Lecture 4: Unicode & bitwise operations

CSE 29: Systems Programming and Software Tools

Olivia Weng

Announcements

- Problem set 1 released

- TA office hours today during discussion

What about non-English characters?

What about non-English characters?

- Thousands more characters used in languages around the world
- ASCII does not define:
  - Spanish: é
  - Chinese: 中
  - Emoji: 🐫
- char datatype of 1 byte only encodes 256 possible bit patterns
- Challenge: Millions of lines of code written that assumed 1 byte ASCII chars

UTF-8: Unicode encoding

- Use more bits to encode more characters!
- Code point: an integer representing a character (e.g., 'A' == 65)

- Normal ASCII code point: Highest order bit of byte is 0xxxxxxx
  - UTF-8 is backwards compatible with ASCII!
- Multi-byte code point: Highest order bit of byte is 1xxxxxxx

How many bytes do you need?

- Multi-byte code point: Highest order bit of byte is 1xxxxxxx
- Bit flags indicate code point length
  - 110xxxxx = 2 bytes
  - 1110xxxx = 3 bytes → 1110xxxx 10xxxxxx 10xxxxxx
  - 11110xxx = 4 bytes

    1110xxxx: 1st byte; 10xxxxxx: 2nd byte; 10xxxxxx: 3rd byte
- Bytes after the first byte start with
  - 10xxxxxx

2 bytes: 110xxxxx 10xxxxxx

110xxxxx: flag + code point part = 1st byte

10xxxxxx: flag + code point part = 2nd byte

Code point construction

0xC3 → 1100 0011

0xA9 → 1010 1001

- é = 11000011 10101001
  - bit flags : only encodes metadata (i.e., only provides information about)
  - the code point = 000011 101001 = 233

128+64+32+8+1

11101001 = 233

$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$

192
+ 32
224
+ 8
232

Demo

- printing codepoints

How many bytes do you need?

1 byte char 0xxxxxxx

- Multi-byte code point: Highest order bit of byte is 1xxxxxxx
- Bit flags indicate code point length
  - 110xxxxx = 2 bytes
  - 1110xxxx = 3 bytes
  - 11110xxx = 4 bytes
- Bytes after the first byte start with
  - 10xxxxxx

- How do we check for these specific bit flags to know how many bytes we have?
  - Need a way of inspecting individual bits!

Bit operations

- Special mathematical operations in C for manipulating bits
  - &: AND
  - |: OR
  - ~: NOT
  - ^: XOR
  - >>: shift right
  - <<: shift left

AND: &

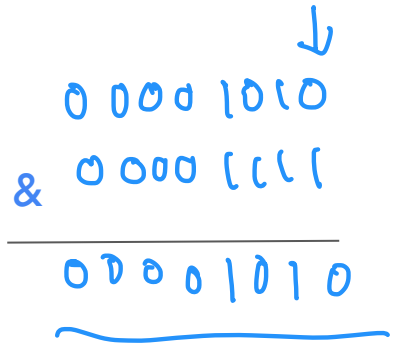
T & T = T

0 = "false"
1 = "true"

- AND each bit together

Truth table

input_a	input_b	Result
0	0	0
0	1	0
1	0	0
1	1	1



What is the result?

```
  00110011
& 10100101
----------
  00100001
```

OR: |

T | F = T
F | T = T
T | T = 1

0 = "false"
1 = "true"

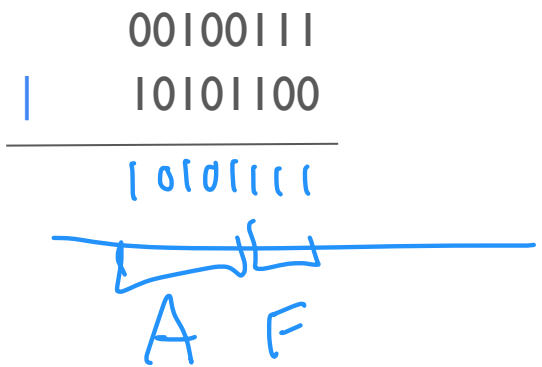
- OR each bit together

Truth table

input_a	input_b	Result
0	0	0
0	1	1
1	0	1
1	1	1

```
  00010001
| 11001010
----------
  11011011
```

What is the result?



NOT: ~ operates on 1 operand

- NOT each bit together

Truth table

input_a	Result
0	1
1	0

~ 1010 1100

0101 0011

Shift right: >>

Ex: 4 bits two's complement

- Shift the bits "off a cliff" to the right

0100 >> 2 = 0001 = 0001 ← sign extending by 0
4 /. 2 /. 2 1

0100 >> 1 = 0010 = 0010
4 /. 2 2

1000 >> 1 = 1200 = 1100 ← sign extending by 1
-8 /. 2 -4

4 bits 2's complement

What is the result in binary and decimal?

- `1010 >> 3` = 1111 -1
- `1000 >> 2` = 1110 -2
- `1011 >> 4` = 1111 -1
- `1101 >> 2` = 1111 -1

$-5 // 2 = -2 // 2 = -1 // 2 = 0 \neq -1$ division pattern breaks

Shift left: <<

4 bits unsigned

- Shift the bits "off a cliff" to the left

0001 << 1 = 0010 = 0010

1 * 2 2

0001 << 3 = 1000 = 1000

1 * 2 * 2 * 2 = 8

What is the result in binary and decimal (unsigned)?

- `0010 << 2` = 1000 8
- `0001 << 3` = 1000 8
- `1000 << 4` = 0000 0
- `0110 << 2` = 1000 8

Demo: practice with bitwise operators

```
int bitwise_is_even(int8_t num);
```

```
int count_1_bits(int32_t num);
```

How to use bit operators to select specific bits?

How to use bit operators to select specific bits?

- Bit masking: select specific individual bits out of a binary representation of a number
- Examples:
  - lowest_four_bits(0b01010101) = 00000101
    & 00001111
    00000101
  - highest_four_bits(0b10110011) = 10110000
    & 11110000
    10110000
  - lowest_four_bits(192) = 0
    128+64 = 192
    11000000
    & 00001111
    00000000
  - highest_four_bits(lowest_four_bits(200)) = 0
    128+64+8 = 200 → 11001000
    & 00001111
    00001000
    => & 11110000
    00000000

How to implement masking?

```
char lowest_four_bits(char c) {
    return c & 0b00001111;
}

char highest_four_bits(char c) {
    return c & 0b11110000;
}
```

Why can't we use the bitwise | operator?

The bitwise & operator selects specific bit positions based on the pattern of 1's that the variable is &'d with.

& truth table	Result
0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

How do we check the codepoint bit flags?

- Bit masking!
- `codepoint_size(char string[])`