CSE 29
Lecture 5 Summary

January 20, 2026

📣 Logistical Things

- Exit slips will now be **online on Gradescope**, not on paper
  - Review checks will still be on paper in the first 5-10 minutes of class
- You will be able to see the code files from class linked on the course website now! You can run these on your own computer and play around with the code!
  - Link for code from today's lecture

Review Questions

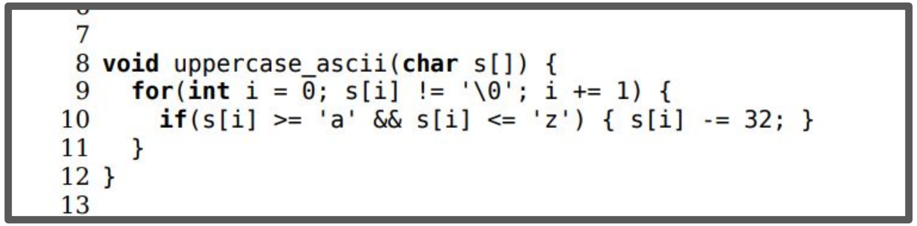
Answers in speaker notes!

Q1: How many UTF-8 code points are encoded by

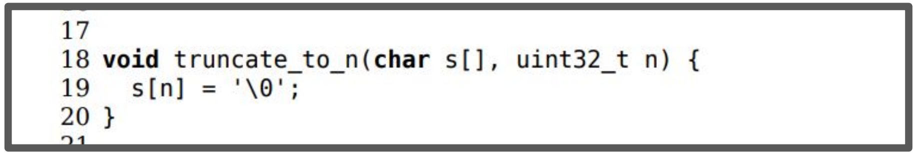
- `0xC4 0x85 0x0E 0xE9 0xA1 0xAA`
- (could also be written as `0xC4850EE9A1AA`)

Q2: Explain in 1 sentence what `uppercase_ascii` does

Q3: Explain in 1 sentence what `truncate_to_n` does



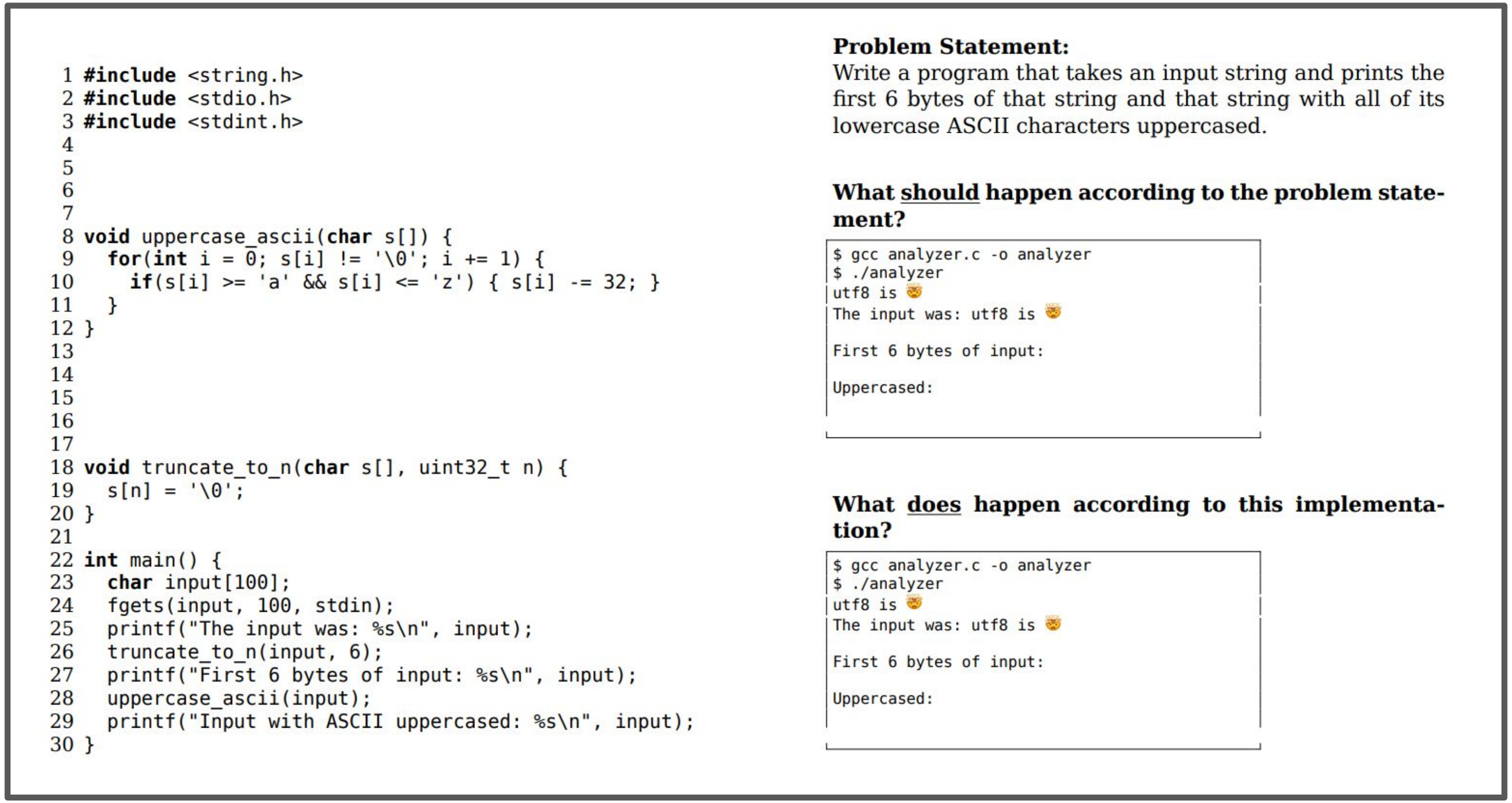
```c

void uppercase_ascii(char s[]) {
  for(int i = 0; s[i] != '\0'; i += 1) {
    if(s[i] >= 'a' && s[i] <= 'z') { s[i] -= 32; }
  }
}

