CSE 29
Lecture 3 Summary

January 13, 2026

📣 Logistical Q&A

> 💡 **Tip**
> In some of the slides, there will be additional information in the **speaker notes**. Don't forget to check!

- Should you expect to be able to do all of assignment 1 right now?
  - Not quite since the beginning of the quarter covers a large breadth but you can definitely get started.
  - As we go along you become more prepared to do the assignment when it is released.
- How does lecture makeup work?
  - Complete the makeup assignment on gradescope before 11am the day of the next lecture to "1 point back no matter how much you missed" - Joe
- How are PAs graded?
  - Mostly autograded in Gradescope, we may grade on code quality. Design questions are a separate Gradescope assignment
- How does the Financial Aid Commencement of Activity work?
  - ANY activity you have done for this class either online or in lecture on paper will cover this

🧠 Review Questions

1. What does this print?

```
char s[] = "cse29"
printf("%c %d", s[3], s[3])
```

2. Fill in the blank to print 0

```
char s2[] = "cse30 next"
printf("%d", s2[<fill in>])
```

3. Fill in the blank to print 0

```
char s2[] = "cse30 next"
printf("%c", s2[<fill in>])
```

Try these questions out yourself before looking at the answers! Answers in speaker notes of this slide 👇

Answer 1: 2 50
(2 comes from the 3rd index of “cse29” and 50 comes from the ASCII integer representation the character “2”)

Answer 2: 10
(s2[10] holds a null terminator, which is 0 in ASCII integer representation)

Answer 3: 4
(s2[4] refers to the “0” character)

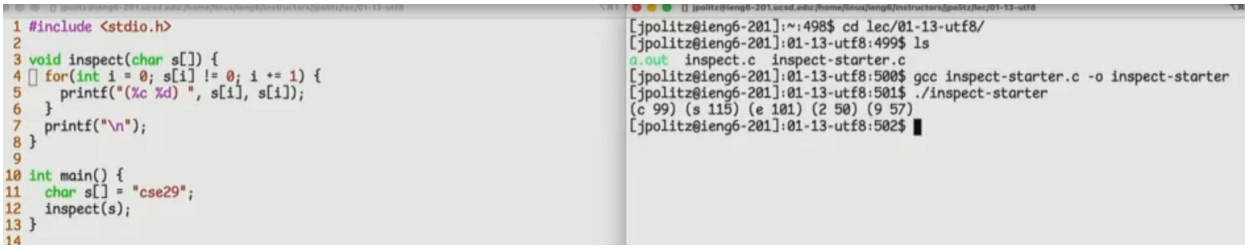
Non-ASCII Characters

So far, we've been working with characters that are represented in as integers between 0-127 using ASCII encoding standards (refer back to the ASCII table in last week's handouts)

But what about characters we don't see on the ASCII table? (Cyrillic characters, Chinese characters, emojis 🐛)

Running `inspect` on ASCII characters

We see the compilation and running of our program, which inspects “cse29” so we see pairs of each character and it’s decimal value.



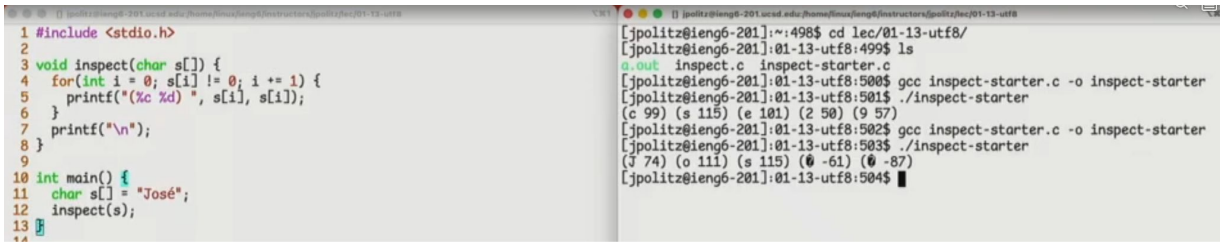
```c
#include <stdio.h>

void inspect(char s[]) {
  for(int i = 0; s[i] != 0; i += 1) {
    printf("(%c %d) ", s[i], s[i]);
  }
  printf("\n");
}

int main() {
  char s[] = "cse29";
  inspect(s);
}
```

```
[jpolitz@ieng6-201]:~:498$ cd lec/01-13-utf8/
[jpolitz@ieng6-201]:01-13-utf8:499$ ls
a.out  inspect.c  inspect-starter.c
[jpolitz@ieng6-201]:01-13-utf8:500$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:501$ ./inspect-starter
(c 99) (s 115) (e 101) (2 50) (9 57)
[jpolitz@ieng6-201]:01-13-utf8:502$
```

Running the `inspect` function on non-ASCII characters

We see here that printing é's info in our inspect function results in two odd symbols with "?"s in them with negative values.

