off!*

- This is even if the length of our string was less than **`size-1`**

- For now, we are going to assume that strings entered by the user will not have any whitespaces

- More on this later!

Practice

Write a program that prompts the user to enter a single word. Then, display the number of vowels it contains. You may assume the word has no whitespace and is at most 100 characters long.

Sample Run

```
Enter Word: Hello
There are 2 Vowels in Hello
```

int count_vowels (char *word)

int is_vowel((char alpha)

for loop is needed to traverse each letter

Notes

- We used a built-in function from **`ctype.h`** to deal with characters

- We used the **`get_length()`** function that we implemented

- However, since it is a common operation, C has a built-in function

The `string.h` Library

- Contains built-in functions for string manipulation

- Ensure to include this in your C code whenever you use a function

Length of a String

A function that returns the number of characters before the `'\0'`

Note: it does not include the null character!

```
size_t strlen(const char *str);
```

Notes

- Its implementation is the same as how we did it earlier; However, its behavior is dependent on how the string was initialized*

- In an advanced course, you will realize that where you put this function call affects the overall performance of your program

- Specific for C, it is often advisable not to call **`strlen()`** as part of the condition in a loop (unless you have to)

Practice

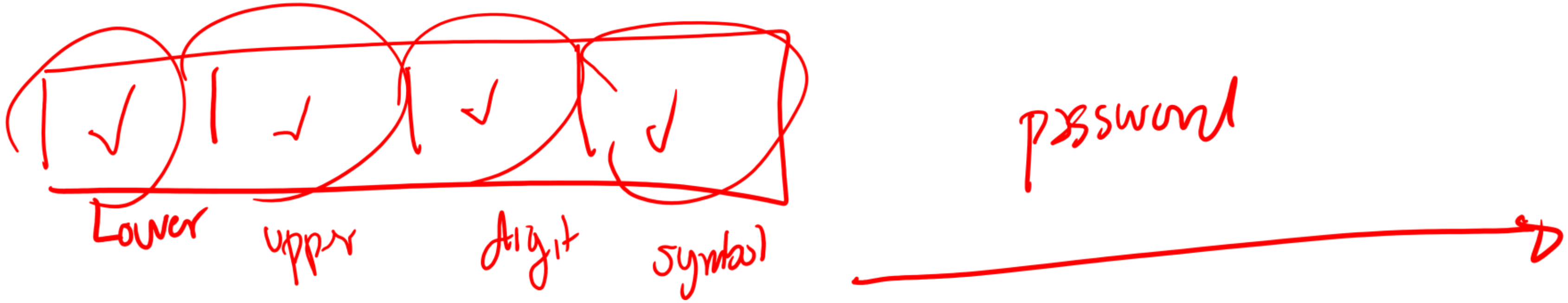
Write a function named **`contains_char()`** that takes two parameters: a string and a character. The function should return **`1`** if the character appears anywhere in the string, and **`0`** otherwise.

"apple" 'p' 'q'

```
int contains_char(char *word, char alpha) {



}
```

Your Turn!

Write a program that prompts the user to enter a password. Implement a function **`is_strong(char *password)`** that returns **1** if the string password contains at least one lowercase letter, one uppercase letter, one digit, and one of the following symbols (`@ ! # $`); otherwise, it returns **`0`**. Assume the password has no whitespace and are at most 100 characters long.



Sample Runs

Run 1

```
Enter Password: Hello!25
Password is Strong
```

Run 2

```
Enter Password: hello
Password is Weak
```

Challenge /1

Write a program that reads passwords from **`passwords.txt`**, with one password per line. Display the count of strong and weak passwords, assuming no whitespace in the passwords. Further, assume that passwords are at most 100 characters long. Refer to the earlier slide for the requirements of a strong password.

Challenge /2

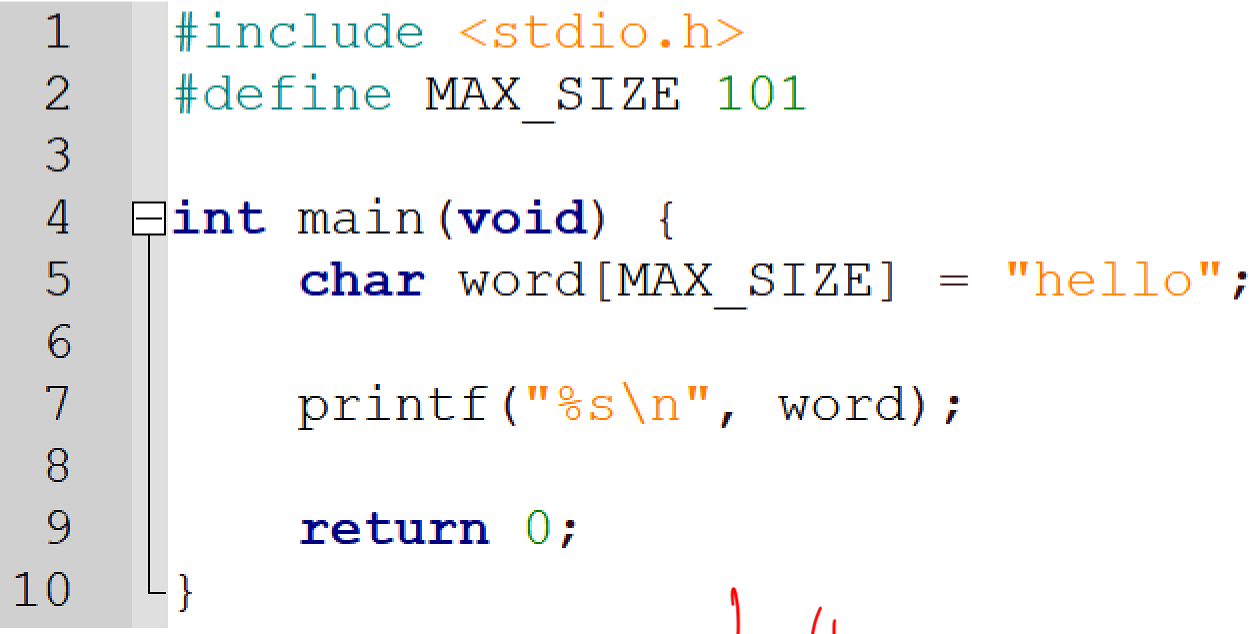
passwords.txt

```
Hello!25
hello
@loha
```

Expected Output

```
Passwords Found: 3
Strong: 1
Weak: 2
```

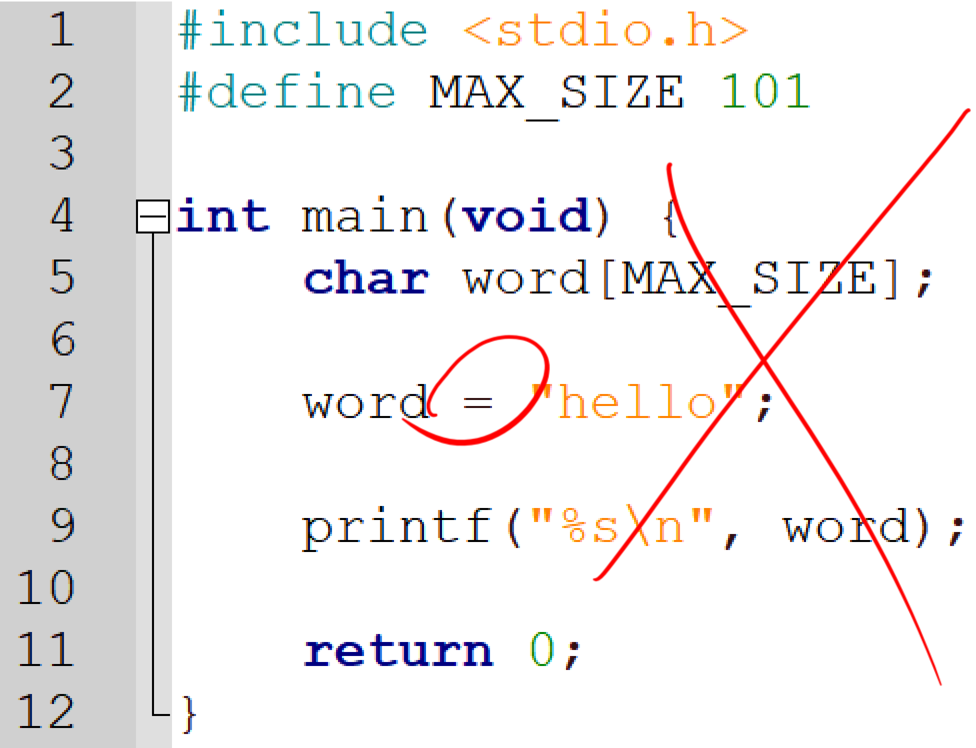
Code Tracing /1



```c
#include <stdio.h>
#define MAX_SIZE 101

int main(void) {
    char word[MAX_SIZE] = "hello";

    printf("%s\n", word);

    return 0;
}
```

hello

Code Tracing /2



```c
#include <stdio.h>
#define MAX_SIZE 101

int main(void) {
    char word[MAX_SIZE];

    word = "hello";

    printf("%s\n", word);

    return 0;
}
```

String Assignments /1

~~char word[10];~~
~~word = "Hello";~~

Unfortunately, you cannot use the **=** (assignment operator) to assign strings (as you may in other programming languages)

```
char *strcpy(char *destination, const char *source);
```

dst = src;
X

The two parameters are **pointers** to character arrays

String Assignments /2

Ensure that the destination has **enough space** (including the `'\0’`)

Otherwise, it causes an undefined behavior

Does not check for number of characters to copy

String Assignments /3

An alternative is to specify the count:

```
char *strncpy(char *dest, const char *src, size_t n);
```

However, it may not include the null character as part of the count. For example, source is “hello” and count is 3.

Notes

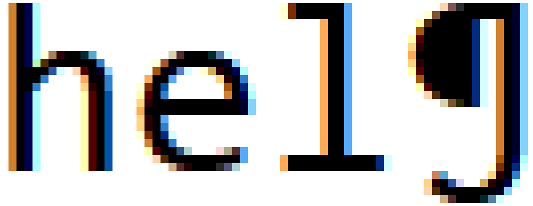
If source is shorter than the specified limit **`n`**, it copies the **`'\0'`**

If source is equal or longer than **`n`**, then **`'\0'`** is not appended

Example (Caution)

```
char str1[10] = "hello";
char str2[10];

strncpy(str2, str1, 3);
printf("%s", str2);
```

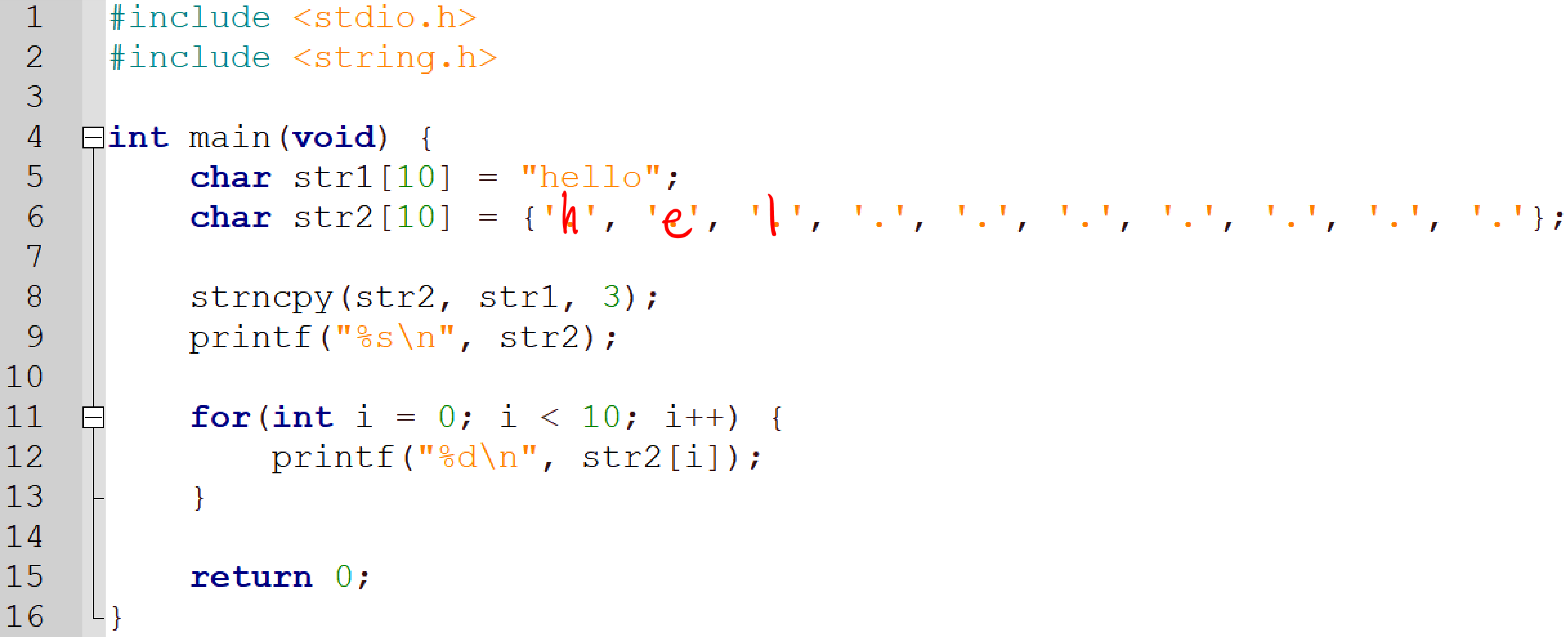
garbage value



```
helg
```

There is no guarantee that strncpy will include the \0 from the source, so it is possible to see garbage values