```



```c

void truncate_to_n(char s[], uint32_t n) {
  s[n] = '\0';
}
```

Functions that Modify Strings

Based on the code, answer the following 2 questions



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




void uppercase_ascii(char s[]) {
  for(int i = 0; s[i] != '\0'; i += 1) {
    if(s[i] >= 'a' && s[i] <= 'z') { s[i] -= 32; }
  }
}





void truncate_to_n(char s[], uint32_t n) {
  s[n] = '\0';
}

int main() {
  char input[100];
  fgets(input, 100, stdin);
  printf("The input was: %s\n", input);
  truncate_to_n(input, 6);
  printf("First 6 bytes of input: %s\n", input);
  uppercase_ascii(input);
  printf("Input with ASCII uppercased: %s\n", input);
}
```

Problem Statement:
Write a program that takes an input string and prints the first 6 bytes of that string and that string with all of its lowercase ASCII characters uppercased.

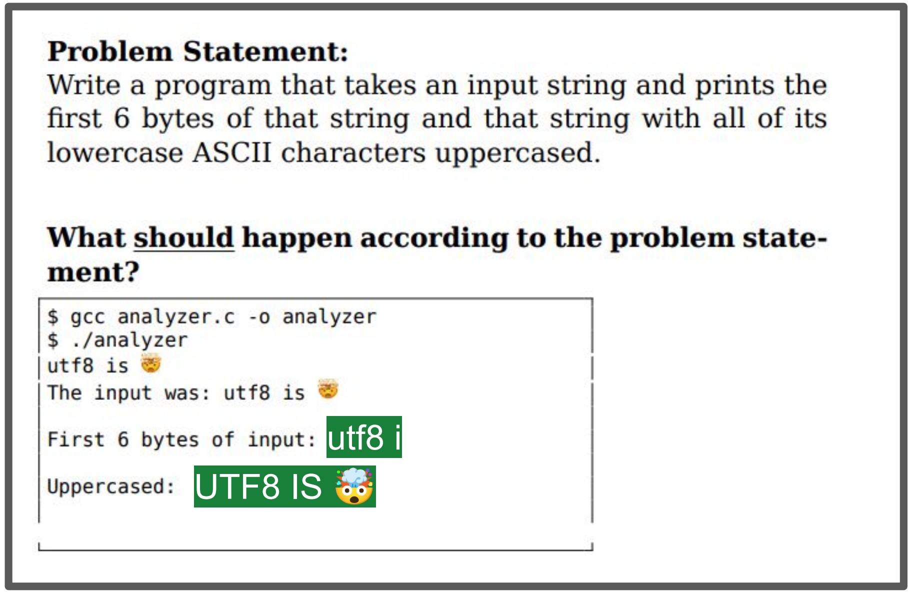
What should happen according to the problem statement?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 🤯
The input was: utf8 is 🤯

First 6 bytes of input:

Uppercased:
```

What does happen according to this implementation?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 🤯
The input was: utf8 is 🤯

First 6 bytes of input:

Uppercased:
```

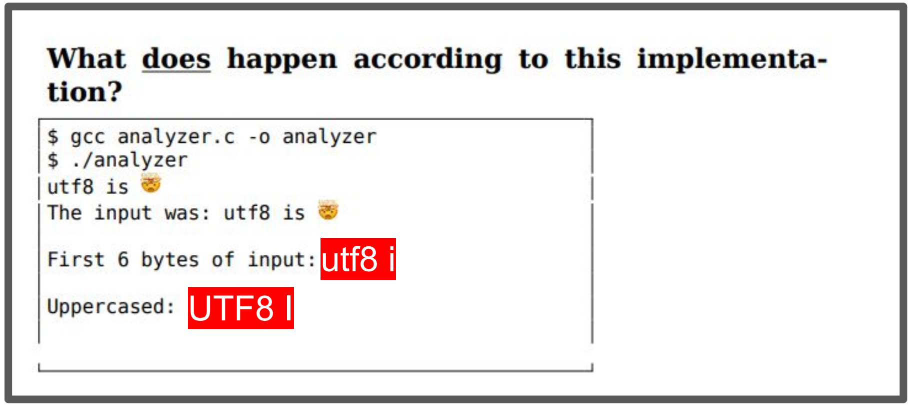
Answers



Problem Statement:
Write a program that takes an input string and prints the first 6 bytes of that string and that string with all of its lowercase ASCII characters uppercased.

What <u>should</u> happen according to the problem statement?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 🤯
The input was: utf8 is 🤯

First 6 bytes of input: utf8 i

Uppercased: UTF8 IS 🤯
```



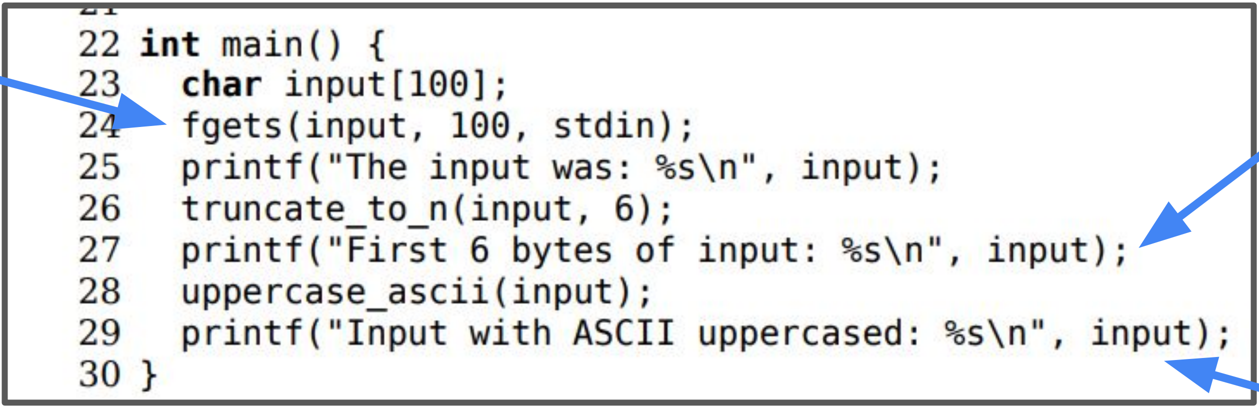
What <u>does</u> happen according to this implementation?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 🤯
The input was: utf8 is 🤯