This will later be revealed to be the result of utf-8 encoding and the use of 2's complement for negative numbers



```
#include <stdio.h>

void inspect(char s[]) {
  for(int i = 0; s[i] != 0; i += 1) {
    printf("(%c %d) ", s[i], s[i]);
  }
  printf("\n");
}

int main() {
  char s[] = "José";
  inspect(s);
}
```

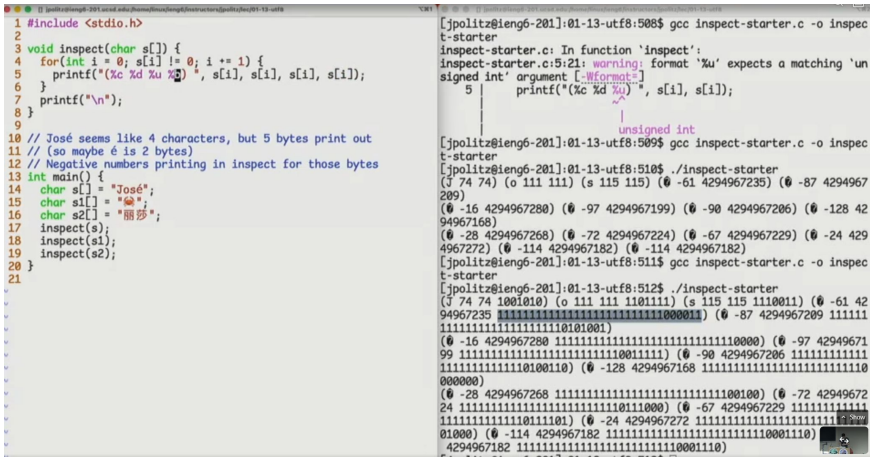
```
[jpolitz@ieng6-201]:~:498$ cd lec/01-13-utf8/
[jpolitz@ieng6-201]:01-13-utf8:499$ ls
a.out  inspect.c  inspect-starter.c
[jpolitz@ieng6-201]:01-13-utf8:500$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:501$ ./inspect-starter
(c 99) (s 115) (e 101) (2 50) (9 57)
[jpolitz@ieng6-201]:01-13-utf8:502$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:503$ ./inspect-starter
(J 74) (o 111) (s 115) (? -61) (? -87)
[jpolitz@ieng6-201]:01-13-utf8:504$
```

More format specifiers

- %b - binary representation (prints as 4 bytes)
  - ‘é’ is 11111111111111111111111111000011 11111111111111111111111110101001
- %u - unsigned number representation
  - ‘é’ is 4294967235 4294967209
- %hhb - single-byte
  - ‘é’ is 11000011 10101001
- %hhu - single-byte unsigned char as a number
  - ‘é’ is 195 169