Discussion



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

int main(void) {
    char str1[10] = "hello";
    char str2[10] = {'h', 'e', 'l', '.', '.', '.', '.', '.', '.', '.'};

    strncpy(str2, str1, 3);
    printf("%s\n", str2);

    for(int i = 0; i < 10; i++) {
        printf("%d\n", str2[i]);
    }

    return 0;
}
```

String Concatenation

+

If you want to combine two strings together

dst = dst + src

Again, you cannot use the assignment and the **+** operators

```
    char *strcat(char *dest, const char *src);


    char *strncat(char *dest, const char *src, size_t n);
```

Notes

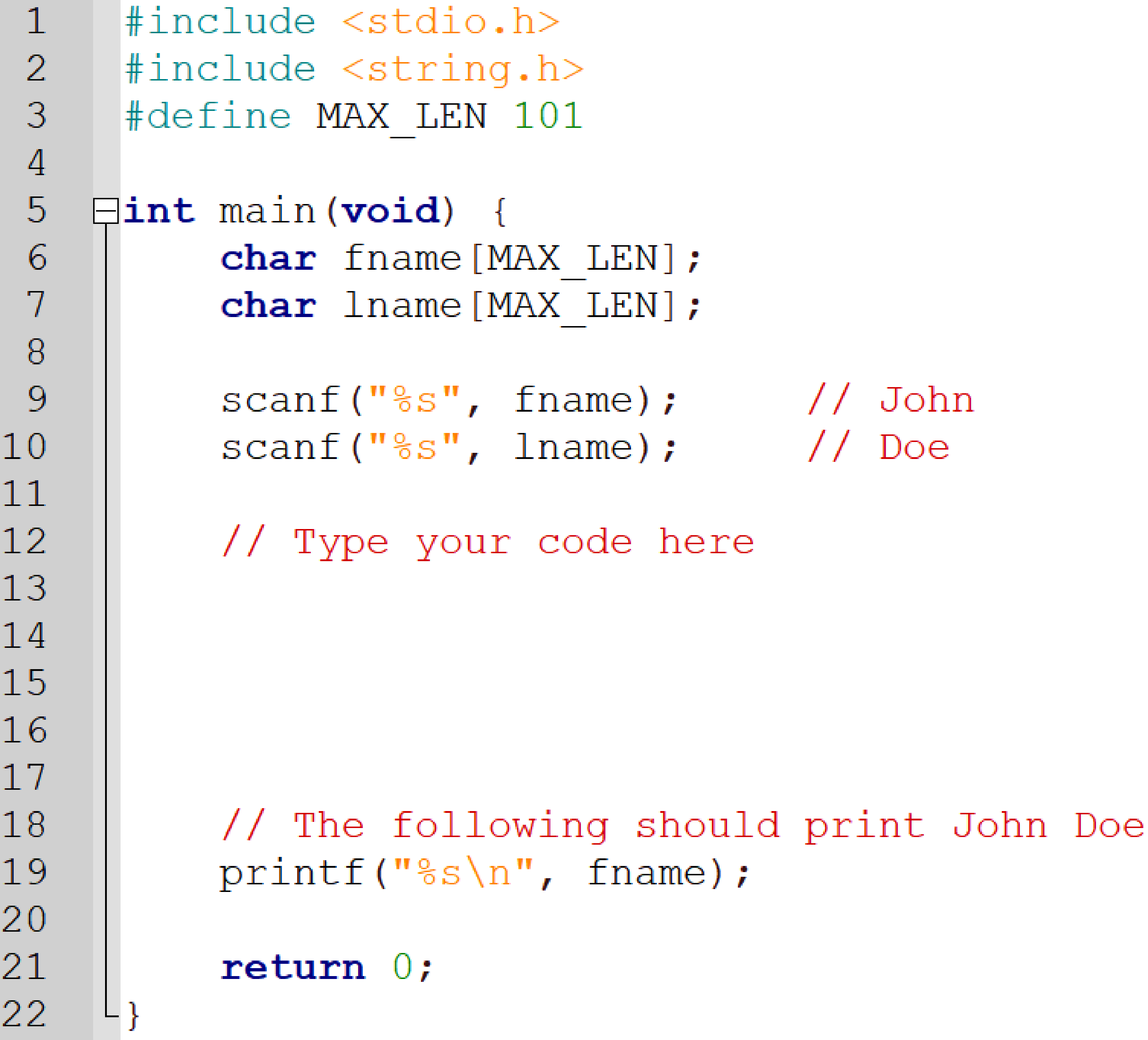
- When the combined size of destination and source is beyond the size of destination, the behavior is undefined (i.e., buffer overflow)

- It does not create a new string, rather it modifies the destination

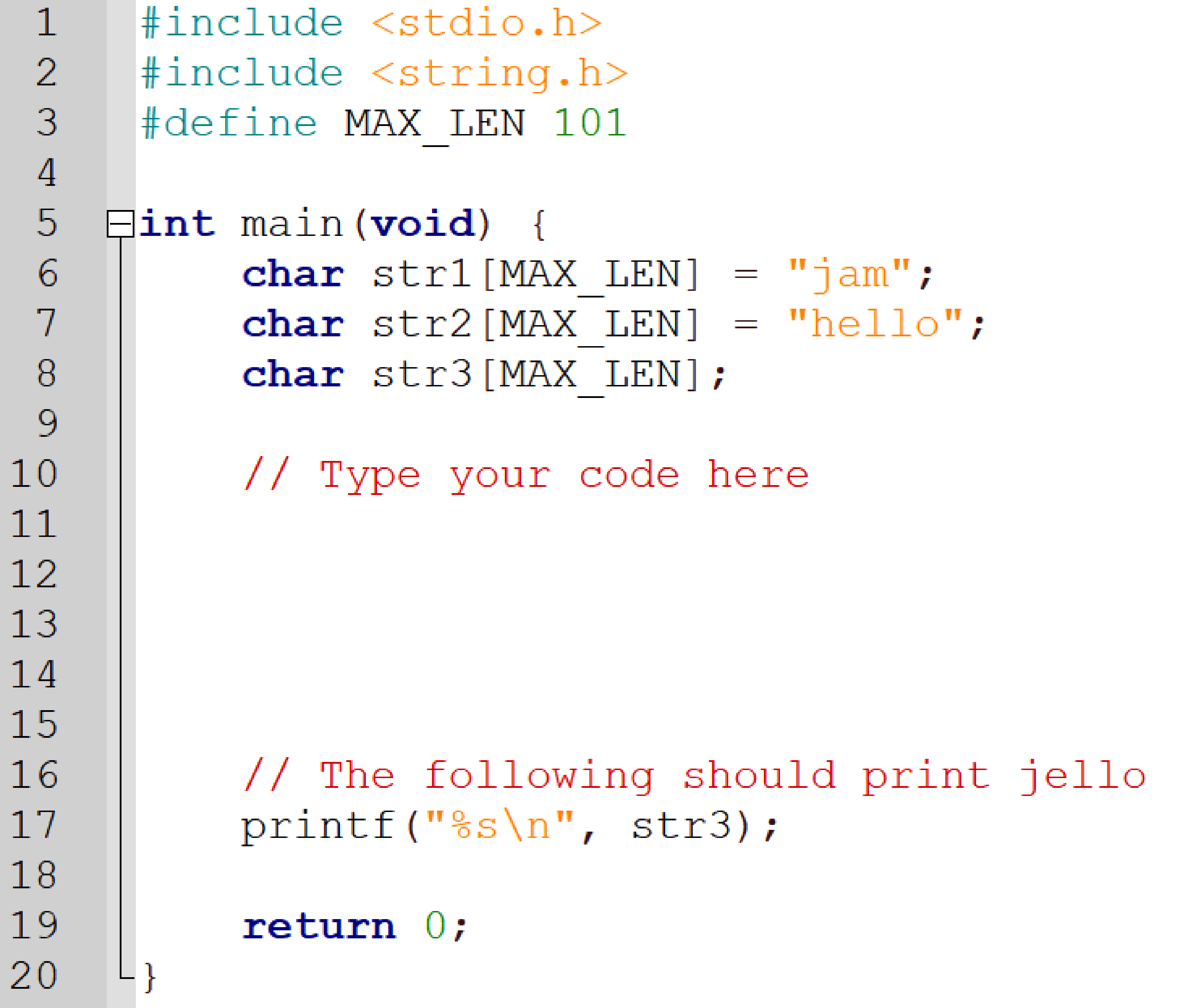
- Reference: https://man7.org/linux/man-pages/man3/strcat.3.html

Practice

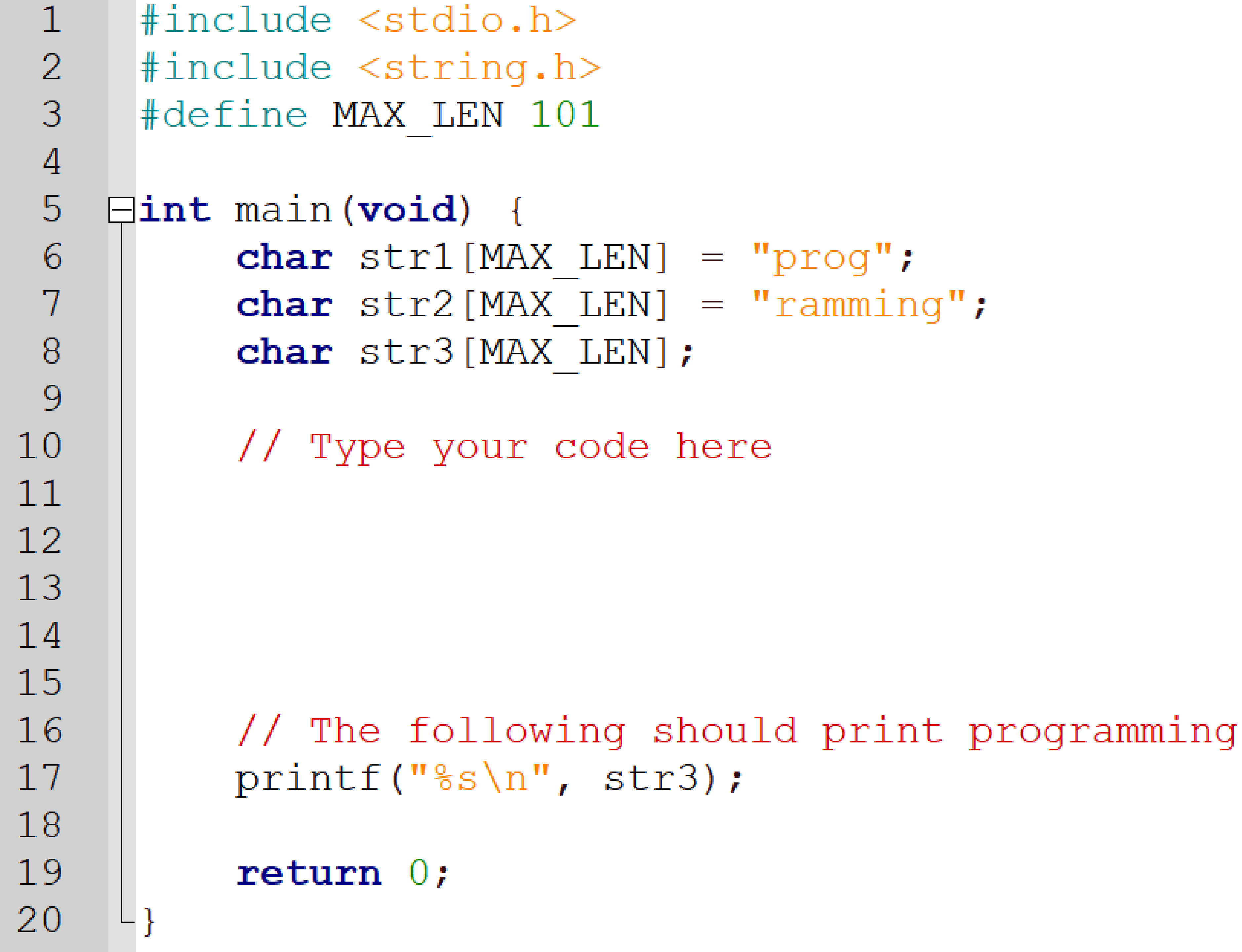
Write a program that asks the user to enter their first and last name. Then, combine the two to form their full name. For example, if the inputs are **`"John"`** and **`"Doe"`**, the result should be **`"John Doe"`**. Print this afterward. Assume both first and last names have no whitespaces and are at most 100 characters long.



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101

int main(void) {
    char fname[MAX_LEN];
    char lname[MAX_LEN];

    scanf("%s", fname);     // John
    scanf("%s", lname);     // Doe

    // Type your code here





    // The following should print John Doe
    printf("%s\n", fname);

    return 0;
}
```