First 6 bytes of input: utf8 i

Uppercased: UTF8 I
```

What happened?

- We only have one `char[]` string
- After it was modified in `truncate_to_n`, it stayed truncated
- This truncated string was passed into `uppercase_ascii`

`input` starts as “utf8 is 🤯”



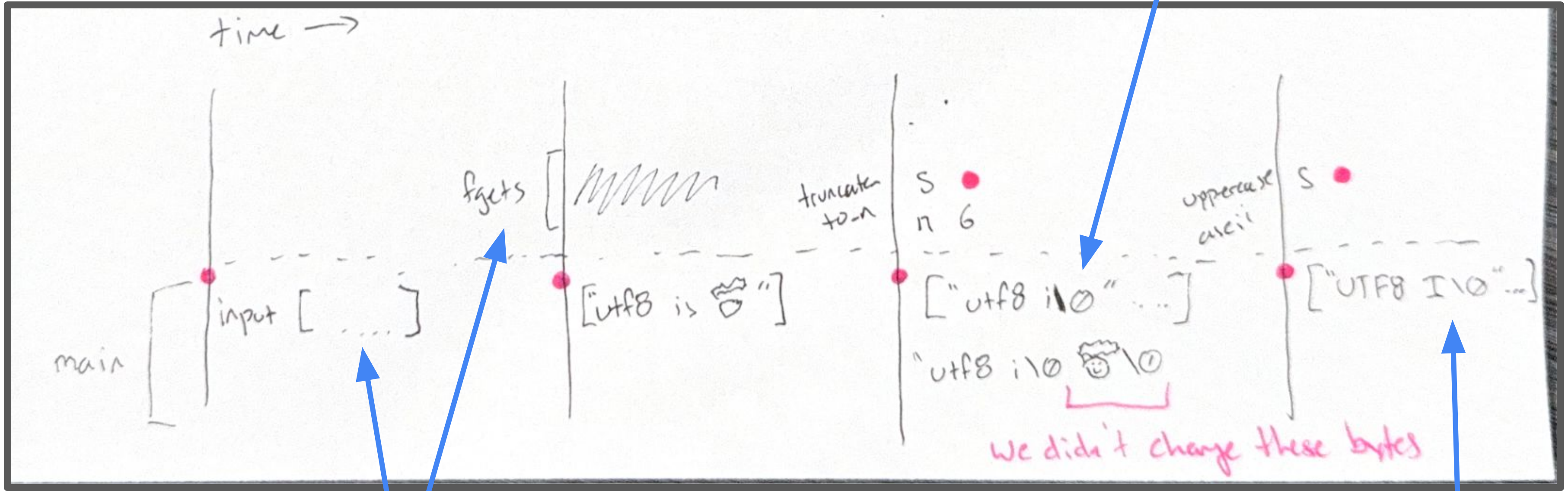
```c
int main() {
  char input[100];
  fgets(input, 100, stdin);
  printf("The input was: %s\n", input);
  truncate_to_n(input, 6);
  printf("First 6 bytes of input: %s\n", input);
  uppercase_ascii(input);
  printf("Input with ASCII uppercased: %s\n", input);
}
```

`input` is now “utf8 i”

`input` is now “UTF8 I”

Program Execution Timeline

truncate_to_n puts a null terminator \0 in the 6th index, leaving everything behind unchanged



input starts as an empty char[] and fgets fills the char[] with user input from the terminal

uppercase_ascii makes all lowercase letters uppercase, up to the null terminator

🤔 Questions & Observations

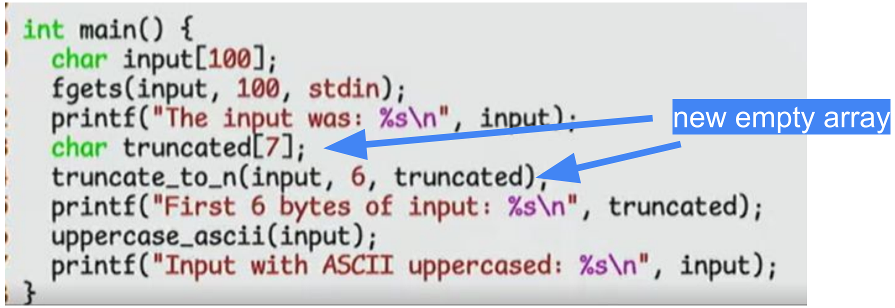
- After you replace the index 6 with null terminator, could you look up the data afterwards?
    - Yes! If you look up indexes after the null terminator, you would still find the remaining bytes. Printing `input[7]` would print out the letter `s`.
- I noticed in the homework that I can't return a `char[]` in my functions
    - That's right! You cannot have `char[]` as a return type, you would get an error.
- After the null terminator, would the stuff afterwards be uppercased or ignored?
    - It will not be uppercase. The loop in the `uppercase_ascii` function will stop once it sees the first null terminator.
- What would `strlen` do after we truncate the string to index 6?
    - It would report 6 characters, `strlen` counts up only to the null terminator

So can we solve this problem?

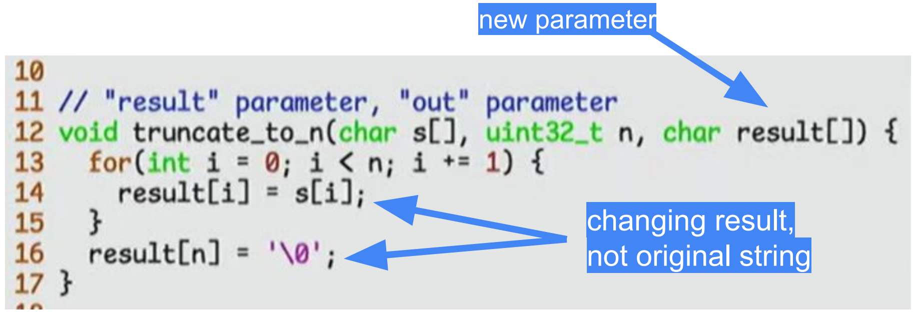
- Yes!
- You might have ran into this problem in PA1
  - Some of you may have used copies of the same string using a for-loop or the `strcpy` function
- Let’s try something better!

Array Return Programming Pattern

- **Problem**: In the C library, there are many functions that need to return a string, but there's no string return type
- **Solution**: A common programming pattern is to include a new parameter in the function that holds the result of function
  - Example: `truncate_to_n` should take in the original `input`, a number `n`, and an empty `result` string as parameters. Then when we want to make modifications to the characters of `input`, we do it in `result` instead!