hh stands for half-half
default is 32 bits, half of 32 is 16, half of 16 is 8

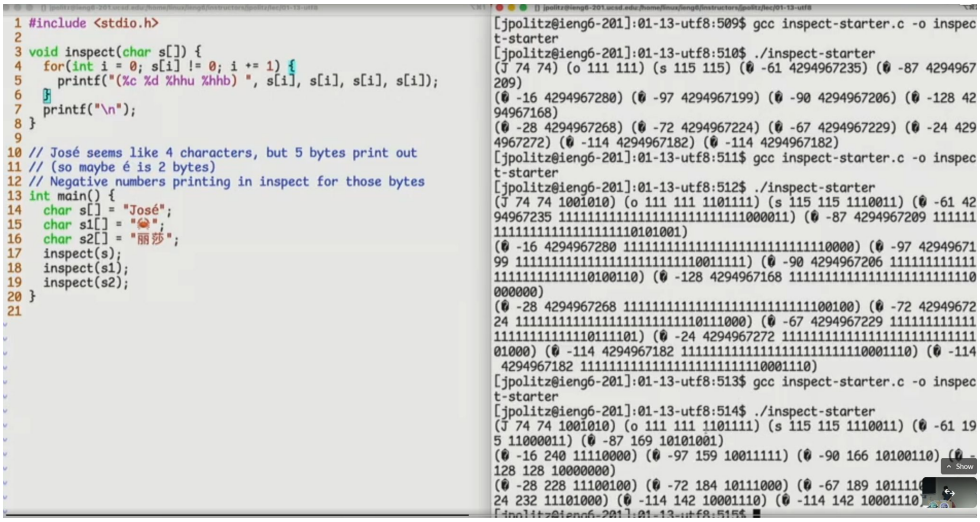
Printing more information with different non-ASCII characters



```c
#include <stdio.h>

void inspect(char s[]) {
  for(int i = 0; s[i] != 0; i += 1) {
    printf("(%c %d %u %x) ", s[i], s[i], s[i], s[i]);
  }
  printf("\n");
}

// José seems like 4 characters, but 5 bytes print out
// (so maybe é is 2 bytes)
// Negative numbers printing in inspect for those bytes
int main() {
  char s[] = "José";
  char s1[] = "☕";
  char s2[] = "丽莎";
  inspect(s);
  inspect(s1);
  inspect(s2);
}
```

```
[jpolitz@ieng6-201]:01-13-utf8:508$ gcc inspect-starter.c -o inspect-starter
inspect-starter.c: In function 'inspect':
inspect-starter.c:5:21: warning: format '%u' expects a matching 'unsigned int' argument [-Wformat=]
    5 |     printf("(%c %d %u %x) ", s[i], s[i], s[i], s[i]);
      |                     ^~
      |                     |
      |                     unsigned int
[jpolitz@ieng6-201]:01-13-utf8:509$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:510$ ./inspect-starter
(J 74 74 4a) (o 111 111 6f) (s 115 115 73) (é -61 4294967235 ffffffc3) (é -87 4294967209 ffffffa9)
(☕ -16 4294967280 fffffff0) (☕ -97 4294967199 ffffffa1) (☕ -90 4294967206 ffffffa6) (☕ -128 4294967168 ffffff80)
(丽 -28 4294967268 ffffffe4) (丽 -72 4294967224 ffffffb8) (丽 -67 4294967229 ffffffbd) (丽 -24 4294967272 ffffffe8)
(莎 -28 4294967268 ffffffe4) (莎 -114 4294967182 ffffff8e) (莎 -114 4294967182 ffffff8e)
[jpolitz@ieng6-201]:01-13-utf8:511$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:512$ ./inspect-starter
(J 74 74 1001010) (o 111 111 1101111) (s 115 115 1110011) (é -61 4294967235 11111111111111111111111111000011) (é -87 4294967209 11111111111111111111111110101001)
(☕ -16 4294967280 11111111111111111111111111110000) (☕ -97 4294967199 11111111111111111111111110011111) (☕ -90 4294967206 11111111111111111111111110100110) (☕ -128 4294967168 11111111111111111111111110000000)
(丽 -28 4294967268 11111111111111111111111111100100) (丽 -72 4294967224 11111111111111111111111111000) (丽 -67 4294967229 11111111111111111111111110111101) (丽 -24 4294967272 11111111111111111111111111101000) (莎 -28 4294967268 11111111111111111111111111100100) (莎 -114 4294967182 11111111111111111111111110001110) (莎 4294967182 11111111111111111111111110001110)
```