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101

int main(void) {
    char str1[MAX_LEN] = "jam";
    char str2[MAX_LEN] = "hello";
    char str3[MAX_LEN];

    // Type your code here





    // The following should print jello
    printf("%s\n", str3);

    return 0;
}
```



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 101

int main(void) {
    char str1[MAX_LEN] = "prog";
    char str2[MAX_LEN] = "ramming";
    char str3[MAX_LEN];

    // Type your code here





    // The following should print programming
    printf("%s\n", str3);

    return 0;
}
```

Practice /1

Pig Latin is a playful form of language often used as a code or secret language. It involves rearranging the letters in a word based on simple rules. The following is a simplified version.

Practice /2

If the word starts with a **vowel**, simply add **“way”** to the end

```
    apple => appleway
```

If the word starts with a **consonant**, move the consonant to the end and add **“ay”**

```
    pig => igpay
```

Practice /3

Write a program that asks the user to enter a single word containing only English letters (1–100 characters). Afterward, it prints the Pig Latin version of that word.

Sample Runs

Run 1

```
Enter Word: apple
Pig Latin: appleway
```

Run 2

```
Enter Word: pig
Pig Latin: igpay
```

Refer to Webcourses for the C Code: Pig Latin

Discussion

Which one comes first in **lexicographic** order?

"**dog**" or "**cat**"

"**cat**" or "**car**"

"**car**" or "**cartoon**"

String Comparisons /1

- Given two strings **`str1`** and **`str2`**, which comes *first*?

- The idea is that it does this character by character of both strings

- Compares their ASCII values, if there is a tie, go to the next

- The one with the lower ASCII value is *lesser than* the other

String Comparisons /2

str1	str2	
apple	ball	(-)
ball	apple	(+)
apple	apple	0

Take note that this is **case-sensitive**

-1 +1

```
int strcmp(const char *s1, const char *s2);
```

It returns 3 possible values:

Negative if **`str1`** comes *before* **`str2`** (i.e., **`str1`** < **`str2`**)

Positive if **`str2`** comes *before* **`str1`** (i.e., **`str1`** > **`str2`**)

Zero if both are *the same* strings (i.e., **`str1`** == **`str2`**)

Discussion

D

200	111	103
'd'	'o'	'g'
99	97	116
'c'	'a'	't'

+1

A

65 – 65535

97

Practice

Will `strcmp()` return a negative, positive, or zero?

`"hello"` and `"world"`

→ `"hello"` and `"hi"`

→ `"hello"` and `"Hello"` +

→ `"hi"` and `"hibernate"` −

Discussion

How do you perform a case-insensitive string comparison?

cat CAT

stricmp()

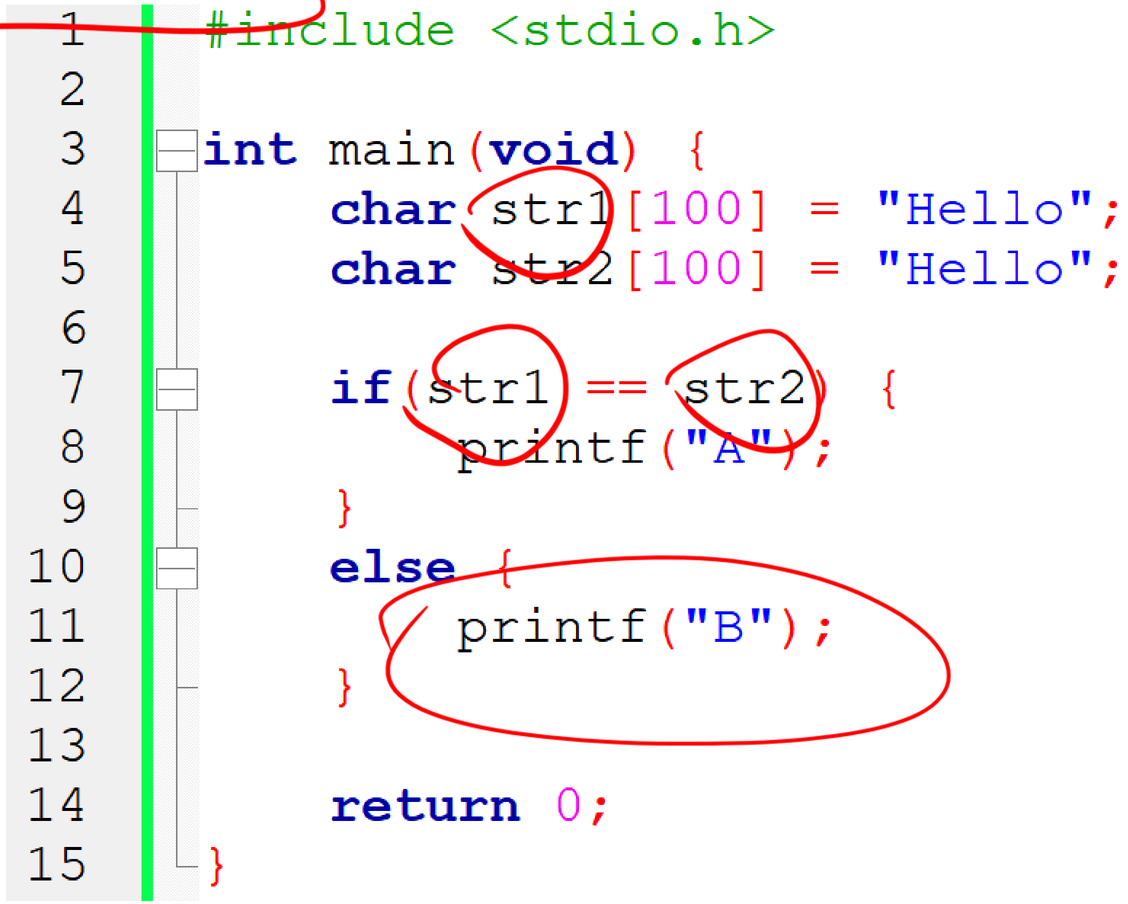
Code Tracing

strcmp(str1, str2)==0

What is the output?

Str1 == str2

address address



```
#include <stdio.h>

int main(void)  {
    char str1[100] = "Hello";
    char str2[100] = "Hello";

    if(str1 == str2)  {
        printf("A");
    }
    else {
        printf("B");
    }

    return 0;
}
```

Practice

Write a program that prompts the user to enter a password and checks if it matches **`OpenSesame`**. Display **`Granted`** if correct, otherwise **`Denied`**. Assume the password contains no whitespace and is at most 100 characters long.

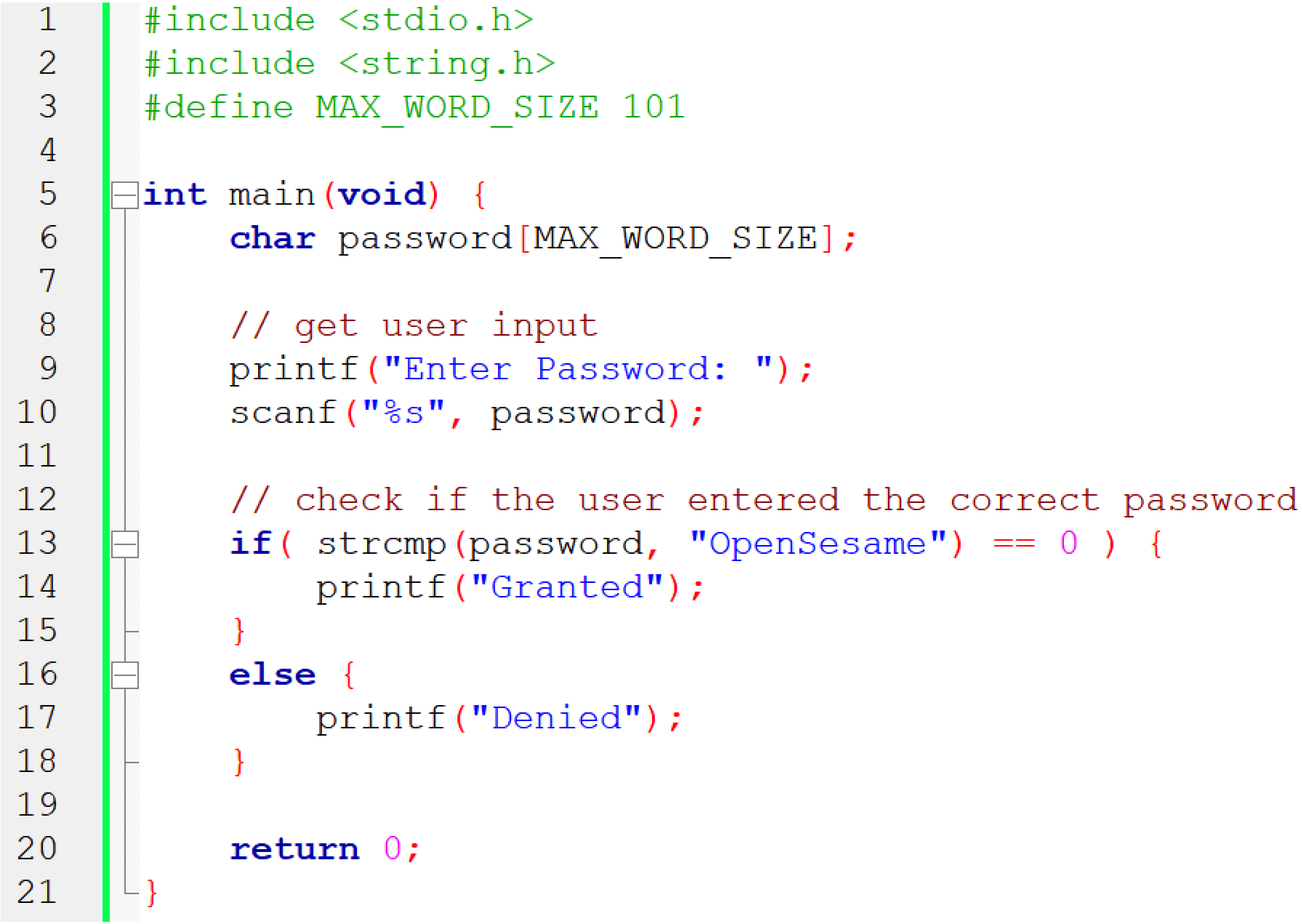
Sample Runs

Run 1

```
Enter Password: Hello
Denied
```

Run 2

```
Enter Password: OpenSesame
Granted
```



```c
#include <stdio.h>
#include <string.h>
#define MAX_WORD_SIZE 101

int main(void) {
    char password[MAX_WORD_SIZE];

    // get user input
    printf("Enter Password: ");
    scanf("%s", password);

    // check if the user entered the correct password
    if( strcmp(password, "OpenSesame") == 0 ) {
        printf("Granted");
    }
    else {
        printf("Denied");
    }

    return 0;
}
```

Challenge

Write a program that reads a file named `words.txt`, which contains an arbitrary number of words (one per line). Each word has no whitespace and is at most 50 characters long. The program must create a new file named `words_info.txt`. For each word from the input file, write the word followed by its length, separated by a single space. Preserve the original order of the words.

