

CSE 29

Lecture 4 Summary

January 15, 2026



Logistical Q&A

- Have we learned everything we need to know to complete PSET 1 and PA 1?
 - Not yet. The first assignment is weird because we haven't covered enough fundamentals yet. We are learning as we go. The next assignments will be different, because we will already learn everything we need to know for the assignment before it is released.



Review Questions

1. What is the signed + unsigned interpretation of 10000001?
2. How many code points are encoded by the UTF-8 sequence?

11010101 10111111 11101011 10110110 10000001

3. `char s[] = { 0b01100001, 98, 'c', '\0' };
printf("%s", s);` //What does this print?

Try these questions out yourself before looking at the answers!

Answers in speaker notes of this slide 🙋

The annotated handout also has the solutions as gone over in lecture

1. Signed: $-128 + 1 = -127$; Unsigned: $128 + 1 = 129$
2. 2 code points. Bytes 1 and 3 do not start with "10" so they are the start of a UTF-8 character. All other bytes are continuations of the same character/code point.
3. It prints abc. $0b01100001 = 97 = 'a'$, $98 = 'b'$

Questions

Q: Would C put a null terminator `'\0'` in `s` in Review Question 3 if we did not explicitly do so?

A: No. C only implicitly adds a null terminator in string literals.

Ex:

```
char s1[] = "abc"; // will have null terminator
```

```
char s2[] = {'a', 'b', 'c'}; // will not have null  
terminator
```

Bitwise Operators

Why bitwise?

It can make certain tasks easier, where using algebra would be difficult

Useful particularly for tasks like assignment 1 where by definition code points use specific bit patterns.

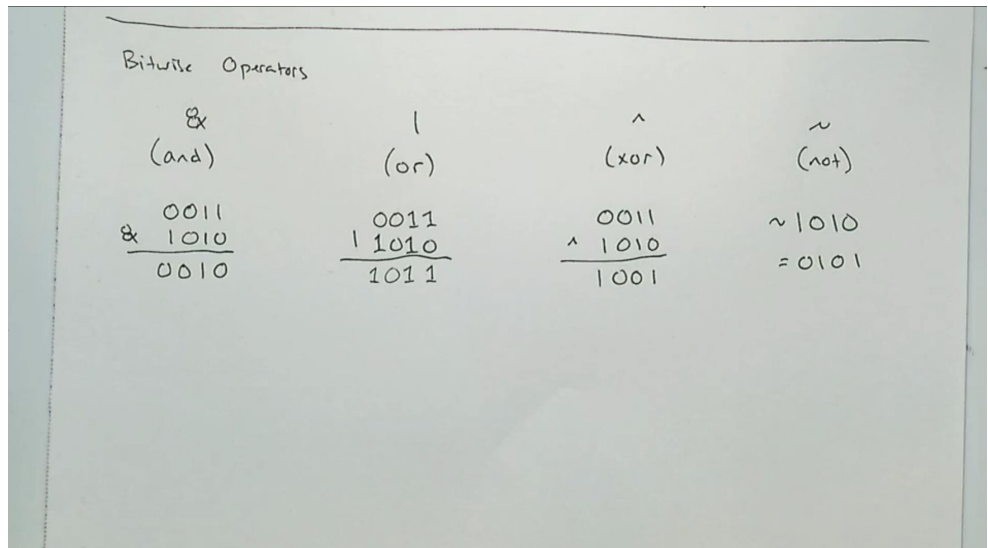
Allows us to check the values of specific bits

Allows us to set bits

Examples and definitions given on 4-bit values, but they extend to any number of bits

Bitwise Operators

- & (and)
- | (or)
- ^ (xor)
- ~ (not)



Bitwise Operator Truth Table

Bitwise Operator Truth Table

X	Y	X&Y	X Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bitwise Operators Example

```
char c1 = 0b01100001;
```

```
char c2 = 0b11001110;
```

```
char c3 = 0b11100001;
```

```
char c4 = 0b11110111;
```

```
char mask = 0b11110000;
```

Try to fill in the blanks, answers in speaker notes!

```
c1 & mask = _____;
```

```
c2 & mask = _____;
```

```
c3 & mask = _____;
```

```
c4 & mask = _____;
```

```
c1 & mask = 0b0110 0000;
```

```
c2 & mask = 0b1100 0000;
```

```
c3 & mask = 0b1110 0000;
```

```
c4 & mask = 0b1111 0000;
```

Implementing the `is_3byte_utf8` function

Function description: Given a byte, return 1 if it is a starting byte for 3-byte UTF-8 characters. Otherwise, return 0.