```c
int main() {
  char input[100];
  fgets(input, 100, stdin);
  printf("The input was: %s\n", input);
  char truncated[7];
  truncate_to_n(input, 6, truncated);
  printf("First 6 bytes of input: %s\n", truncated);
  uppercase_ascii(input);
  printf("Input with ASCII uppercased: %s\n", input);
}
```



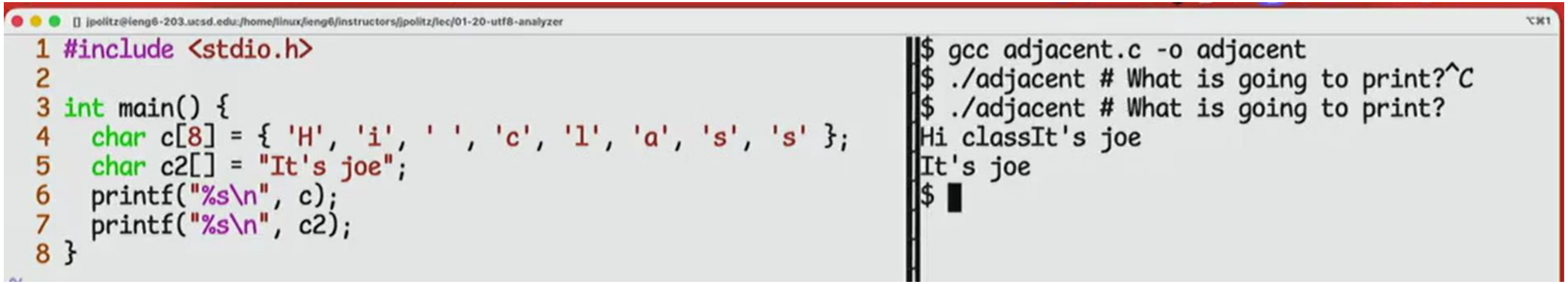
```c
// "result" parameter, "out" parameter
void truncate_to_n(char s[], uint32_t n, char result[]) {
  for(int i = 0; i < n; i += 1) {
    result[i] = s[i];
  }
  result[n] = '\0';
}
```

Questions

- Could I just make a copy of the string in `main` and then pass that in to different functions?
  - Yes you could. But it would basically be the same thing where you would need to declare two `char[]`. And sometimes, you won't need the full copy. For example, in the `truncate_to_n` function, if the original string is 10,000 characters and we truncate to 6, our result array only needs to be 6 characters long.
- Could you just print it in the function?
  - For your PA, you could use that. However, in general, printing in functions makes it less reusable. If we need the result of the function in another part of the program, we wouldn't be able to use it because it's only printed. Overall, it's not a practice we would recommend.
- Could you just create a `char result[]` variable in the function and return it?
  - We can't use `char[]` as a return type, we would get a syntax error

Variables Stored in Memory

Look at this program’s output. What is off about this?



[] jpolitz@ieng6-203.ucsd.edu:/home/linux/ieng6/instructors/jpolitz/lec/01-20-utf8-analyzer

```c
#include <stdio.h>

int main() {
  char c[8] = { 'H', 'i', ' ', 'c', 'l', 'a', 's', 's' };
  char c2[] = "It's joe";
  printf("%s\n", c);
  printf("%s\n", c2);
}
```

```
$ gcc adjacent.c -o adjacent
$ ./adjacent # What is going to print?^C
$ ./adjacent # What is going to print?
Hi classIt's joe
It's joe
$ 
```

adjacent.c

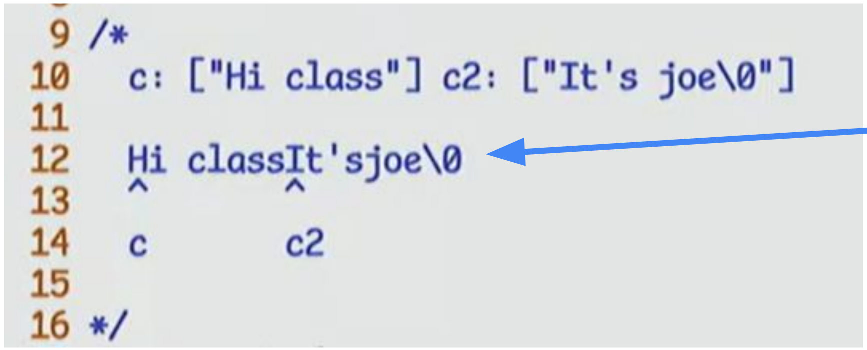
Terminal

How did Joe split this screen to look like this? Look at the end of the slides to find out! 🤓

What’s happening?

What’s happening on line 6 where it prints “HiclassIt’sjoe” ?

- This is what happens when you leave off the null terminator
- `printf` prints all the way until it sees a null terminator, but `c` doesn’t have one.
- gcc (the compiler) chose to store the data of `c` and `c2` one after the other
- So when we ask `printf` to print `c`, it goes from `H` to the null terminator

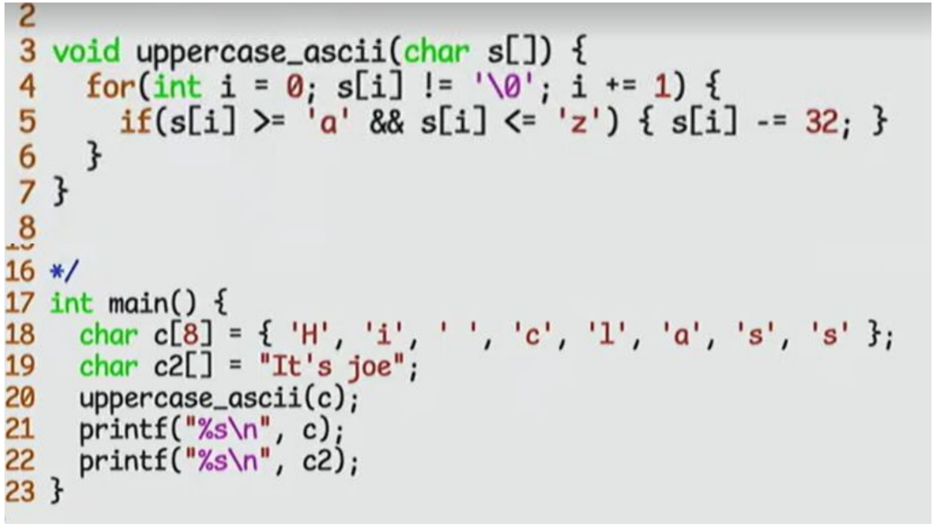