Sample Files

words.txt

```
banana
apple
kiwi
grape
orange
```

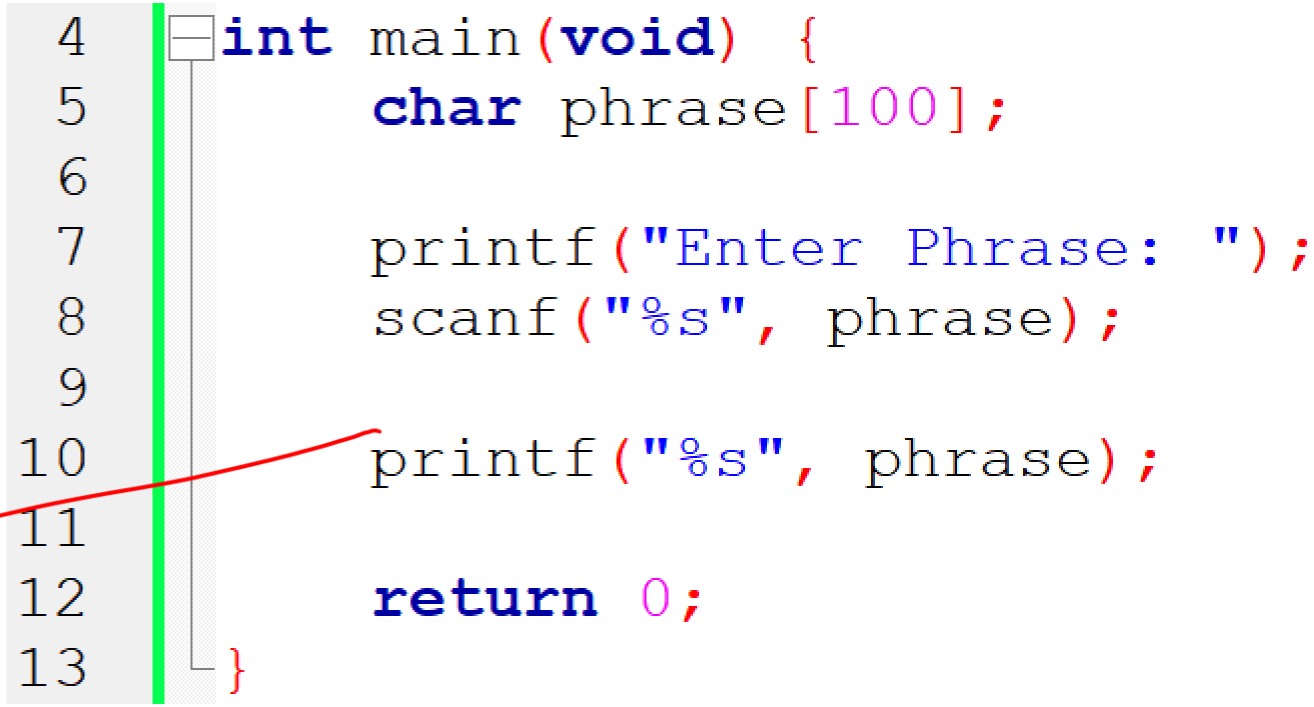
words_info.txt

```
banana 6
apple 5
kiwi 4
grape 5
orange 6
```

Code Tracing /1

What is the output?

Charge



```c
int main(void) {
      char phrase[100];

    printf("Enter Phrase: ");
    scanf("%s", phrase);

    printf("%s", phrase);

      return 0;
}
```

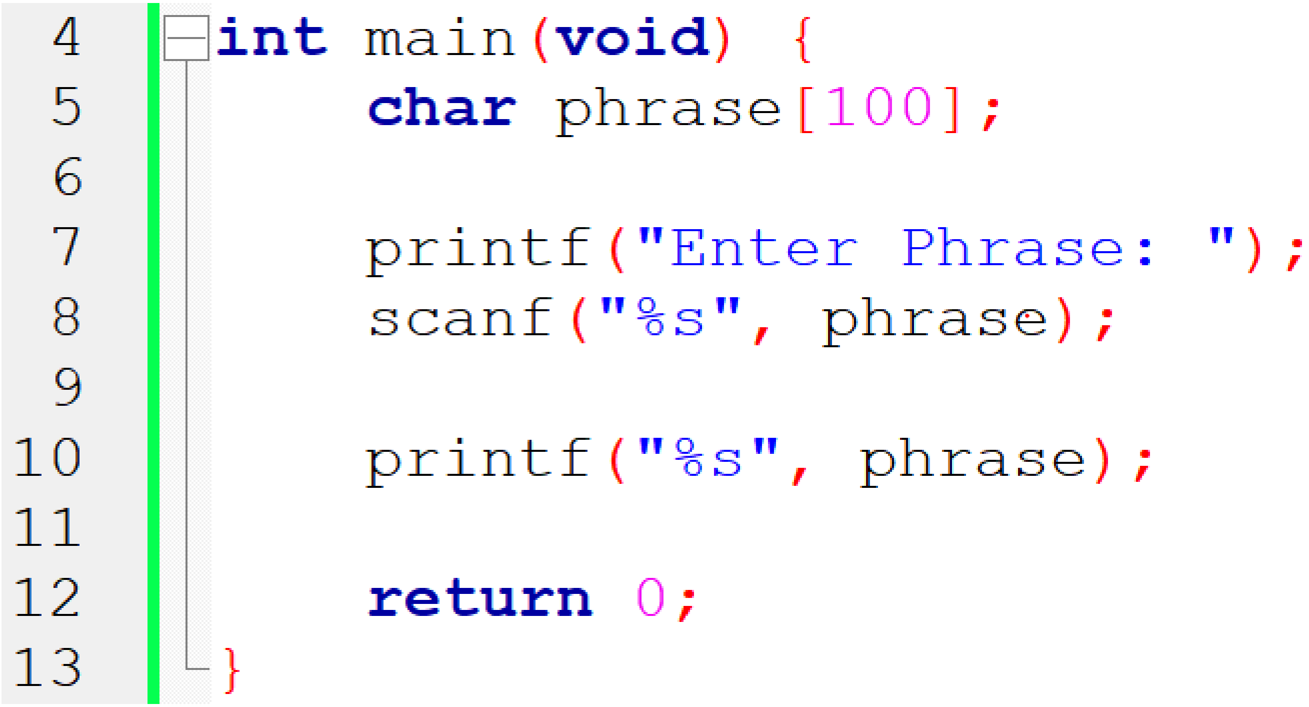
[illegible]

Charge

Code Tracing /2

What is the output?

Charge On



```c
int main(void) {
    char phrase[100];

    printf("Enter Phrase: ");
    scanf("%s", phrase);

    printf("%s", phrase);

    return 0;
}
```

Recall

- We had an issue reading **strings with whitespaces**
- This is due to the behavior of the format specifier **`%s`** of **`scanf()`**
- It reads until a whitespace is found (whitespace not included)
- It also ignores leading spaces

Behind the Scenes

- User types input using the **keyboard**
- Operating System stores characters in the **input buffer**
- Standard Input Stream (**`stdin`**) reads from the input buffer
- **`scanf()`** reads from **`stdin`**

Buffer

- **Temporary storage** that stores characters before it gets processed (read or written)

- To properly read input, the buffer *must always be empty*

- When **`scanf()`** is used to read a string, any other character including the whitespace remains in the buffer

Example /1

Assume that the user was asked to enter an input and typed the following using the keyboard:

```
Charge On
```

Example /2

Assume that the user was asked to enter an input and typed the following using the keyboard:

```
Charge On
```

```
     'C' 'h' 'a' 'r' 'g' 'e' ' ' 'O' 'n' '\n'
```

Example /3

Assume that the user was asked to enter an input and typed the following using the keyboard:

\n ' '

```
Charge On
```

read

not read

```
'C' 'h' 'a' 'r' 'g' 'e' ' ' 'O' 'n' '\n'
```

Notes

Let’s investigate further and determine the contents of the buffer

The `getchar()` Function

- Returns one character from the standard input stream (**`stdin`**)

- If the input buffer is empty, it waits for user input

- It does not skip whitespaces

Practice

- Read a string with a white space

- Try to check the contents of the buffer

- Read a character when the buffer is already empty

Notes

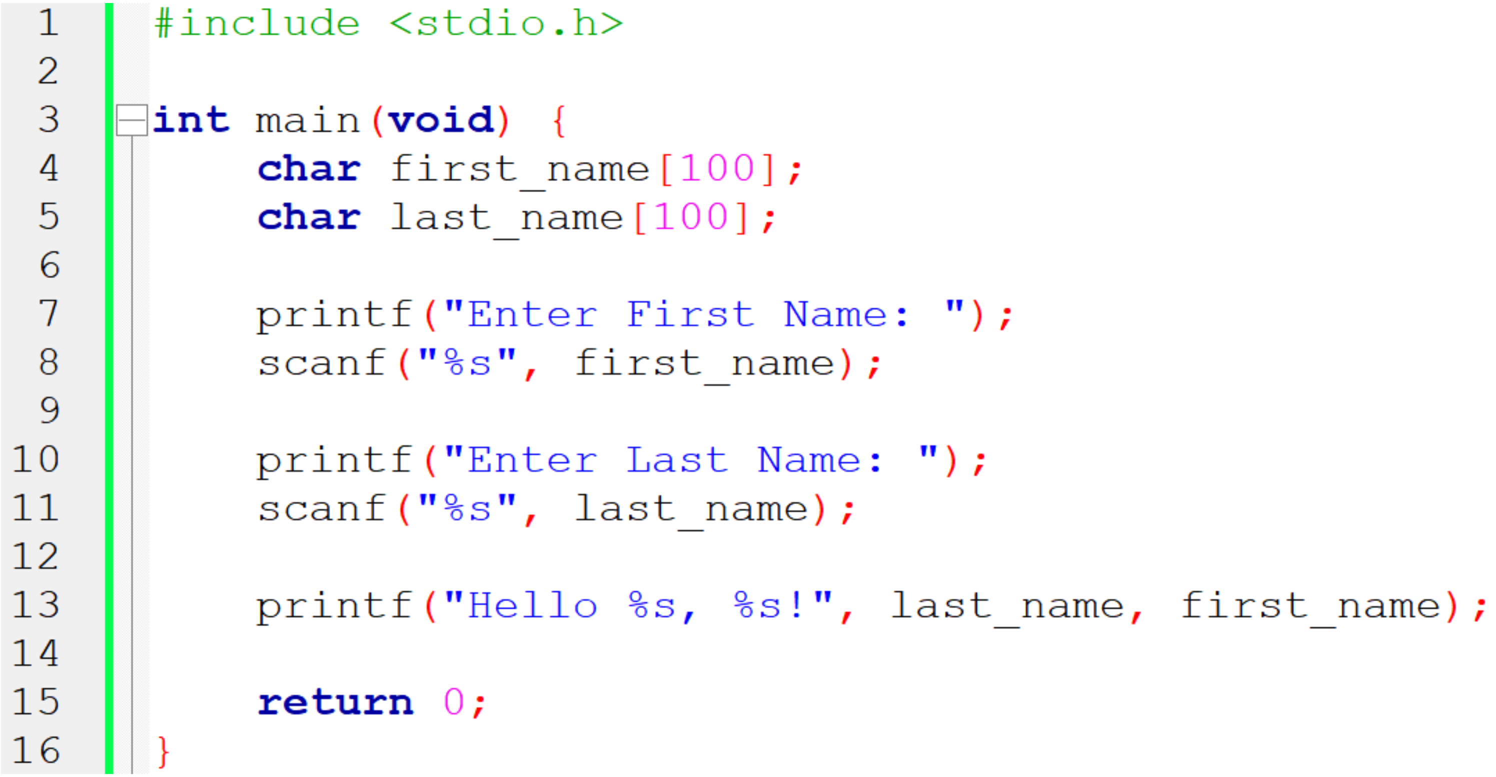
- It turns out that the buffer was not empty
- This was due to **`scanf()`** not reading everything the user typed
- When the buffer is empty and we called **`getchar()`**, it is now waiting for user input (e.g., from the keyboard)
- More on this later!

Code Tracing /1

What is the output?