Function Header:

```
int is_3byte_utf8(char c) {}
```

Try it out using bitwise operators and masks!
Solution on the next slide

Implementing the `is_3byte_utf8` function

Solution: Use a mask to only get the 4 left bits of the char and see if it matches up with the prefix for the starting byte of a 3-byte utf-8 character (1110)

```
int8_t is_3byte_utf8(char c) {  
    return (c & mask) == 0b11100000;  
}
```



`mask` was previously defined as `0b11110000`

Bitwise Operator Q&A

- Why does this function have an `int8_t` (or `int`) return type?
 - There is not a boolean data type in C. In 0, it is interpreted as false. If it's non-zero, it is interpreted as true. Commonly, 1 is true. `int8_t` is the smallest available size since it is 8 bits, but it will function the same for our purposes as an `int`(4 bytes) or `int8_t` (1 byte)
- What does `(c & mask) == 0b11100000` evaluate to?
 - It's a number (0 or 1) that can be interpreted as false or true (0 and anything not 0, respectively)
- What does the `==` evaluate to?
 - False evaluates to 0, and true evaluates to not 0, commonly 1
- Could we use division or less than/greater than a value to do this?
 - Yes, but using bitwise operators matches the specs and is more computationally efficient
- Could less than 8 bits be used to store a boolean (or anything else)?
 - There is no data type in C that does this, but you could create your own encoding/decoding scheme.

Bitwise Q&A continued

- Does the mask look the same for different functions
 - This mask is useful for the 3-byte UTF-8 character case because the prefix of the start byte is 4 bits (1110). Masks can look different based on what bits you need to mask out
- Is there a reason C really likes 32 bit things?
 - Historically, the most efficient layout of hardware called SRAM cells was 32 side by side. This got us the 32 bit int which holds about 4 billion different values which was very good. Then 32 was programmed in many places so it's still prevalent. It was good engineering awhile ago and now it's historical.

Shifting Operators (covered in 12:30 lecture)

- << (left shift)
 - Move all the bits over to the left by a certain number, then pad with 0's on the empty right slots
- >> (right shift)
 - Move all the bits over to the right by a certain number
 - Padding the empty slots on the left side depends on the data type (see next slide)

0b00110001 << 2 = 0b11000100

These red bits are "lost"

0b00110001 << 3 = 0b10001000

Blue bits is the empty slots
we filled in

0b00110001 >> 2 = 0b00001100

Right Shifts are special

We are using an **Arithmetic Right Shift**, where the left is padded with the most significant (leftmost) bit

In **C**, right shift behavior depends on the **type of value** being shifted.

If the value is **signed** (data type is `int` or `int32_t`)

`0b10001100 >> 2 → 0b11100011`

Here, the leftmost bit is a 1, so the left will be padded with 1s

If the value is **unsigned** (data type is `unsigned int` or `uint32_t`)

`0b10001100 >> 2 → 0b00100011`

Here, the leftmost bit is a 0, and the left is padded with 0s

Shifting Operator Q&A

- How do you know when to use a data type that specifies the number of bits
 - Different compilers interpret `ints` as different amounts of bits, which could mess a up a program if you code expecting 32 bits and compile it with a compiler that has a default `int` size of 16 bits. An `int32_t` will always be interpreted as 32 bits. It's good practice to be specific in the number of bits you are expecting.
- Is the default `int` on most systems signed 32-bit?
 - Yes

Hexadecimal

Hexadecimal

- Hexadecimal came about because programmers realized it was much easier to convert binary (base 2) to base 16 than to decimal (base 10), especially as numbers get increasingly bigger
- It is *significantly* easier to write a binary sequence in hexadecimal than binary (and you're much less likely to make a mistake)
- You can write `0x<hex_num>` in C, and it will have the same value as the decimal and binary equivalent, just like how you can write `0b<bin_num>`