```
/*
  c: ["Hi class"] c2: ["It's joe\0"]

  Hi classIt'sjoe\0
  ^       ^
  c       c2

*/
```

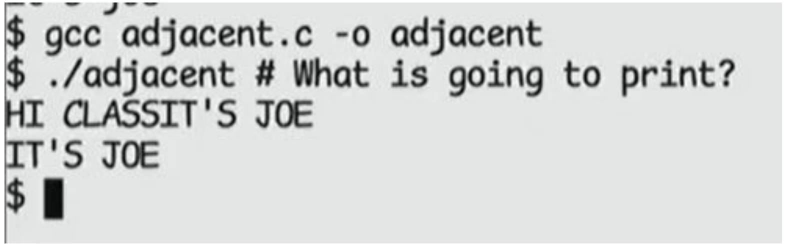
What it looks like in memory

Does this also apply to `uppercase_ascii`?

- Yes! `uppercase_ascii` has a for-loop where the ending condition looks for the null terminator
- It affects both `c` and `c2`



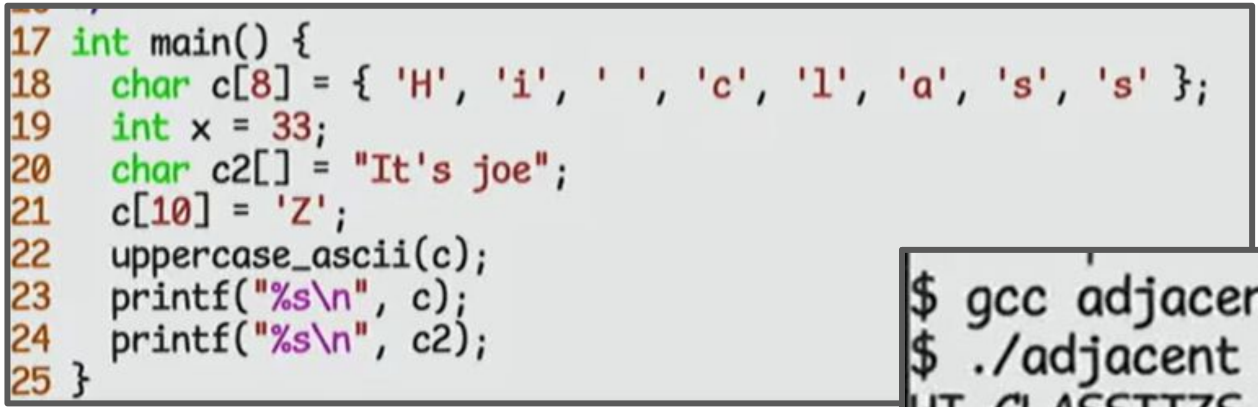
```c

void uppercase_ascii(char s[]) {
  for(int i = 0; s[i] != '\0'; i += 1) {
    if(s[i] >= 'a' && s[i] <= 'z') { s[i] -= 32; }
  }
}

*/
int main() {
  char c[8] = { 'H', 'i', ' ', 'c', 'l', 'a', 's', 's' };
  char c2[] = "It's joe";
  uppercase_ascii(c);
  printf("%s\n", c);
  printf("%s\n", c2);
}
```



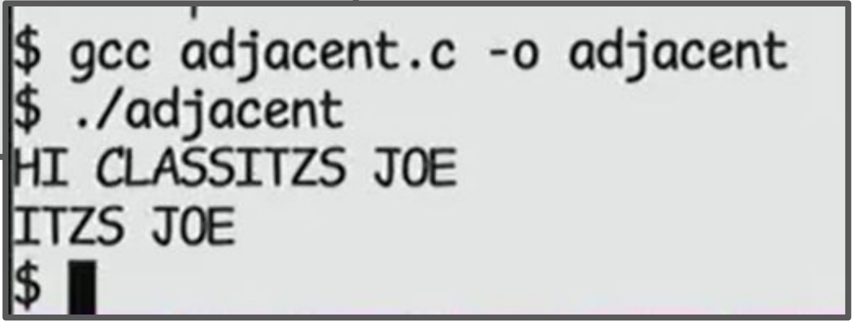
```
$ gcc adjacent.c -o adjacent
$ ./adjacent # What is going to print?
HI CLASSIT'S JOE
IT'S JOE
$ 
```

C lets you access things that you can’t in Java and Python

- We can access `c[10]` even though `c` was only declared with 8 characters.



```c
int main() {
  char c[8] = { 'H', 'i', ' ', 'c', 'l', 'a', 's', 's' };
  int x = 33;
  char c2[] = "It's joe";
  c[10] = 'Z';
  uppercase_ascii(c);
  printf("%s\n", c);
  printf("%s\n", c2);
}
```

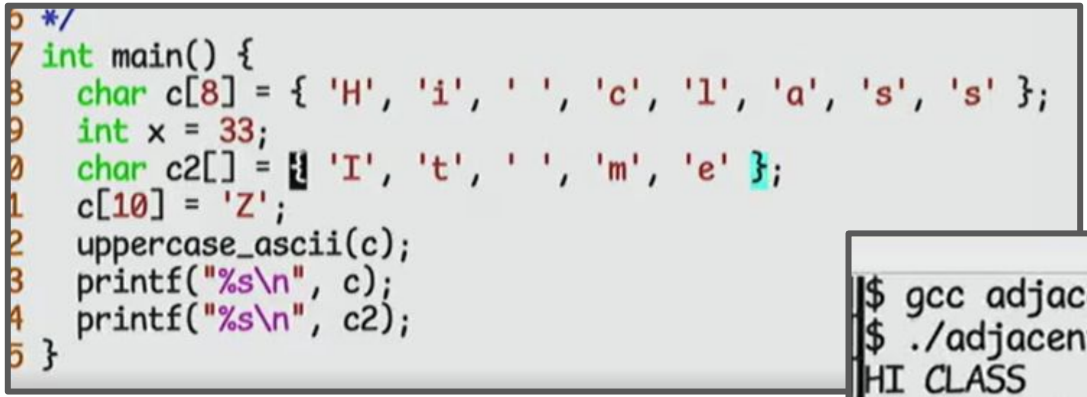