John

Doe



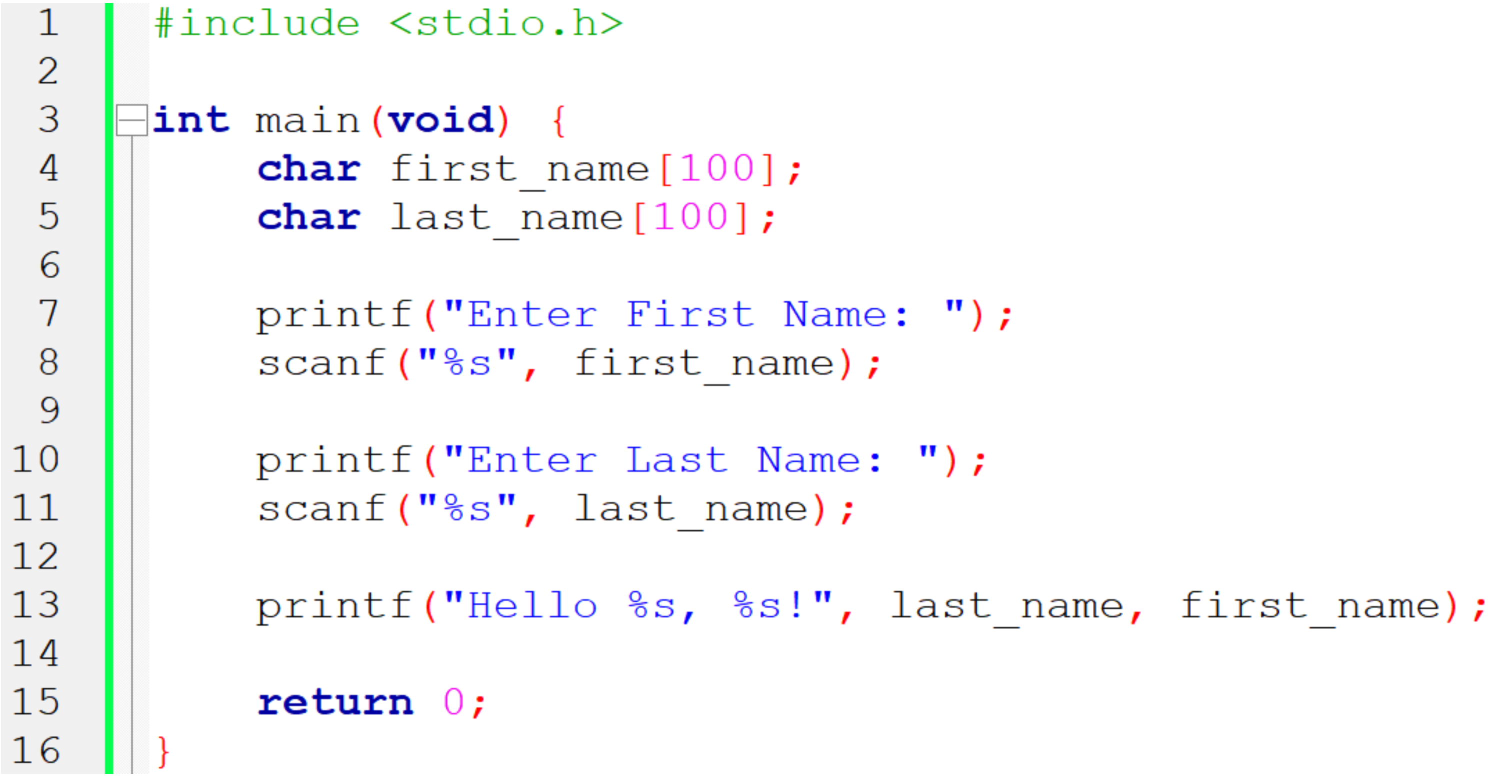
```c
#include <stdio.h>

int main(void)  {
        char first_name[100];
        char last_name[100];

        printf("Enter First Name: ");
        scanf("%s", first_name);

        printf("Enter Last Name: ");
        scanf("%s", last_name);

        printf("Hello %s, %s!", last_name, first_name);

        return 0;
}
```

Code Tracing /2

What is the output?

John Smith

Doe



```c
#include <stdio.h>

int main(void) {
    char first_name[100];
    char last_name[100];

    printf("Enter First Name: ");
    scanf("%s", first_name);

    printf("Enter Last Name: ");
    scanf("%s", last_name);

    printf("Hello %s, %s!", last_name, first_name);

    return 0;
}
```

Notes

- We noticed that if there was no whitespace for both the first name and last name, the program behaved correctly

- However, when the user enters a first name that has a whitespace, it spills over the last name

- Why?

Discussion

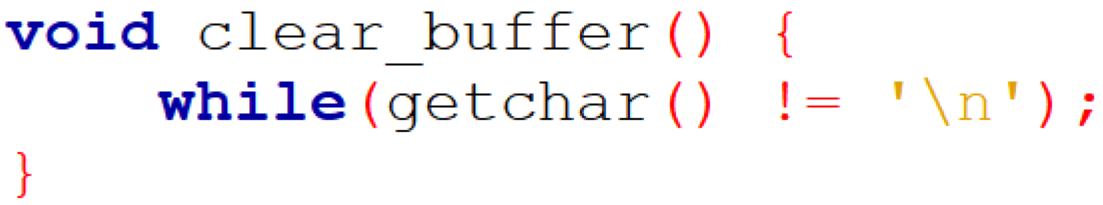
- The previous issue, again, is due to the buffer not being empty

- Therefore, **`scanf()`** did not let the user type in an input

Helper Function /1

If you encounter this, it means that your buffer is not empty!

You can create and use a helper function **`clear_buffer()`**

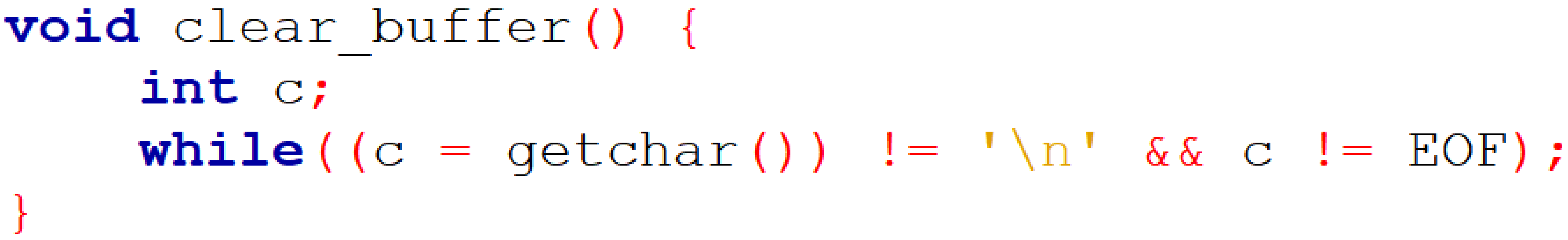


```
void clear_buffer() {
    while(getchar() != '\n');
}
```

Helper Function /2

We can also have an expanded version of this helper function

Notice the condition of the loop



```
void clear_buffer() {
    int c;
    while((c = getchar()) != '\n' && c != EOF);
}
```

Notes

- The behavior of the program changed, but it is still *wrong*
- We can get user input for the second time
- However, for the first input, it ignored those after the whitespace
- How can we fix this?

The `fgets()` Function

If you want to read a string that has a whitespace

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

It reads a sequence of characters until a `'\n'` (new line character) is encountered or the specified buffer length is exceeded

Syntax:

```
fgets(str, MAX_WORD_SIZE, stdin);
```

the name of the array of char (string)

size limit to read, usually the same as size of array

standard input stream

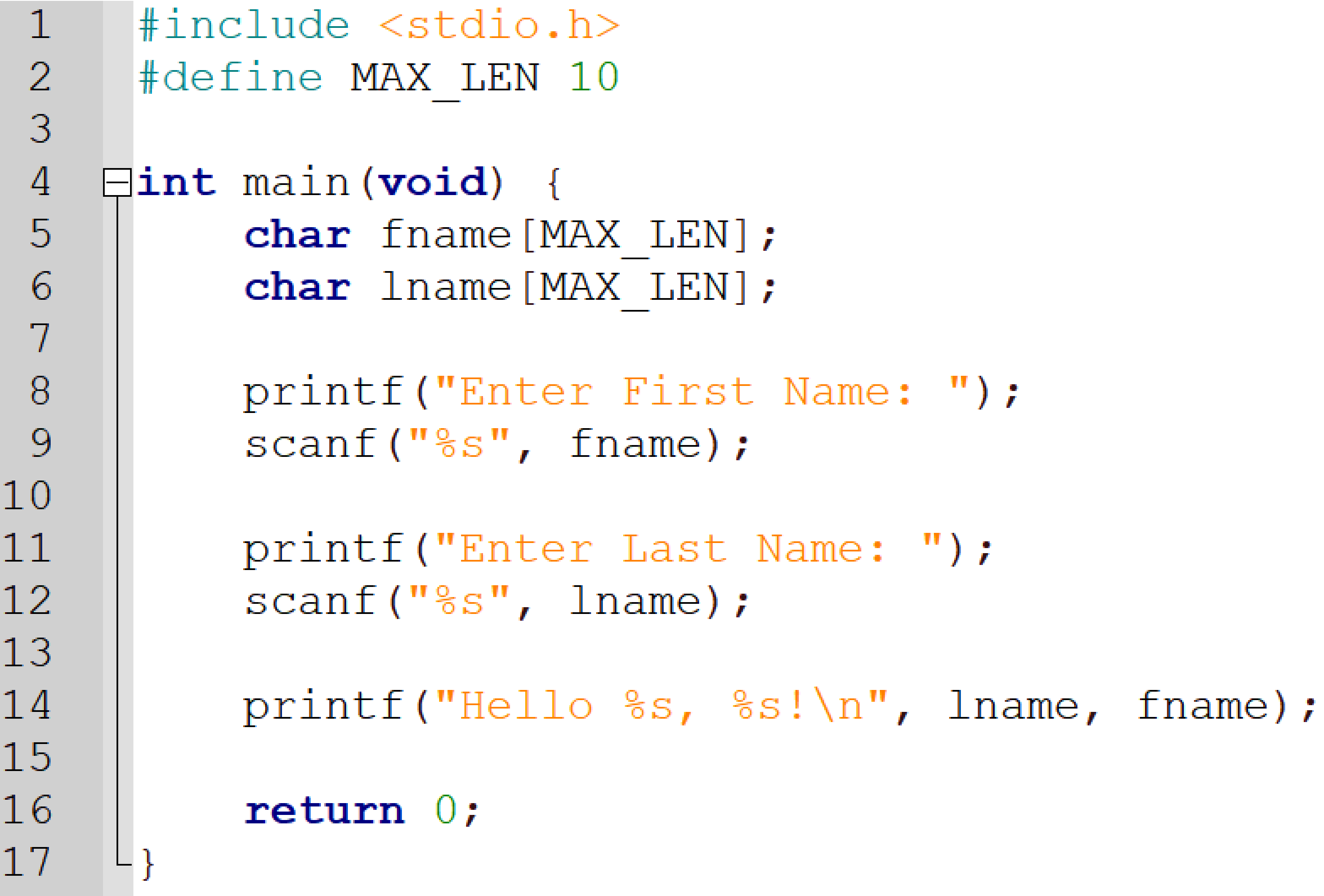
Practice

- Set the size of the string to be 10

- Enter the name **`John`** then **`D`**

- Enter the name **`John`** then **`Doe`**

- Enter the name **`John Smith`** then **`Doe`**



```c
#include <stdio.h>
#define MAX_LEN 10

int main(void) {
    char fname[MAX_LEN];
    char lname[MAX_LEN];

    printf("Enter First Name: ");
    scanf("%s", fname);

    printf("Enter Last Name: ");
    scanf("%s", lname);

    printf("Hello %s, %s!\n", lname, fname);

    return 0;
}
```

Visualizing the Array /1

John

0	1	2	3	4	5	6	7	8	9
J	o	h	n	\n	\0				

Visualizing the Array /2

`John Smith`

0	1	2	3	4	5	6	7	8	9
J	o	h	n	_	S	m	i	t	\0

Notes /1

- It seems that we have been seeing an extra newline
- However, there are times in which we don’t see it
- Take note of the limit of the length of the string: **`size-1`**
- One less because of the null character

Notes /2

- When we use **`fgets()`** and we provide a string that has a length less than the limit, it will include the **`'\n'`**

- Where did it come from?