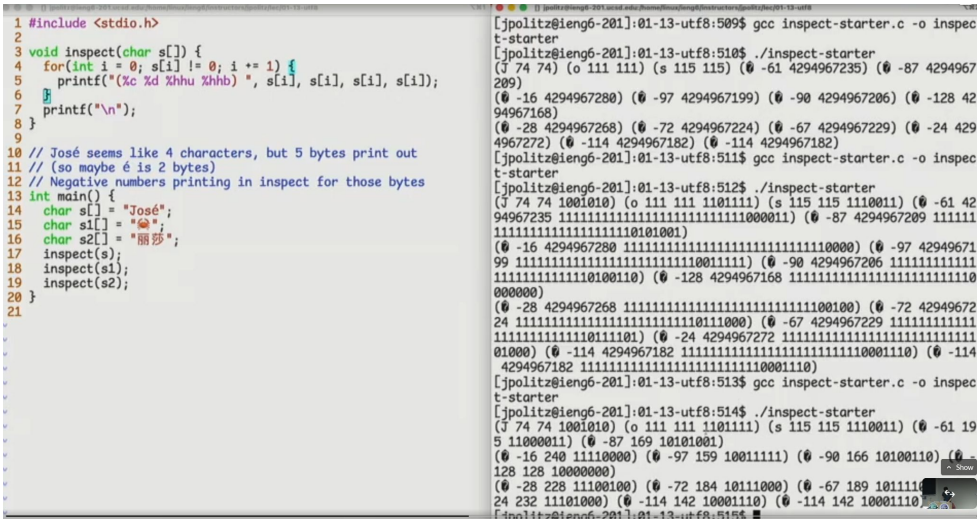
Printing with the addition of hh for “half half” so it only prints as 1 byte



```c
#include <stdio.h>

void inspect(char s[]) {
  for(int i = 0; s[i] != 0; i += 1) {
    printf("(%c %d %hhu %hhb) ", s[i], s[i], s[i], s[i]);
  }
  printf("\n");
}

// José seems like 4 characters, but 5 bytes print out
// (so maybe é is 2 bytes)
// Negative numbers printing in inspect for those bytes
int main() {
  char s[] = "José";
  char s1[] = "🍔";
  char s2[] = "丽莎";
  inspect(s);
  inspect(s1);
  inspect(s2);
}
```

```
[jpolitz@ieng6-201]:01-13-utf8:509$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:510$ ./inspect-starter
(J 74 74) (o 111 111) (s 115 115) (@ -61 4294967235) (@ -87 4294967
209)
(@ -16 4294967280) (@ -97 4294967199) (@ -90 4294967206) (@ -128 42
94967168)
(@ -28 4294967268) (@ -72 4294967224) (@ -67 4294967229) (@ -24 429
4967272) (@ -114 4294967182) (@ -114 4294967182)
[jpolitz@ieng6-201]:01-13-utf8:511$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:512$ ./inspect-starter
(J 74 74 1001010) (o 111 111 1101111) (s 115 115 1110011) (@ -61 42
94967235 11111111111111111111111111000011) (@ -87 4294967209 111111
11111111111111110101001)
(@ -16 4294967280 11111111111111111111111110000) (@ -97 42949671
99 11111111111111111111111110011111) (@ -90 4294967206 111111111111
11111111111110100110) (@ -128 4294967168 11111111111111111111111110
000000)
(@ -28 4294967268 11111111111111111111111100100) (@ -72 42949672
24 11111111111111111111111110111000) (@ -67 4294967229 111111111111
11111111111110111101) (@ -24 4294967272 11111111111111111111111111
01000) (@ -114 4294967182 11111111111111111111110001110) (@ -114
 4294967182 11111111111111111111111110001110)
[jpolitz@ieng6-201]:01-13-utf8:513$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:514$ ./inspect-starter
(J 74 74 1001010) (o 111 111 1101111) (s 115 115 1110011) (@ -61 19
5 11000011) (@ -87 169 10101001)
(@ -16 240 11110000) (@ -97 159 10011111) (@ -90 166 10100110) (@ -
128 128 10000000)
(@ -28 228 11100100) (@ -72 184 10111000) (@ -67 189 1011110
24 232 11101000) (@ -114 142 10001110) (@ -114 142 10001110)
[jpolitz@ieng6-201]:01-13-utf8:515$
```

What observations or questions do you have about this?



```c
#include <stdio.h>

void inspect(char s[]) {
  for(int i = 0; s[i] != 0; i += 1) {
    printf("(%c %d %hhu %hhb) ", s[i], s[i], s[i], s[i]);
  }
  printf("\n");
}

// José seems like 4 characters, but 5 bytes print out
// (so maybe é is 2 bytes)
// Negative numbers printing in inspect for those bytes
int main() {
  char s[] = "José";
  char s1[] = "🍔";
  char s2[] = "丽莎";
  inspect(s);
  inspect(s1);
  inspect(s2);
}
```

```
[jpolitz@ieng6-201]:01-13-utf8:509$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:510$ ./inspect-starter
(J 74 74) (o 111 111) (s 115 115) (é -61 4294967235) (é -87 4294967
209)
(🍔 -16 4294967280) (🍔 -97 4294967199) (🍔 -90 4294967206) (🍔 -128 42
94967168)
(丽 -28 4294967268) (丽 -72 4294967224) (丽 -67 4294967229) (丽 -24 429
4967272) (莎 -114 4294967182) (莎 -114 4294967182)
[jpolitz@ieng6-201]:01-13-utf8:511$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:512$ ./inspect-starter
(J 74 74 1001010) (o 111 111 1101111) (s 115 115 1110011) (é -61 42
94967235 11111111111111111111111111000011) (é -87 4294967209 111111
11111111111111110101001)
(🍔 -16 4294967280 11111111111111111111111110000) (🍔 -97 42949671
99 11111111111111111111111110011111) (🍔 -90 4294967206 111111111111
11111111111110100110) (🍔 -128 4294967168 11111111111111111111111110
000000)
(丽 -28 4294967268 11111111111111111111111100100) (丽 -72 42949672
24 11111111111111111111111110111000) (丽 -67 4294967229 111111111111
11111111110111101) (丽 -24 4294967272 11111111111111111111111111
01000) (莎 -114 4294967182 11111111111111111111110001110) (莎 -114
 4294967182 11111111111111111111110001110)
[jpolitz@ieng6-201]:01-13-utf8:513$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:514$ ./inspect-starter
(J 74 74 1001010) (o 111 111 1101111) (s 115 115 1110011) (é -61 19
5 11000011) (é -87 169 10101001)
(🍔 -16 240 11110000) (🍔 -97 159 10011111) (🍔 -90 166 10100110) (🍔 -
128 128 10000000)
(丽 -28 228 11100100) (丽 -72 184 10111000) (丽 -67 189 1011111
24 232 11101000) (莎 -114 142 10001110) (莎 -114 142 10001110)
[jpolitz@ieng6-201]:01-13-utf8:515$
```

