

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 

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 its decimal value.

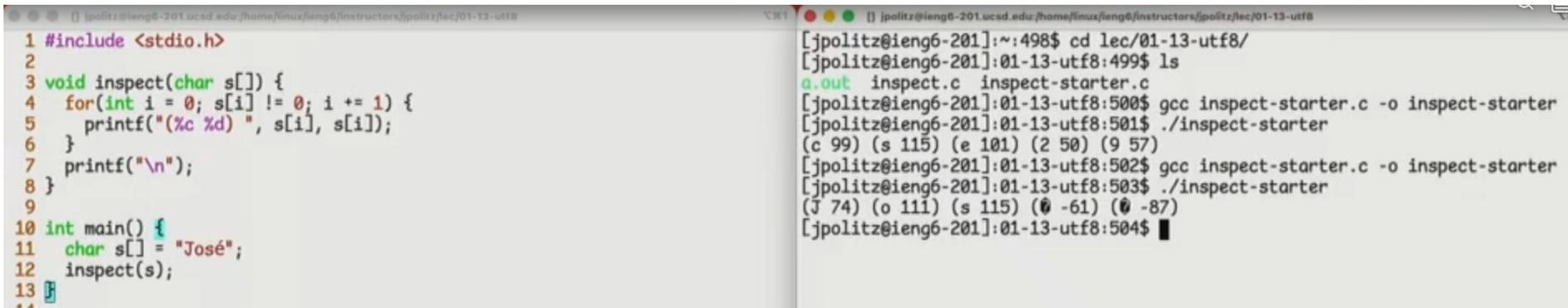
```
1 #include <stdio.h>
2
3 void inspect(char s[]) {
4     for(int i = 0; s[i] != 0; i += 1) {
5         printf("(%c %d) ", s[i], s[i]);
6     }
7     printf("\n");
8 }
9
10 int main() {
11     char s[] = "cse29";
12     inspect(s);
13 }
14
```

```
[jpolitz@ieng6-201.ucsd.edu:~/home/linux/ieng6/instructors/jpolitz/lec/01-13-utf8/
[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



```
1 #include <stdio.h>
2
3 void inspect(char s[]) {
4     for(int i = 0; s[i] != 0; i += 1) {
5         printf("(%c %d) ", s[i], s[i]);
6     }
7     printf("\n");
8 }
9
10 int main() {
11     char s[] = "José";
12     inspect(s);
13 }
14
```

```
[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) (0 -61) (0 -87)
[jpolitz@ieng6-201]:01-13-utf8:504$
```

More format specifiers

- `%b` - binary representation (prints as 4 bytes)
 - 'é' is 1111111111111111111111111000011 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

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 $128 + 4 + 1 = \mathbf{133}$
 - If this was signed, we would get $-128 + 4 + 1 = \mathbf{-123}$
 - 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

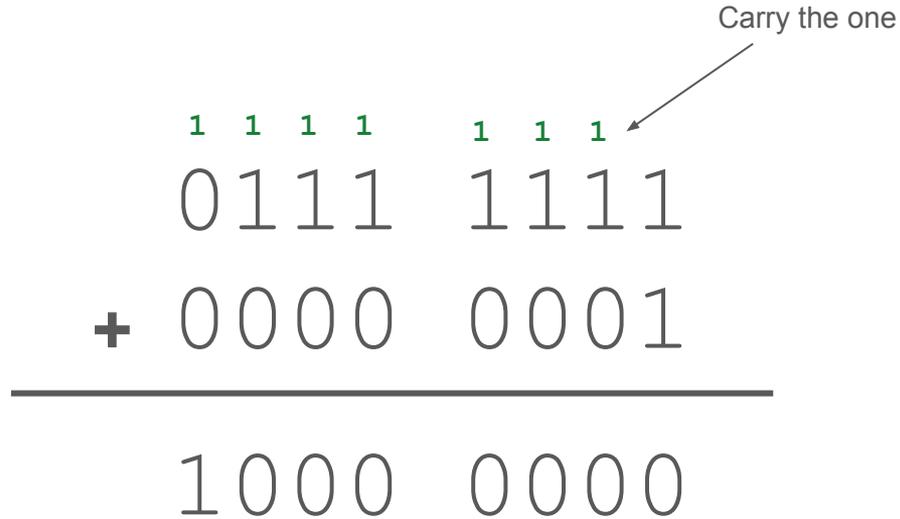
- If we only have 8 bits to represent numbers, an unsigned int can hold [0,255] or in binary [00000000 to 11111111]

Side Note: Binary addition works like normal addition

What is 0111 1111 + 0000 0001?

$$\begin{array}{r} 0111 \\ + 0000 \\ \hline 1000 \end{array}$$

Carry the one



The diagram illustrates the binary addition of 0111 1111 and 0000 0001. The first number is 0111 1111, with green '1's above the last seven bits. The second number is 0000 0001. A horizontal line is drawn below the numbers. The result is 1000 0000. An arrow points from the text 'Carry the one' to the rightmost '1' of the second number, indicating the carry that propagates through all seven bits of the first number.

2's Complement

	Unsigned	Signed
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0000 0011	3	3
⋮	⋮	⋮
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
⋮	⋮	⋮
1111 1111	255	-1

ASCII

$$\begin{array}{r}
 0'111 \quad 1'111 \\
 + 0000 \quad 0001 \\
 \hline
 10000000
 \end{array}$$

2's complement

most significant digit
counts as negative

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

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 than 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**

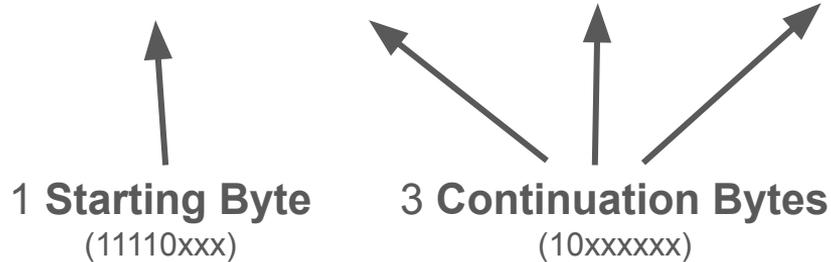
- A **start byte** indicates how many bytes are used to represent the character
 - The prefix will tell you how many bytes are in the character
 - 0xxxxxxx is a 1 byte character (ASCII, backward-compatible)
 - 110xxxxx is a 2 byte character
 - 1110xxxx is a 3 byte character
 - 11110xxx is a 4 byte character
- 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: **11110000 10011111 10100110 10000000**



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** -> **000011111100110000000** -> 1F980

UTF-8 Codepoints

0xxxxxxx 1-byte UTF8 = ASCII

~~10~~

110xxxxx 10yyyyyy 2-byte UTF8 "code point"

xxxxx yyyyyy ← this is a number in the
Unicode standard for
a "glyph" or character
we see

1110xxxx 10yyyyyy 10zzzzzz - 3-byte

11110xxx 10yyyyyy 10zzzzzz 10wwwww - 4-byte

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!