Notes /3

trailing space

- If we provide a string that has a length that exceeds the limit, it will ignore the remaining and will not add the `'\n'`

- The remaining characters are left in the buffer

Notes /4

- Our **`scanf()`** also was skipped due to the buffer not being empty
- We already know how to solve the issue with the buffer

Discussion

- When calling **`fgets()`**, if the limit is exceeded, the remaining characters remain in the buffer

- Now, how do we get rid of the extra **`'\n'`**

"\n"

Helper Function /1

"abc"

Use the **strcspn()** function from the **string.h** library

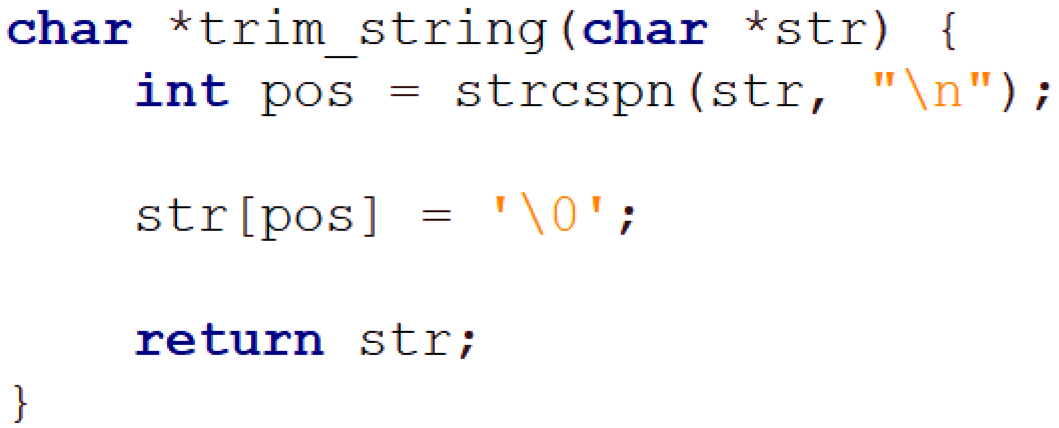
```
size_t strcspn(const char *str, const char *reject);
```

Return the index of the first character in **str** that is in **reject**

Otherwise, it returns the length of **str**

Helper Function /2

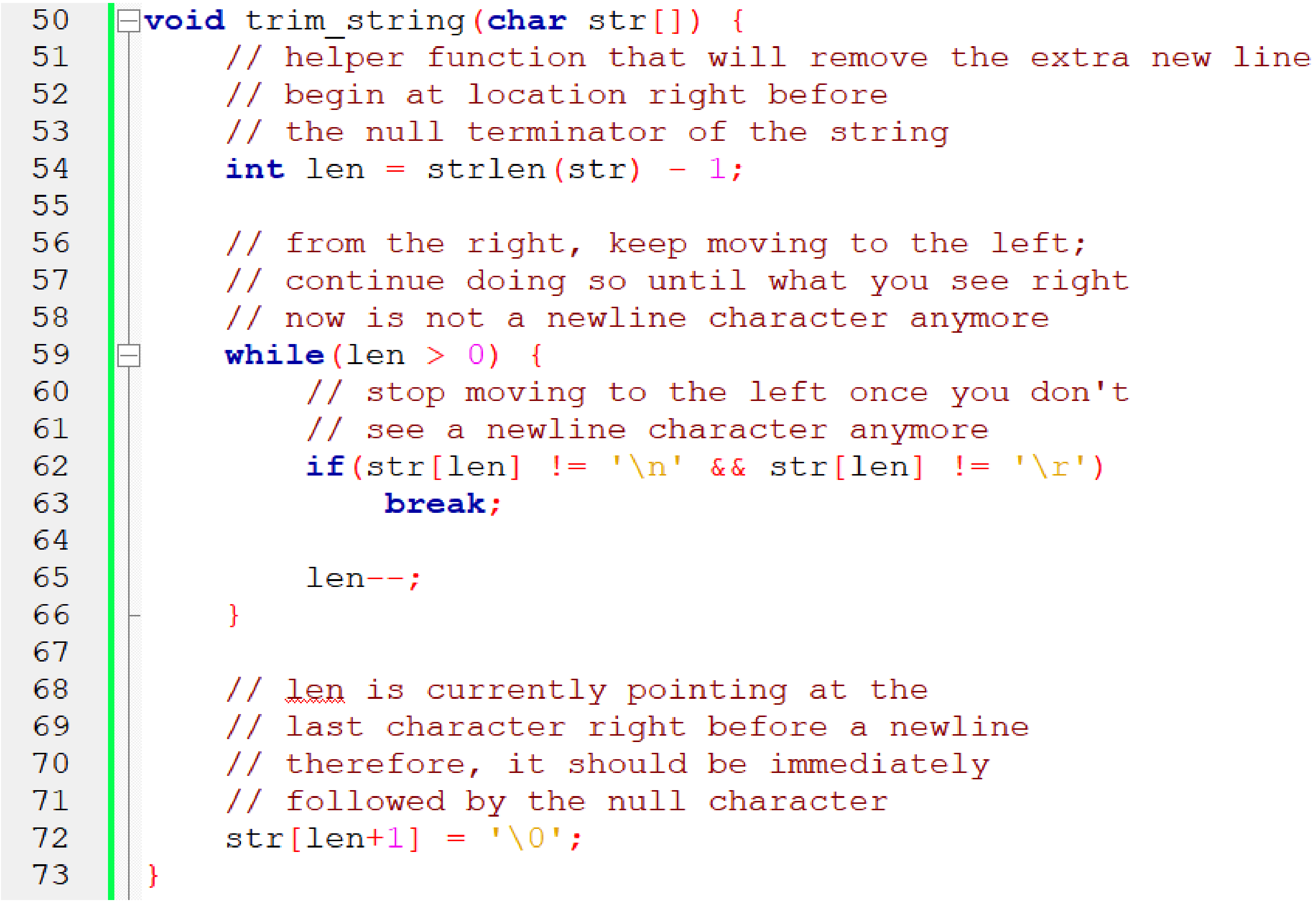
Strategy is to find the location or index of the first instance of `'\n'`



```c
char *trim_string(char *str) {
    int pos = strcspn(str, "\n");

    str[pos] = '\0';

    return str;
}
```

This is simpler as it relies on built-in function

Helper Function /3



```c
void trim_string(char str[]) {
    // helper function that will remove the extra new line
    // begin at location right before
    // the null terminator of the string
    int len = strlen(str) - 1;

    // from the right, keep moving to the left;
    // continue doing so until what you see right
    // now is not a newline character anymore
    while(len > 0) {
        // stop moving to the left once you don't
        // see a newline character anymore
        if(str[len] != '\n' && str[len] != '\r')
            break;

        len--;
    }

    // len is currently pointing at the
    // last character right before a newline
    // therefore, it should be immediately
    // followed by the null character
    str[len+1] = '\0';
}
```

Discussion /1

- Our helper function **`trim_string()`** takes a **`char *`** as input and also returns a **`char *`**

- It may appear to return a new string, but it simply returns the same pointer to the original string, which has been modified **in place** (*unless a new string is explicitly allocated; more in the future*).

- Why use this design?

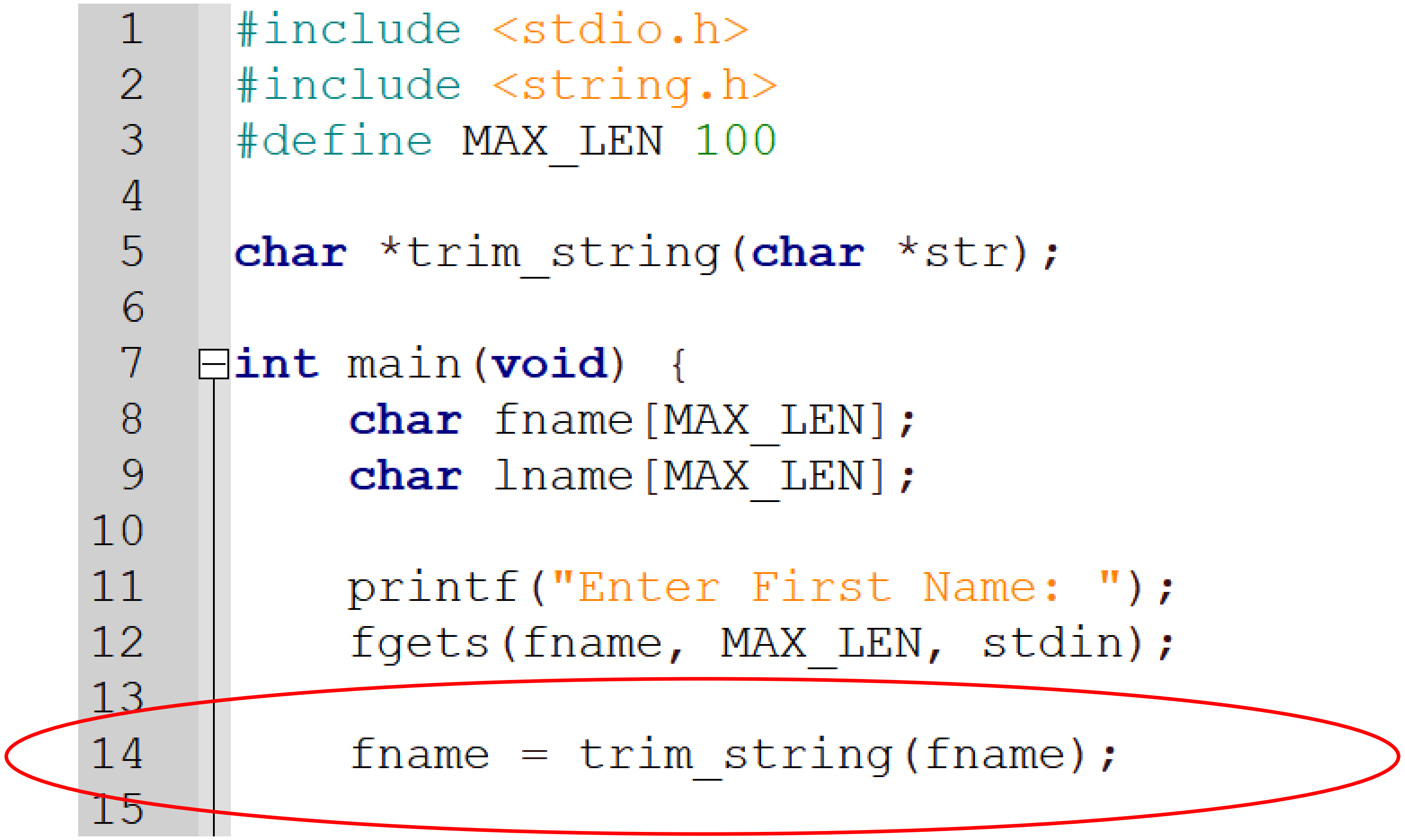
Discussion /2

- This design prepares us for scenarios where we may later allocate and return new dynamically created strings.

- While an array name decays into a pointer when passed to a function, it still cannot be reassigned to point somewhere else.