Observations/Questions from students

- All of the characters with leading 1's in the binary print out as question marks
- The difference between the unsigned and signed representation is always 256
  - Ex. 240 - (-16) = 256
- ASCII characters have the same integer printed for signed and unsigned, but not the question-mark/non-ASCII characters
- The number of 1's in the first byte of a multibyte character corresponds to the number of bytes in the character
  - 110 -> 2-byte character
  - 1110 -> 3-byte character
  - 1111 -> 4-bytes character
- It's interesting that the binary 10000000 is unsigned 128 and signed -128

Negative Numbers

Negative Numbers in Memory

- Q: How are negative numbers represented in memory?
- A: We use 2's complement!
- In 2's complement, the value of the most significant bit is turned negative and all other bits stay positive
  - For example, if we have **1000 0101** and our number is only represented with 8 bits
    - If this was unsigned, we would get 128 + 4 + 1 = **133**
    - If this was signed, we would get -128 + 4 + 1 = -**123**
  - 1111 1111 in 2's complement is equal to -1
- Properties of 2's complement
  - 0 can remain unchanged and unambiguous
  - You can add two binaries together and you will get the value you expect
  - If you're extending the number of bits, leading 1's won't affect your number's value
    - Ex. 1000 0101 is the same as 1111 1111 1000 0101

In C, we can specify if we want our variables to be able to hold **only positive numbers** (unsigned) or **positive and negative numbers** (signed)

`signed int` -> use 2's complement to interpret this

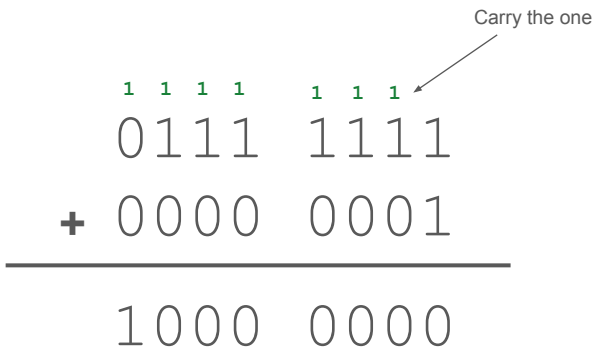
- If we only have 8 bits to represent numbers, a signed int can hold [-128,127] or in binary [10000000 to 01111111]