```
$ gcc adjacent.c -o adjacent
$ ./adjacent
HI CLASSITZS JOE
ITZS JOE
$ 
```

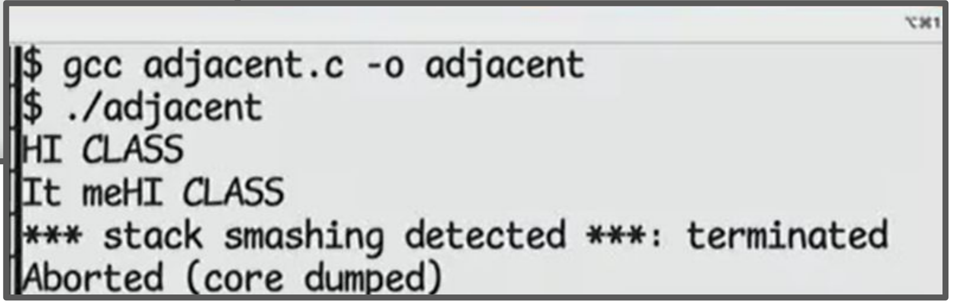
🤔 Questions & Observations

- If you declare multiple variables in a function, are they stored next to each other?
    - gcc will put variables near each other in storage
- If you put another random variable in the middle of c and c2, what would happen?
    - Joe put an `int` in between `c` and `c2`, and it didn't do anything apparently. Interesting 🤔
- Is this behavior reliable? If i compile on different computers or compilers, would the behavior be the same?
    - gcc will find a spot but there is no guarantee of the relative order of how things are stored.
- After the program is done running, what happens to everything in memory?
    - When the program starts, the operating system allocates memory to the program. When the program stops, the operating system takes back the allocated memory.

What would happen if c2 didn't have a null terminator?



```
*/
int main() {
  char c[8] = { 'H', 'i', ' ', 'c', 'l', 'a', 's', 's' };
  int x = 33;
  char c2[] = { 'I', 't', ' ', 'm', 'e' };
  c[10] = 'Z';
  uppercase_ascii(c);
  printf("%s\n", c);
  printf("%s\n", c2);
}
```

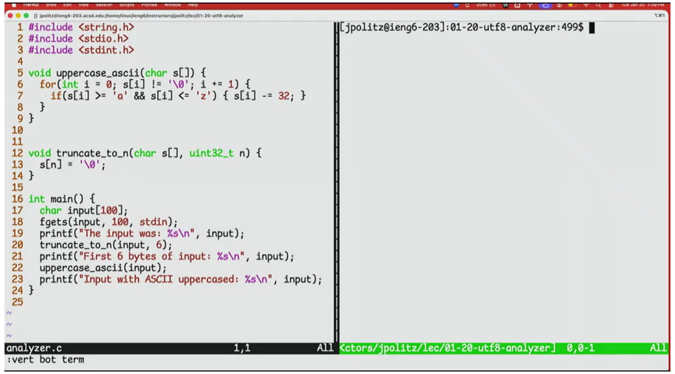


```
$ gcc adjacent.c -o adjacent
$ ./adjacent
HI CLASS
It meHI CLASS
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

We get errors! We started accessing things in memory that we shouldn't be allowed to access/modify.

💡 Pro Vim Tip: Terminal/Vim Split Screen

- In vim normal mode (press ESC while in vim), enter `:vert bot term`
- This will split the term vertically (`vert`) and open a terminal (`term`) on the right (`bot`) of the screen
- You will have a terminal and a file side by side
- Ctrl+W+W moves you back and forth between the two screens
- Type `exit` in the terminal side to get rid of the terminal



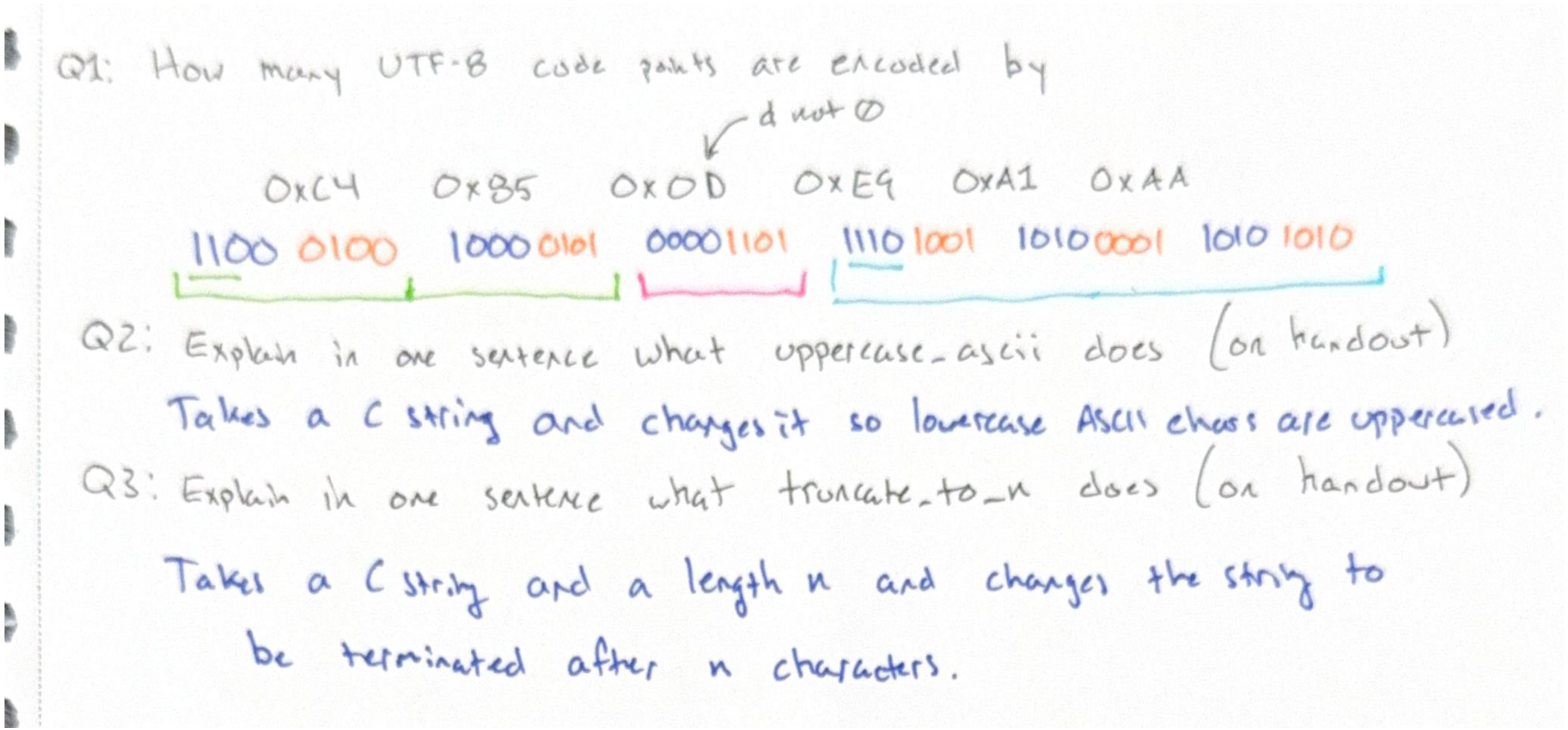
```
jpolitz@ieng6-203.ucsd.edu:/home/linux/ieng6/instructors/jpolitz/lec/01-20-utf8-analyzer
```