- We can still pass an array to our function, but we do not rely on array semantics; we treat the argument simply as a **`char *`**

Example of Compile Error



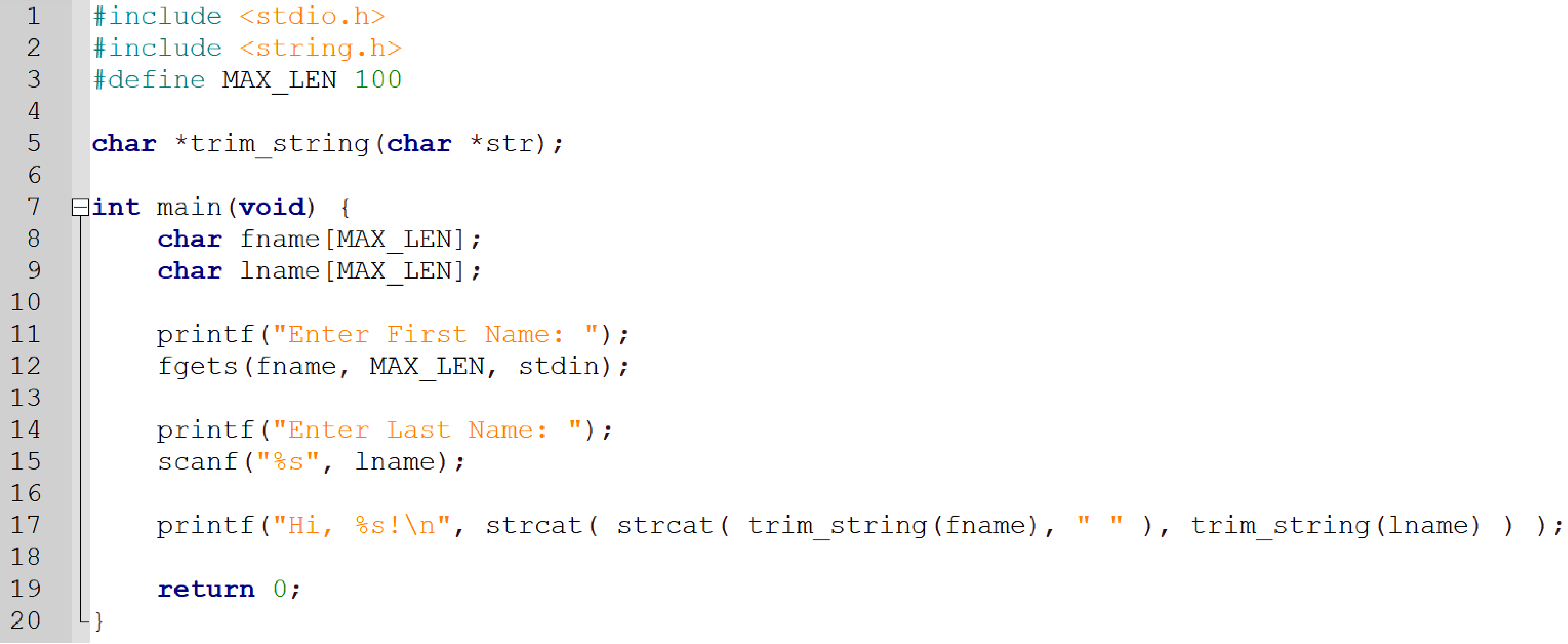
```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 100

char *trim_string(char *str);

int main(void) {
    char fname[MAX_LEN];
    char lname[MAX_LEN];

    printf("Enter First Name: ");
    fgets(fname, MAX_LEN, stdin);

    fname = trim_string(fname);

```

Discussion /3

- What’s the benefit? Returning the same pointer allows us to chain multiple string-processing calls, making the code more concise and expressive.

- This pattern is also used by several standard C library functions (e.g., `strcpy()` and `strcat()`), which return a pointer to the destination string to support chaining.

Example



```c
#include <stdio.h>
#include <string.h>
#define MAX_LEN 100

char *trim_string(char *str);

int main(void) {
    char fname[MAX_LEN];
    char lname[MAX_LEN];

    printf("Enter First Name: ");
    fgets(fname, MAX_LEN, stdin);

    printf("Enter Last Name: ");
    scanf("%s", lname);

    printf("Hi, %s!\n", strcat( strcat( trim_string(fname), " " ), trim_string(lname) ) );

    return 0;
}
```

The `fputs()` Function

Prints a string to an output stream (e.g., **`stdout`**)

```
int fputs(const char *str, FILE *stream);
```

Syntax:

fputs (str , stdout);

name of the array of char (string)

the output stream where to print (stdout is standard output)

Other Functions

- There are also other functions that you can use:

  `gets()` and **`puts()`**

- Difference is that they don’t account for the size of the array

- Not advisable to use them and could lead to **Segmentation Fault**

Strings and Files

fgets(str, MAX, ifile);

- Similar to how we did it with numbers
- We can use **`fscanf()`** and use the format specifier **`"%s"`**
- We can also use **`fgets()`**
- Because we are reading from a file instead of **`stdin`**, we simply change it to the **`FILE`** pointer (e.g., **`ifile`**)

Practice

- Read strings from a file **`"fruits.txt"`**

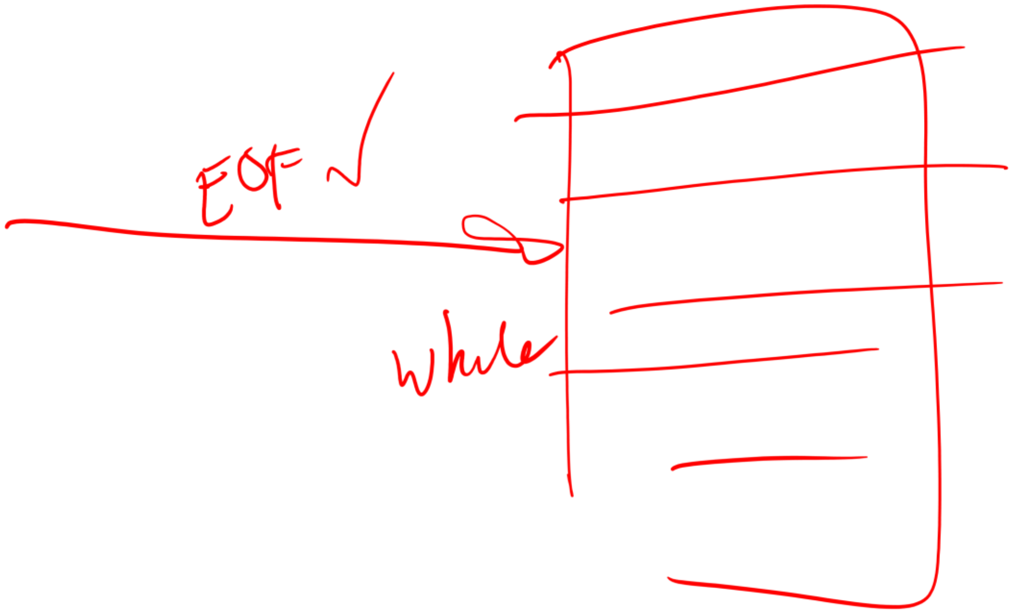
- Use both **`fscanf()`** and **`fgets()`**

- Use the **`trim_string()`** function

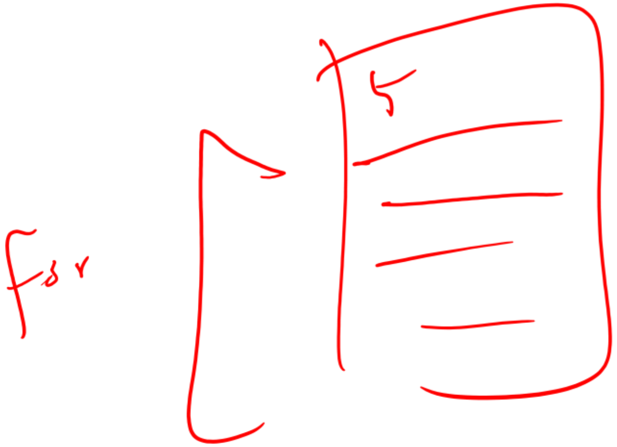
Discussion /1

Recall, there are two common design scenarios:

The number of records is **unknown**



The number of records is **known**



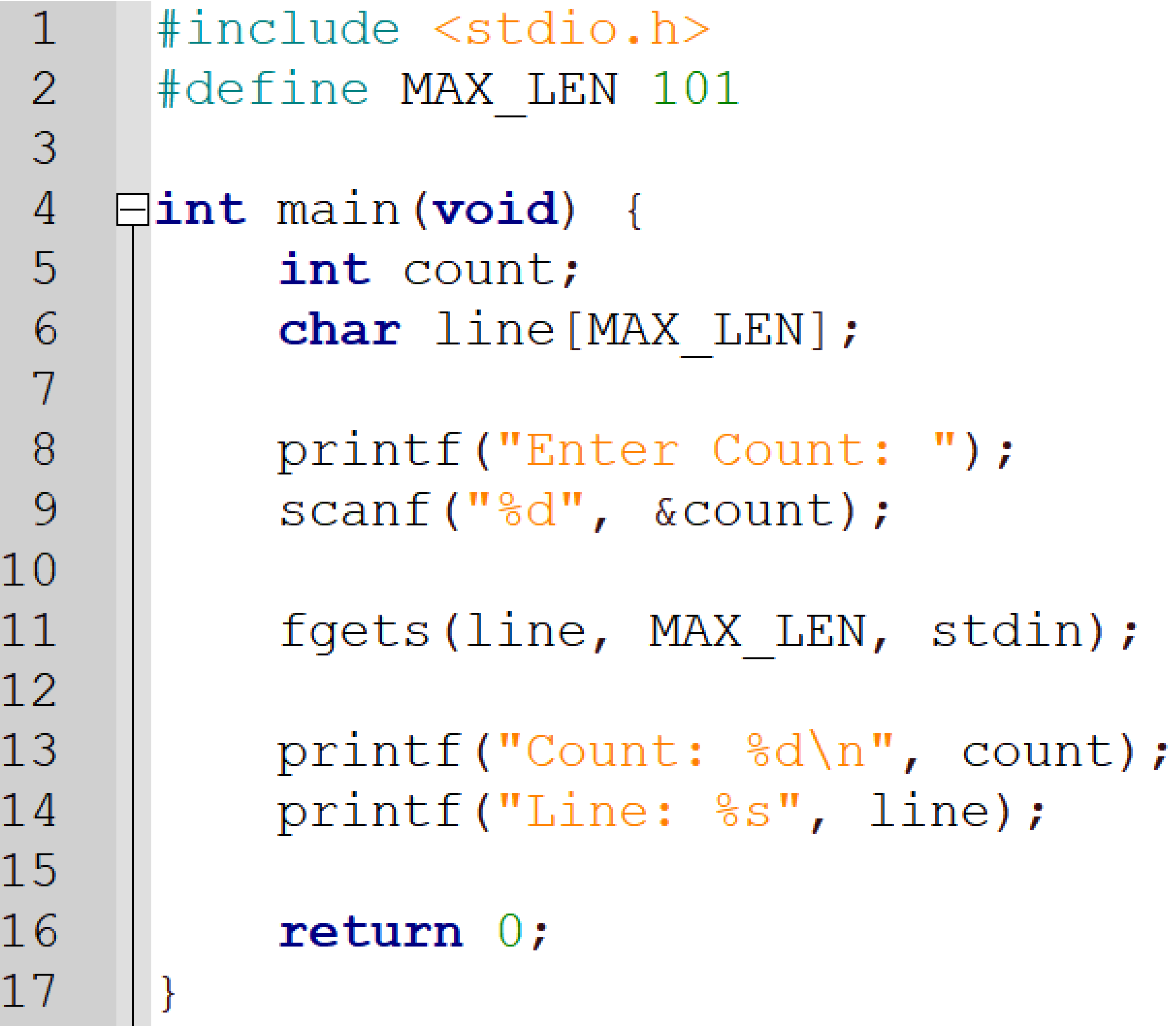
Discussion /2

- A common pitfall occurs when a program reads a number and then immediately reads a string.

- Functions like **`scanf("%d", …)`** leave the trailing newline in the input buffer, which can cause the next string read to be skipped or appear empty.