Bin	Dec	Hex	Bin	Dec	Hex
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

16's 1's

0xA7

$$A * 16 + 7 * 1$$

$$160 + 7$$

$$167$$

0xA7

/ \

0b1010 0111

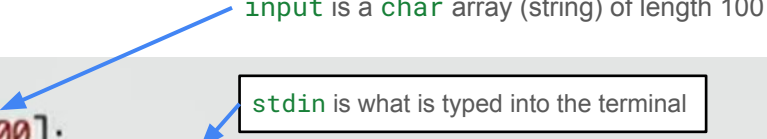
$$\begin{aligned}
 &A * 16^1 + 7 * 16^0 \\
 &= 10 * 16^1 + 7 * 16^0 \\
 &= 167
 \end{aligned}$$

Implementing Functions from PSET1

Creating `main` function

This is similar to the main functions in your starter code for PSETs!

```
int main() {  
    char input[100];  
    fgets(input, 100, stdin);  
    printf("The input was: %s\n", input);  
    printf("The strlen of input is: %ld\n", strlen(input));  
}
```



input is a `char` array (string) of length 100

stdin is what is typed into the terminal

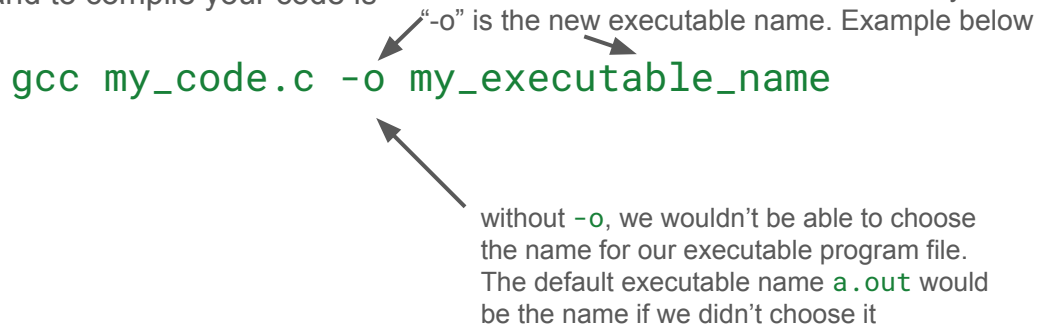
`fgets` stores what's typed in `stdin` into `input`

Side Note: How do I Compile my Code?

The command to compile your code is

What comes after “gcc” can occur in any order
EXCEPT that whatever comes immediately after
“-o” is the new executable name. Example below

`gcc my_code.c -o my_executable_name`



without `-o`, we wouldn't be able to choose
the name for our executable program file.
The default executable name `a.out` would
be the name if we didn't choose it

For this PA, we are going to compile using `gcc utf8analyzer.c -o utf8analyzer` and run the executable by using `./utf8analyzer`

```
gcc my_code.c -o my_executable_name
gcc -o my_executable_name my_code.c
```

But the following are NOT the same

```
gcc my_executable_name -o my_code.c
gcc -o my_code.c my_executable_name
gcc -o my_code.c my_executable_name
gcc my_executable my_code.c -o
```

Exercise: Implementing Functions from PSET

Try to implement the three functions below before looking at the answers in the following slides! These are also functions that are part of PSET 1.

- `int8_t is_ascii (char s[])`
- `int8_t width_from_start_byte (char c)`
- `int32_t utf8_len (char s[])`

Function descriptions are in the following slides!

Implementing `is_ascii` function

Function Description: Given a string (`char s[]`), determine if all of the characters are ASCII characters. If yes, return `1`, if not return `0`.

Function Header: `int8_t is_ascii (char s[])`

Implementing `is_ascii` function

Function Description: Given a string (`char s[]`), determine if all of the characters are ASCII characters. If yes, return `1`, if not return `0`.

```
int8_t is_ascii(char s[]) {  
    for(int i = 0; s[i] != '\0'; i += 1) {  
        if((s[i] & 0b10000000) == 0b10000000) {  
            return 0; // This 0 means "false"  
        }  
    }  
    return 1;  
}
```

Implementing `width_from_start_byte`

Function Description: Given a `char c` (a byte) that represents a start byte, determine how many bytes the character has and return that number. If it is not a valid start byte, return `-1`.