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

void uppercase_ascii(char s[]) {
  for(int i = 0; s[i] != '\0'; i += 1) {
    if(s[i] >= 'a' && s[i] <= 'z') { s[i] -= 32; }
  }
}


void truncate_to_n(char s[], uint32_t n) {
  s[n] = '\0';
}

int main() {
  char input[100];
  fgets(input, 100, stdin);
  printf("The input was: %s\n", input);
  truncate_to_n(input, 6);
  printf("First 6 bytes of input: %s\n", input);
  uppercase_ascii(input);
  printf("Input with ASCII uppercased: %s\n", input);
}

~
~
~
analyzer.c                                   1,1          All
:vert bot term
```

```
[jpolitz@ieng6-203]:01-20-utf8-analyzer:499$

<ctors/jpolitz/lec/01-20-utf8-analyzer]  0,0-1          All
```

Especially helpful in PrairieLearn workspaces!

Joe's Notes (11am)



Q1: How many UTF-8 code points are encoded by

d not 0

0xC4 0x85 0x0D 0xE9 0xA1 0xAA

1100 0100 1000 0101 0000 1101 1110 1001 1010 0001 1010 1010

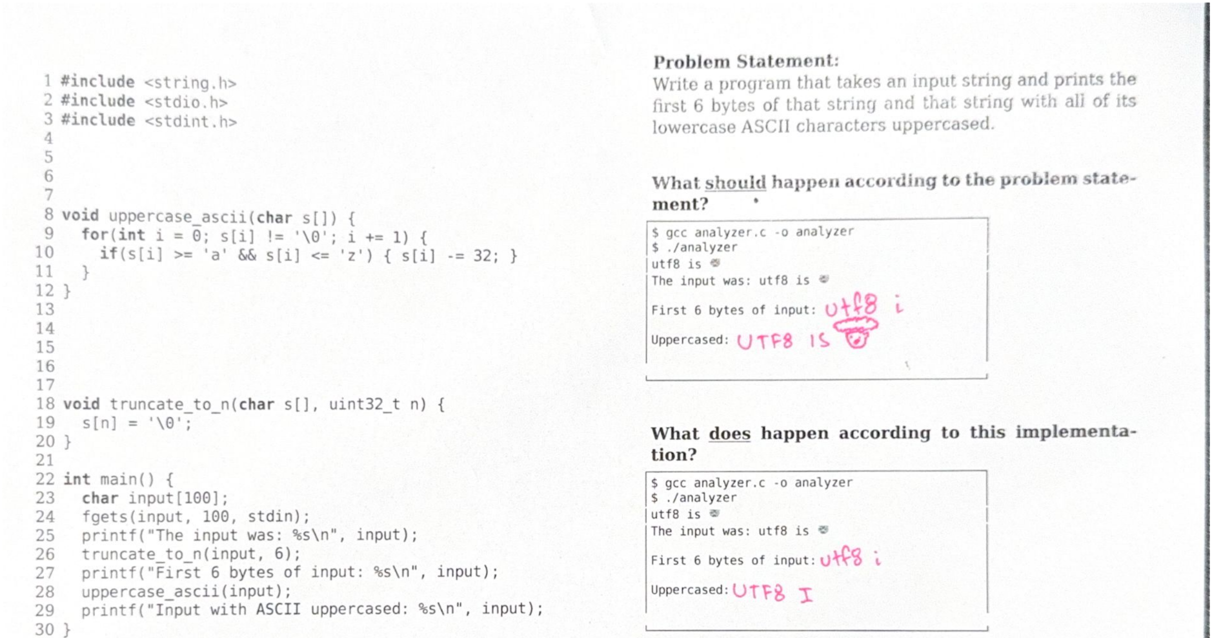
Q2: Explain in one sentence what uppercase_ascii does (on handout)

Takes a C string and changes it so lowercase ASCII chars are uppercased.

Q3: Explain in one sentence what truncate_to_n does (on handout)

Takes a C string and a length n and changes the string to be terminated after n characters.

Joe's Notes (11am)



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




void uppercase_ascii(char s[]) {
  for(int i = 0; s[i] != '\0'; i += 1) {
    if(s[i] >= 'a' && s[i] <= 'z') { s[i] -= 32; }
  }
}





void truncate_to_n(char s[], uint32_t n) {
  s[n] = '\0';
}

int main() {
  char input[100];
  fgets(input, 100, stdin);
  printf("The input was: %s\n", input);
  truncate_to_n(input, 6);
  printf("First 6 bytes of input: %s\n", input);
  uppercase_ascii(input);
  printf("Input with ASCII uppercased: %s\n", input);
}
```

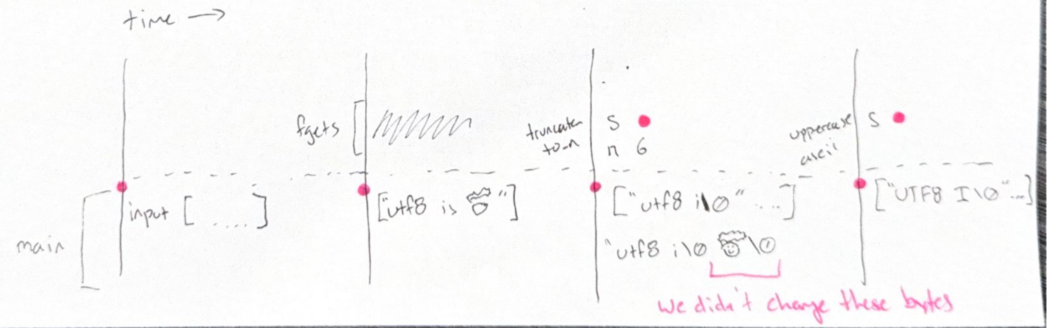
Problem Statement:
Write a program that takes an input string and prints the first 6 bytes of that string and that string with all of its lowercase ASCII characters uppercased.