- The solution here is to clear the buffer first

Example



```c
#include <stdio.h>
#define MAX_LEN 101

int main(void) {
    int count;
    char line[MAX_LEN];

    printf("Enter Count: ");
    scanf("%d", &count);

    fgets(line, MAX_LEN, stdin);

    printf("Count: %d\n", count);
    printf("Line: %s", line);

    return 0;
}
```

Notes /1

- Recall that **`scanf()`** and **`fscanf()`** return a number indicating the total number of successfully read token or **`EOF`**

- On the other hand, **`fgets()`** returns a pointer to the destination buffer if it successfully read at least one character

- Otherwise, it returns a **`NULL`** if end-of-file is encountered or there were any errors

Notes /2

- Our **`clear_buffer()`** helper function only works if the stream is coming from **`stdin`**

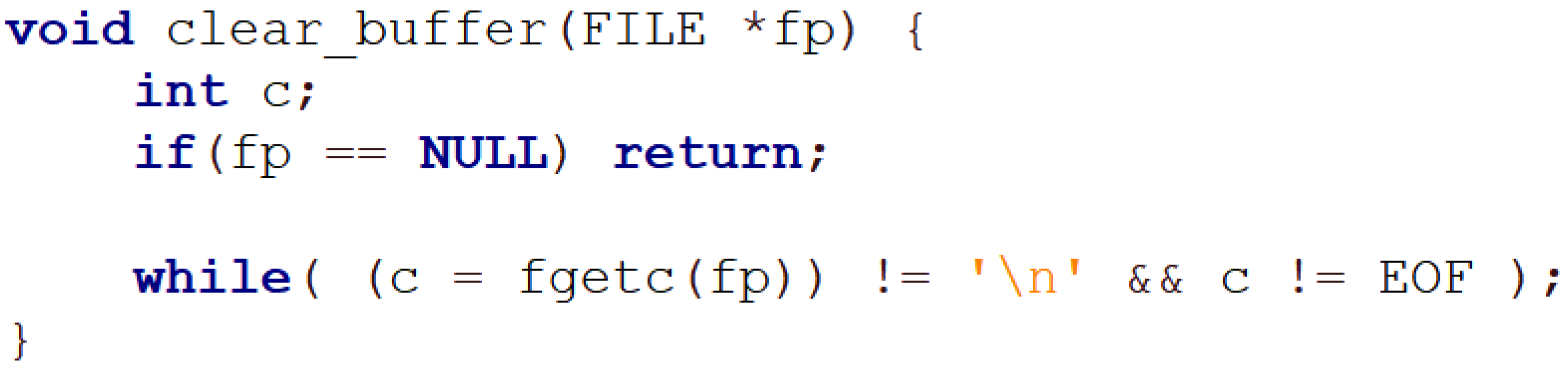
- However, it will not be able to clear the buffer if the input stream is from a file

- The next slide shows an updated and more generic version

Updated Helper Function

The following can handle cases where the stream is coming either from **`stdin`** or a file

`clear_buffer(stdin)` or **`clear_buffer(ifile)`**



```
void clear_buffer(FILE *fp) {
    int c;
    if(fp == NULL) return;

    while( (c = fgetc(fp)) != '\n' && c != EOF );
}
```

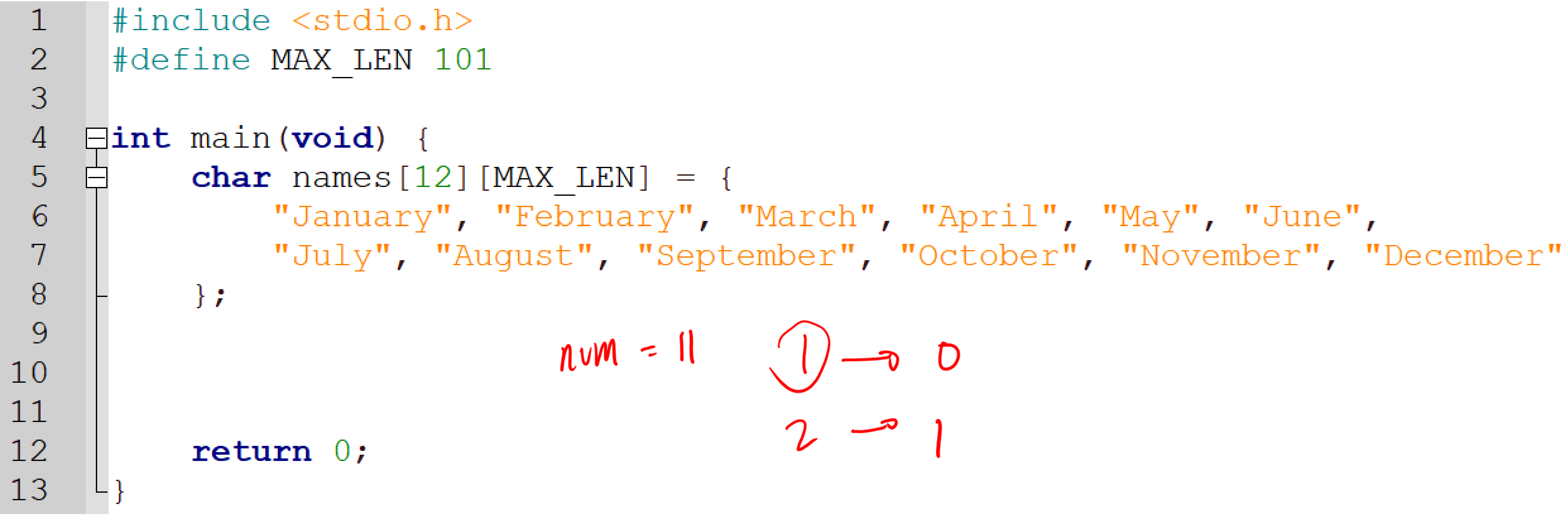
Array of Strings

- Recall that in C, a string is an array of **`char`** terminated by **`'\0'`**

- One way to store multiple strings is an array of arrays of **`char`**

- For now, let’s use a 2D array

Example



```c
#include <stdio.h>
#define MAX_LEN 101

int main(void) {
    char names[12][MAX_LEN] = {
        "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "December"
    };



    return 0;
}
```

num = 11

1 → 0

2 → 1

Practice

Write a program that reads a positive integer representing a month (1–12) and prints the corresponding month name. You may assume the input will always be between 1 and 12, inclusive.

Sample Run

```
Enter Month: 11
November
```

Discussion

How do you print all the values of this list?

More String Functions

- Learn how to read a programming language’s **documentation**
- Try to start searching for a functionality you want to implement
- If it is already existing, use it (i.e., *don’t reinvent the wheel*)

Function	Purpose
`strcpy()`	Makes a copy of source, a string, in the character array accessed by dest:
`strncpy()`	Makes a copy of up to n characters from source in dest: strncpy(dest, source, 5) stores the first five characters of the source and does NOT add a null character.
`strcat()`	Appends source to the end of dest: strcat(dest, source)
`strncat()`	Appends up to n characters of source to end of dest, adding the null character if necessary.
`strcmp()`	Compares s1 and s2 alphabetically. Returns a negative value if s1 should precede s2, a zero if strings are equal, and a positive value if s2 should precede s1 in an alphabetized list. strcmp(s1,s2)
`strncmp()`	Compares the first n characters in s1 and s2 returning positive, zero, and negative values like strcmp.
`strlen()`	Returns the number of characters in s, not counting the terminating null. strlen(s)
`strtok()`	Breaks the parameter string into tokens finding groups of characters separated by any of the delimiter characters. Each group is separated with '\0'.
`strchr()`	Returns a pointer to the first location of a character located in the string. Null is returned if character is not found.
`strpbrk()`	Return a pointer to the first location in the strings that holds any character found in another string.
`strrchr()`	Returns a pointer to the last occurrence of a character in the string. Null is returned if character not found.
`strstr()`	Returns a pointer to the first occurrence of string s2 in string s1. Null is returned if character not found.

Challenge

Write a program that reads a file named `unsorted.txt` containing up to 200 lowercase strings, one per line. Each string contains no whitespace and is at most 100 characters long. The program should then create a new file named `sorted.txt` that contains the same strings sorted in ascending order.

Sample Files

unsorted.txt

```
banana
apple
kiwi
grape
orange
```

sorted.txt

```
apple
banana
grape
kiwi
orange
```

Challenge

Complete the **`remove_whitespaces(str)`** function so that it removes all whitespace characters from the given string. The function should modify the string in place and return the same pointer that was passed to it.

```
char *remove_whitespaces(char *str) {



}
```

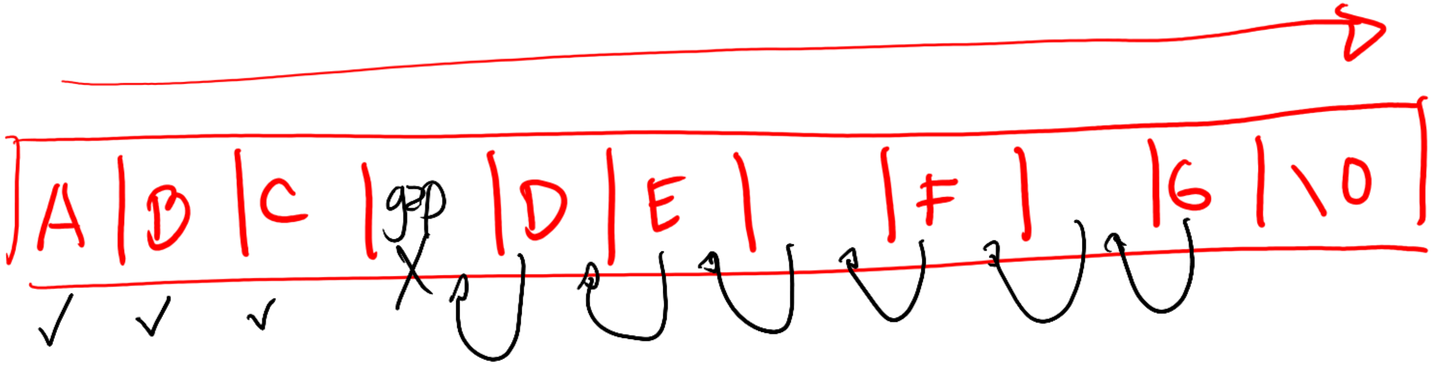
Sample Runs

Run 1

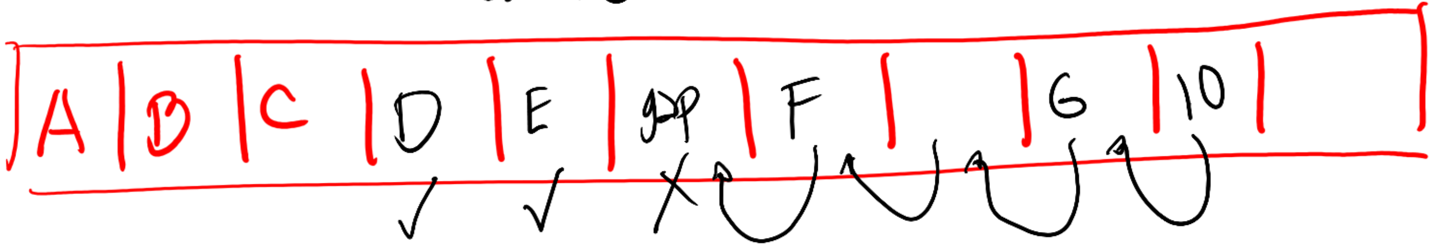
```
Enter Phrase: Hi My Name Is
HiMyNameIs
```

Run 2

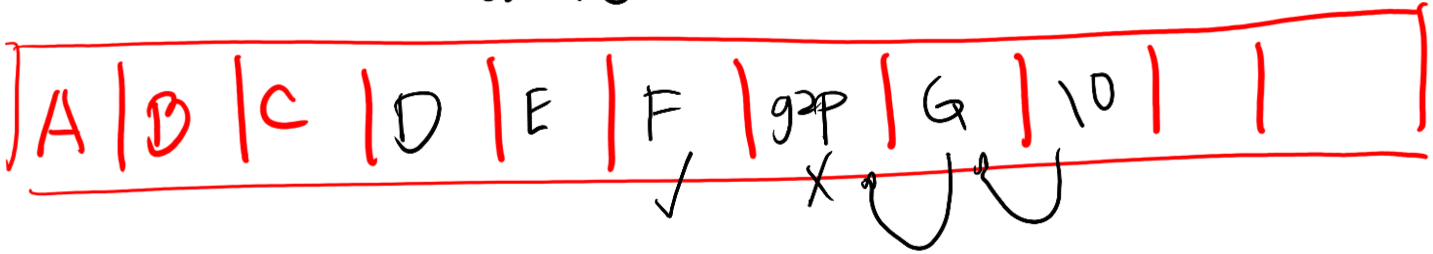
```
Enter Phrase: Hello
Hello
```



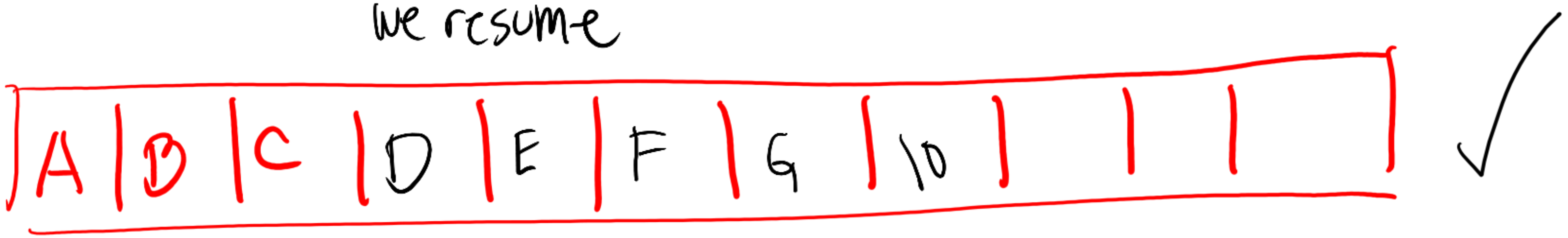
we resume



we resume



we resume



Questions?