```
1 #include <stdio.h>
2
3 void inspect(char s[]) {
4     for(int i = 0; s[i] != 0; i += 1) {
5         printf("%c %hhu %d %hbb) ", s[i], s[i], s[i], s[i]);
6     }
7     printf("\n");
8 }
9
10 int main() {
11     char jose[] = "José";
12     char s[] = "Hiên";
13     char crab[] = "🦀";
14     //inspect(jose);
15     //inspect(s);
16     //inspect(crab);
17     printf("%s %s %s\n", jose, s, crab);
18
19     printf("%c %c %c %c\n", crab[0], crab[1], crab[2], crab[3]);
20     printf("%c%c%c%c\n", crab[0], crab[1], crab[2], crab[3]);
21 }
22
```

```
[jpolitz@ieng6-201]:01-13-utf8:530$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:531$ ./inspect-starter
jose: José
[jpolitz@ieng6-201]:01-13-utf8:532$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:533$ ./inspect-starter
José Hiên 🦀
[jpolitz@ieng6-201]:01-13-utf8:534$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:535$ ./inspect-starter
José Hiên 🦀
🦀🦀🦀🦀
[jpolitz@ieng6-201]:01-13-utf8:536$ gcc inspect-starter.c -o inspect-starter
[jpolitz@ieng6-201]:01-13-utf8:537$ ./inspect-starter
José Hiên 🦀
🦀🦀🦀🦀
[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[] = "é";
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)
0 (169 0b10101001)
ピカチュウ length 15:
ピ (227 0b11100011)
カ (131 0b10000011)
チ (148 0b10010100)
// ... continues
$ ./inspect
length 4:
0 (240 0b11110000)
0 (159 0b10011111)
0 (166 0b10100110)
0 (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 233"

```
char s1[] = "csc2q";
printf("%c %d", s[3], s[3]);
```

A1: What does this print? 2 50

```

0000 0000
0000 0000
0011 0010

```

```

char s2[] = "cse30_next";
printf("%d", s2[10]); // fill in to print 0 (A2)
printf("%c", s2[4]); // fill in to print 0 (A3)

```

	unsigned	signed
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0000 0011	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

1000 0000

2's complement

most significant digit counts as negative

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

Annotated Handouts

UTF8

~~0xxxxxx~~

0xxxxxx 1-byte UTF8 = ASCII

~~110xxxx~~

110xxxx 10yyyyyy 2-byte UTF8 "code point"

xxxxxyyyyyy ← this is a number in the Unicode standard for a "glyph" or character we see

1110xxx 10yyyyyy 10zzzzzz - 3-byte

11110xxx 10yyyyyy 10zzzzzz 10wwwwww - 4-byte

```
char s[] = "csc29";
```

```
printf("%c %d", s[2], s[3]);
```

```
□□□□ □□□□
0011 0010
```

A1 What does this print? 2 50

```
char s2[] = "csc30 next";
```

```
printf("%d", s2[10]);
```

```
printf("%c", s2[4]);
```

// fill in to print 0

// fill in to print ~~0~~ 0

A2

A3

	unsigned	signed	
0000 0000	0	0	$ \begin{array}{r} 0111\ 1111 \\ + \quad 1 \\ \hline 1000\ 0000 \end{array} $
0000 0001	1	1	
0000 0010	2	2	
0000 0011	3	3	
0000 0100	4	4	
⋮			
0111 1111	127	127	2's complement
1000 0000	128	-128	Idea: most significant bit is interpreted as <u>negative</u>
1000 0001	129	-127	- addition "just works"
1000 0010	130	-126	- unique/unambiguous 0
⋮			- adding leading 1's to a negative # does not change its meaning
1111 1111	255	-1	

UTF-8

1-byte 0xxxxxx ASCII

2-byte 110xxxx 10yyyyyy

3-byte 1110xxx 10yyyyyy 10zzzzzz

4-byte 11110xxx 10yyyyyy 10zzzzzz 10wwwwww

xxxxxyyyyyy → this is binary # of a code point in Unicode