What should happen according to the problem statement?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 😊
The input was: utf8 is 😊

First 6 bytes of input: utf8 i

Uppercased: UTF8 IS 😊
```

What does happen according to this implementation?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 😊
The input was: utf8 is 😊

First 6 bytes of input: utf8 i

Uppercased: UTF8 I
```



Joe's Notes (12:30pm)

0xC485 0EE9 A1AA

B 1

Q1: How many UTF-8 code points are encoded by:

0xC4 0x85 0x0E 0xE9 0xA1 0xAA

$\boxed{11000100 \quad 10000101}$ $\boxed{00001110}$ $\boxed{11101001 \quad 10100001 \quad 10101010}$

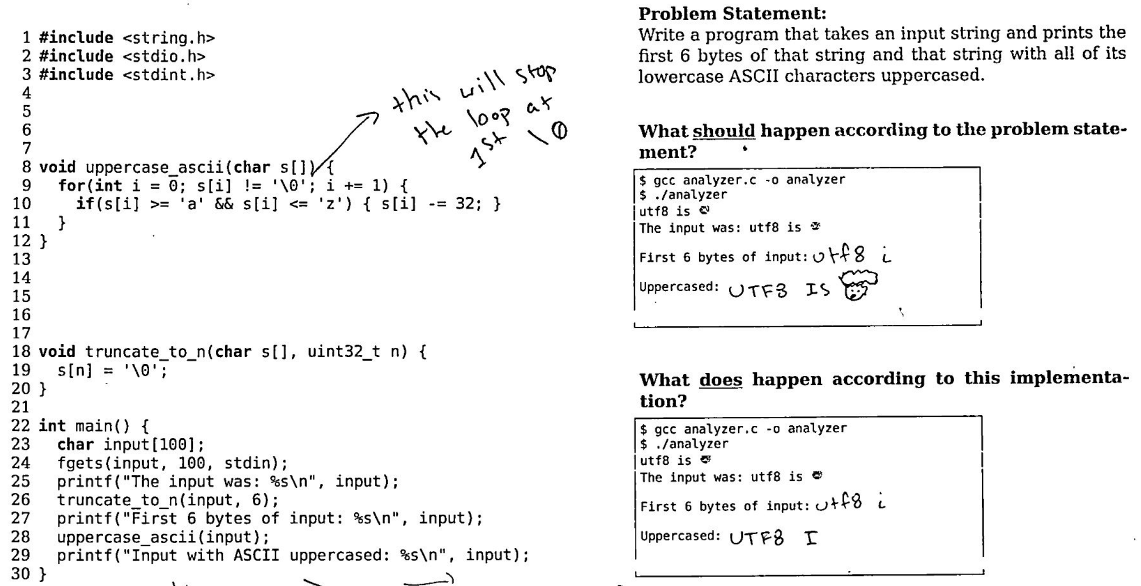
Q2: ~~[illegible]~~ Explain in 1 sentence what uppercase_ascii does (on handout)

Takes a C string and changes the string to replace all lowercase ASCII characters with their uppercase equivalent

Q3: Explain in 1 sentence what truncate_to_n does (on handout)

Takes a C string s and a length n and ~~ma~~ changes s by setting the nth index to a null terminator.

Joe’s Notes (12:30pm)



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




void uppercase_ascii(char s[]) {
  for(int i = 0; s[i] != '\0'; i += 1) {
    if(s[i] >= 'a' && s[i] <= 'z') { s[i] -= 32; }
  }
}





void truncate_to_n(char s[], uint32_t n) {
  s[n] = '\0';
}

int main() {
  char input[100];
  fgets(input, 100, stdin);
  printf("The input was: %s\n", input);
  truncate_to_n(input, 6);
  printf("First 6 bytes of input: %s\n", input);
  uppercase_ascii(input);
  printf("Input with ASCII uppercased: %s\n", input);
}
```

this will stop the loop at 1st \0

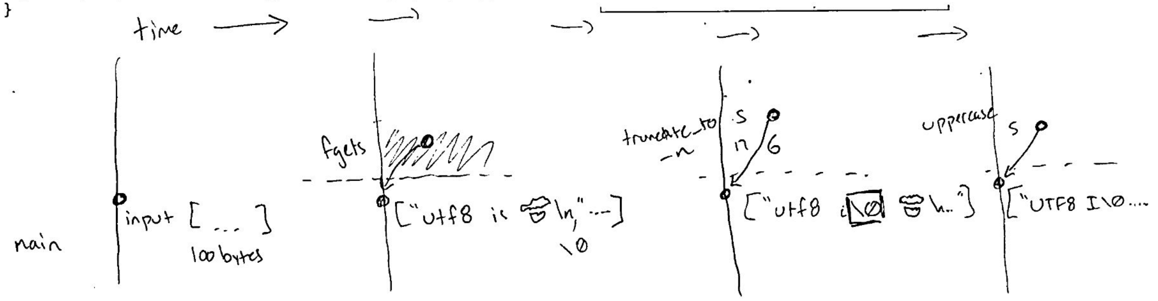
Problem Statement:
Write a program that takes an input string and prints the first 6 bytes of that string and that string with all of its lowercase ASCII characters uppercased.

What should happen according to the problem statement?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 🧑‍🍳
The input was: utf8 is 🧑‍🍳

First 6 bytes of input: utf8 i

Uppercased: UTF8 IS 🧑‍🍳
```

What does happen according to this implementation?

```
$ gcc analyzer.c -o analyzer
$ ./analyzer
utf8 is 🧑‍🍳
The input was: utf8 is 🧑‍🍳

First 6 bytes of input: utf8 i

Uppercased: UTF8 I
```



time
main
input [...]
100 bytes
fgets
["utf8 is 🧑‍🍳\n" ---]
\0
truncate_to_n
s
n
6
["utf8 i\0 🧑‍🍳\n..."]
uppercase
s
["UTF8 I\0 ....