`unsigned int` -> don't use 2's complement to interpret this

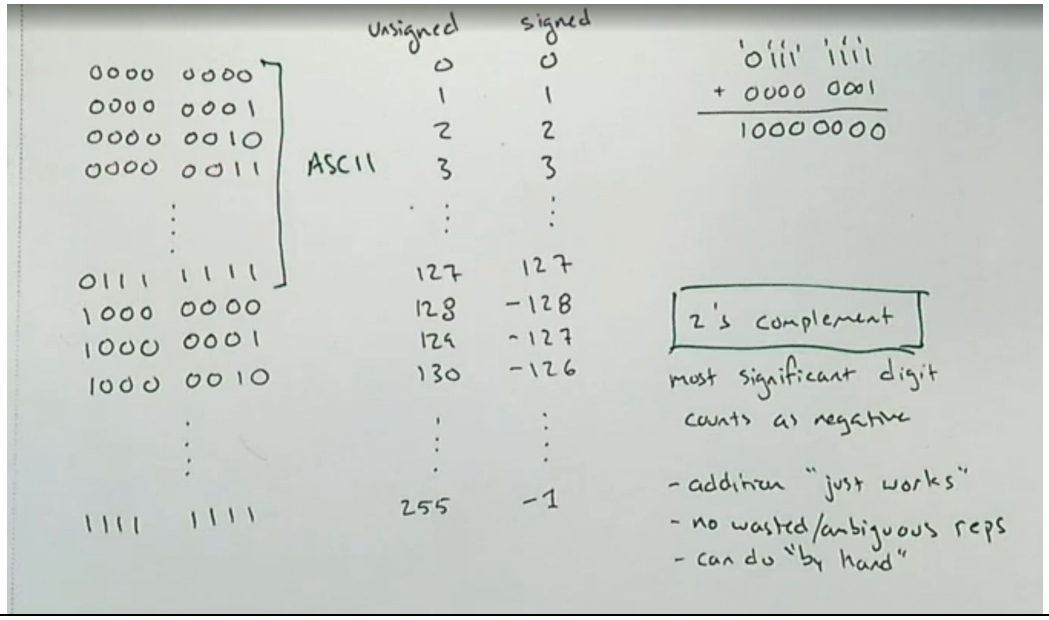
- If we only have 8 bits to represent numbers, an unsigned int can hold [0,255] or in binary [00000000 to 11111111]

Side Note: Binary addition works like normal addition

What is 0111 1111 + 0000 0001?



2’s Complement



Further Learning: In modern computers, nearly everyone uses 2’s complement. However, there is another (less good) way of representing negative numbers using binary. If you’re interested, look up Signed Magnitude (we will not use this in this class)

UTF-8 and Codepoints

What is UTF-8 and why do we have it?

- **UTF-8** (Unicode Transformation Format - 8-bit) is another character encoding standard that gives us **more characters that what's in ASCII**
  - ASCII doesn't give us enough characters to represent all the characters we need (non-Latin alphabets, emojis, etc.)
- It is a **universal standard** used across the internet and globe
- In the modern day, all the text data we use are in this standard
  - 99% of text we interact with use UTF-8 encoding (websites, text, emails, etc.)
  - There are old systems that don't allow you to use special characters because they don't use UTF-8 encoding
- It uses **variable-length encoding** that's **backward-compatible** with ASCII
  - More about this on the next slide

How are characters encoded in UTF-8?

Codepoints!

The binary representation of a character in UTF-8 can be 1-4 bytes long. **But in a string of characters, how do we know if a character is 1, 2, 3, or 4 bytes long?**

Codepoints are made up of **one start byte** and **0-3 continuation bytes**

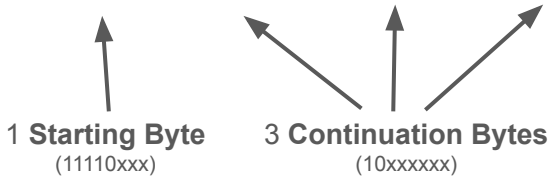
- A **start byte** indicates how many bytes are used to represent the character
  - The prefix will tell you how many bytes are in the character
  - 0xxxxxxx is a 1 byte character (ASCII, backward-compatible)
  - 110xxxxx is a 2 byte character
  - 1110xxxx is a 3 byte character
  - 11110xxx is a 4 byte character
- A **continuation byte** is what comes after start bytes
  - This is always prefixed with “10”
  - 10xxxxxx