Function Header: `int8_t width_from_start_byte (char c)`

Implementing `width_from_start_byte`

Function Description: Given a `char c` (a byte) that represents a start byte, determine how many bytes the character has and return that number. If it is not a valid start byte, return `-1`.

```
int8_t width_from_start_byte(char c) {  
    if((c & 0b10000000) == 0b00000000) { return 1; } // ASCII = 1 byte  
    if((c & 0b11100000) == 0b11000000) { return 2; } // 2 byte case  
    if((c & 0b11110000) == 0b11110000) { return 3; } // 3 byte case  
    if((c & 0b11111000) == 0b11111000) { return 4; } // 3 byte case  
    // if((c & 0xF8) == 0xF0)    thanks robot 🤖  
    return -1;  
}
```

Implementing `utf8_len`

Function Description: Given a string `char s[]` with UTF-8 characters, count the number of code points in the string.

Function Header: `int32_t utf8_len (char s[])`

Hint: Think about how you can use `width_from_start_byte` here

Implementing `utf8_len`

Function Description: Given a string `char s[]` with UTF-8 characters, count the number of code points in the string.

Hint: Think about how you can use `width_from_start_byte` here

```
// Count the number of _code points_ in the string
int32_t utf8_len(char s[]) {
    int byte_index = 0, cp_count = 0;
    while(s[byte_index] != '\0') {
        int width = width_from_start_byte(s[byte_index]);
        cp_count += 1;
        byte_index += width;
    }
    return cp_count;
}
```

Result from testing all 3 functions

```
34
35 int main() {
36     char input[100];
37     fgets(input, 100, stdin);
38     printf("The input was: %s\n", input);
39     printf("The strlen of input is: %ld\n", strlen(input));
40     printf("The utf8len of input is: %d\n", utf8_len(input));
41     printf("Is ASCII? %d\n", is_ascii(input));
42     printf("Width based on byte 0: %d\n", width_from_start_byte(input[0]));
43 }
```

strlen is wrong
because it counts the
bytes up to the null
terminator, but
non-ASCII characters
have more than 1 byte

```
$ ./utf8analyzér
🍷 ate 🍌
The input was: 🍷 ate 🍌
The strlen of input is: 14
The utf8len of input is: 8
Is ASCII? 0
Width based on byte 0: 4
$ █
```

utf8len is one off because when we
type in the terminal, we press Enter,
which is a newline character (\n).
This is NOT counting the null
terminator.

“🍷 ate 🍌\n” ← invisible
newline character

Joe's Notes (11am)

Review Qs:

1. What is the signed + unsigned interpretation of 1000001?
 (decimal)
 -128 + 0 + 0 + 0 + 0 + 0 + 1 = -127 128 + 0 + 0 + 0 + 0 + 0 + 1 = 129

2. How many code points are encoded by this UTF-8 sequence?

11010101 10111111 11101011 1010110 10000001
 2 code points

3. char s[] = {0b01100001, 98, 'c', '\0'};
 printf("%s", s); // What does this print? abc

José are Yummy ^{at} beach
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
 byte_index

Bitwise Operators

Hexadecimal

Bitwise Operators

&
(and)

|
(or)

^
(xor)

~
(not)

0011
1010
0010

0011
1010
1011

0011
1010
1001

~1010
= 0101

char c1 = 0b0110 0001;
 char c2 = 0b1100 1100;
 char c3 = 0b1110 1001;
 char c4 = 0b1110 1101;

c1 & mask_3 = 0110 0000
 c2 & mask_3 = 1100 0000
 c3 & mask_3 = 1110 0000
 c4 & mask_3 = 1111 0000

char mask_3 = 0b1111 0000;

is 3 byte utf8(char c) ?
 return (c & mask_3) == 0b11100000

0000 0
 0001 1
 0010 2
 0011 3
 0100 4
 0101 5
 0110 6
 0111 7
 1000 8
 1001 9
 1010 10
 1011 11
 1100 12
 1101 13
 1110 14
 1111 15

16's
 0xA7
 A*16 + 7*1
 160 + 7
 167
 0xA7
 0b1010 0111