UTF-8 Example

Unicode Character: 🦀 ← Joe's favorite emoji

Codepoint (in hex): U+1F980

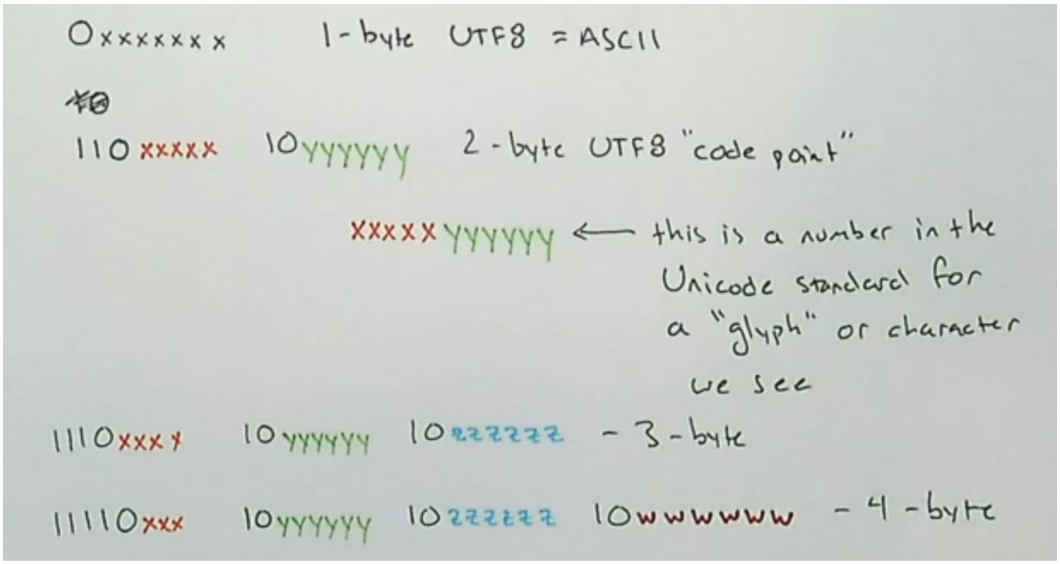
Binary representation: **11110**000 **10**011111 **10**100110 **10**000000



To get the codepoint, we use the non-prefix bits of the binary and transform it into hex!

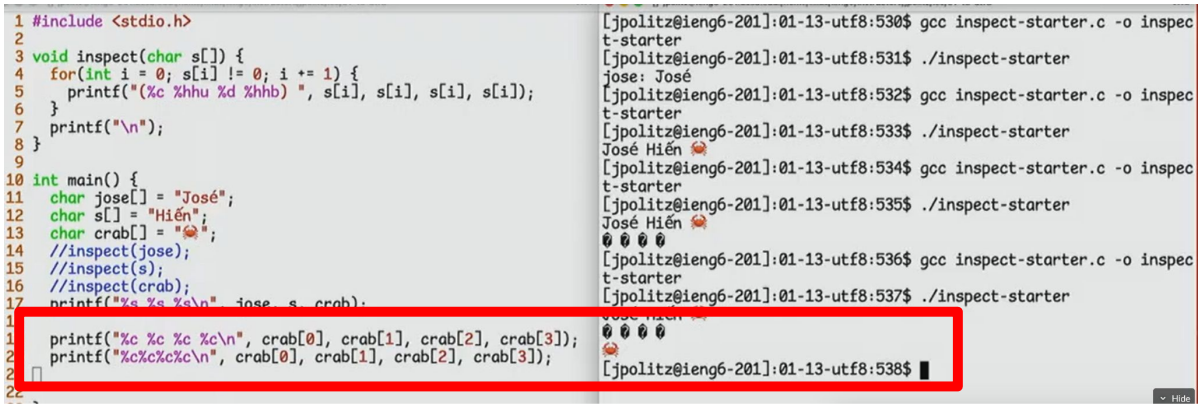
11110**000** 10**011111** 10**100110** 10**000000** -> **00001111110011000000** -> 1F980

UTF-8 Codepoints



Printing in the terminal

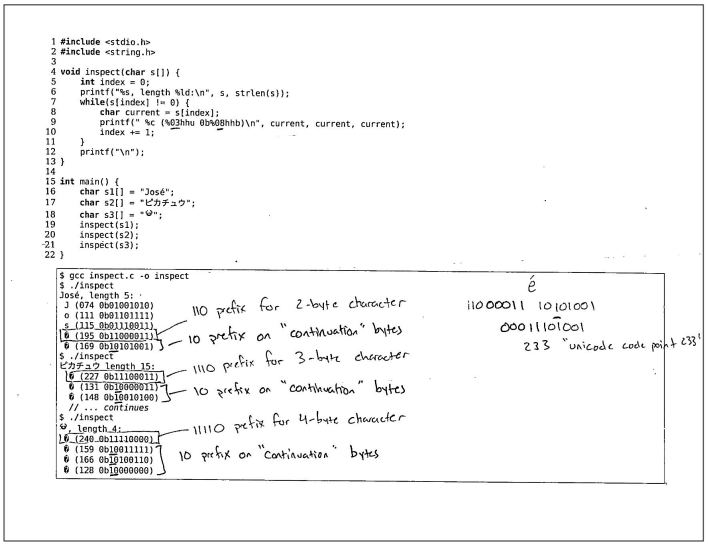
We see if we print 4 characters with no spaces, it prints as the emoji in the terminal! (rather than 4 black question marks). The bits are sent to the terminal and the terminal tries to interpret the characters using UTF-8!



```
#include <stdio.h>

void inspect(char s[]) {
  for(int i = 0; s[i] != 0; i += 1) {
    printf("(%c %hhu %d %hhb) ", s[i], s[i], s[i], s[i]);
  }
  printf("\n");
}

int main() {
  char jose[] = "José";
  char s[] = "Hiến";
  char crab[] = "🦀";
  //inspect(jose);
  //inspect(s);
  //inspect(crab);
  printf("%s %s %s\n", jose, s, crab);

  printf("%c %c %c %c\n", crab[0], crab[1], crab[2], crab[3]);
  printf("%c%c%c%c\n", crab[0], crab[1], crab[2], crab[3]);
```

```
[jpolitz@ieng6-201]:01-13-utf8:530$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:531$ ./inspect-starter
jose: José
[jpolitz@ieng6-201]:01-13-utf8:532$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:533$ ./inspect-starter
José Hiến 🦀
[jpolitz@ieng6-201]:01-13-utf8:534$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:535$ ./inspect-starter
José Hiến 🦀
🦀
[jpolitz@ieng6-201]:01-13-utf8:536$ gcc inspect-starter.c -o inspec
t-starter
[jpolitz@ieng6-201]:01-13-utf8:537$ ./inspect-starter
José Hiến 🦀
🦀
🦀
[jpolitz@ieng6-201]:01-13-utf8:538$
```

Annotated Handouts



```
1 #include <stdio.h>
2 #include <string.h>
3
4 void inspect(char s[]) {
5     int index = 0;
6     printf("%s, length %ld:\n", s, strlen(s));
7     while(s[index] != 0) {
8         char current = s[index];
9         printf(" %c (%03hhu 0b%08hhb)\n", current, current, current);
10        index += 1;
11    }
12    printf("\n");
13 }
14
15 int main() {
16    char s1[] = "José";
17    char s2[] = "ピカチュウ";
18    char s3[] = "🦆";
19    inspect(s1);
20    inspect(s2);
21    inspect(s3);
22 }
```

```
$ gcc inspect.c -o inspect
$ ./inspect
José, length 5:
 J (074 0b01001010)
 o (111 0b01101111)
 s (115 0b01110011)
 é (195 0b11000011)
 ? (169 0b10101001)
$ ./inspect
ピカチュウ length 15:
 ? (227 0b11100011)
 ? (131 0b10000011)
 ? (148 0b10010100)
  // ... continues
$ ./inspect
🦆, length 4:
 ? (240 0b11110000)
 ? (159 0b10011111)
 ? (166 0b10100110)
 ? (128 0b10000000)
```

110 prefix for 2-byte character

10 prefix on "continuation" bytes

1110 prefix for 3-byte character

10 prefix on "continuation" bytes

11110 prefix for 4-byte character

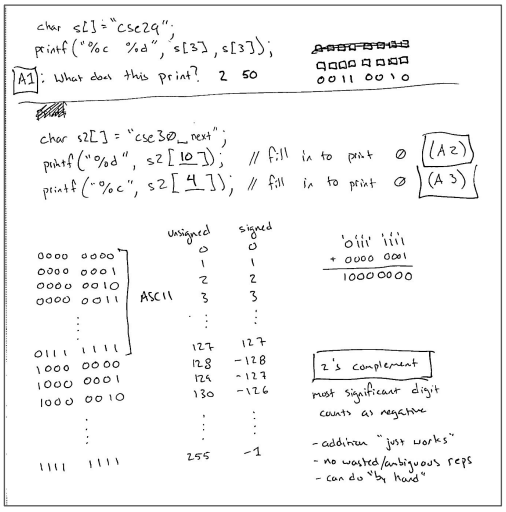
10 prefix on "continuation" bytes

é

11000011 10101001

00011101001

233 "unicode code point e9"



```
char s[] = "csc29";
printf("%c %d", s[3], s[3]);
```

A1: What does this print? 2 50

0011 0010

```
char s2[] = "csc3@_next";
printf("%d", s2[10]);   // fill in to print @ (A2)
printf("%c", s2[4]);    // fill in to print @ (A3)
```

	unsigned	signed
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0000 0011 (ASCII)	3	3
⋮	⋮	⋮
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
⋮	⋮	⋮
1111 1111	255	-1

'0111' '1111'
+ 0000 0001
10000000

2's complement

most significant digit counts as negative

- addition "just works"
- no wasted/ambiguous reps
- can do "by hand"

Annotated Handouts