# CMPUT201W20B2 Week 11

Abram Hindle

March 26, 2020

# Contents

# 1 Week11

## 1.1 Copyright Statement

### 1.1.1 License

### 1.1.2 Hazel Code is licensed under AGPL3.0+

## 1.2 Alternative version

Checkout the .txt, the .pdf, and the .html version

## 1.3 Init ORG-MODE

```
;; I need this for org-mode to work well
;; If we have a new org-mode use ob-shell
;; otherwise use ob-sh --- but not both!
```

```
(if (require 'ob-shell nil 'noerror)
  (progn
    (org-babel-do-load-languages 'org-babel-load-languages '((shell . t))))
  (progn
    (require 'ob-sh)
    (org-babel-do-load-languages 'org-babel-load-languages '((sh . t)))))
(org-babel-do-load-languages 'org-babel-load-languages '((C . t)))
(org-babel-do-load-languages 'org-babel-load-languages '((python . t)))
(setq org-src-fontify-natively t)
(setq org-confirm-babel-evaluate nil) ;; danger!
(custom-set-faces
 '(org-block ((t (:inherit shadow :foreground "black"))))
 '(org-code ((t (:inherit shadow :foreground "black")))))
```

### 1.3.1 Org export

```
(org-html-export-to-html)
(org-latex-export-to-pdf)
(org-ascii-export-to-ascii)
```

### 1.3.2 Org Template

Copy and paste this to demo C

```
#include <stdio.h>

int main(int argc, char**argv) {
    return 0;
}
```

## 1.4 Remember how to compile?

gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o programname programname.c

## 1.5 Numbers!

Computers love powers of 2 because we calculate everything via bits.

Bases used on computers:

2, 8, 10, 16

- 32 in base 16 is 2*pow(16,1) = 0x20

- 32 in base 10 is 3*pow(10,1) + 2*pow(10,0) = 3*10 + 2*1 = 32

- 32 in base 8 is 4*pow(8,1) = 4*8 = 32 = 040

- 32 in base 2 is 1*pow(2,5) = 0b100000

- 31 in base 16 is 1*pow(16,1) + 15*pow(16,0)

- 31 in base 10 is 3*pow(10,1) + 1*pow(10,0)

- 31 in base 8 is 3*pow(8,1) + 7*pow(8,0)

- 31 in base 2 is 1*pow(2,4) + 1*pow(2,3) + 1*pow(2,2) + 1*pow(2,1) + 1*pow(2,0)

- Notation for base 2 for 31: 0b11111 # not available in C, good in python

- Notation for base 8 for 31: 037 # available in C , good in python

- Notation for base 10 for 31: 31 # available in C , good in python

- Notation for base 16 for 31: 0x1F # available in C , good in python

### 1.5.1   Binary

```
Base 2: powers of 2

Digits: 0,1

0:   0b00000  8:   0b01000
1:   0b00001  9:   0b01001
2:   0b00010 10:   0b01010
3:   0b00011 11:   0b01011
4:   0b00100 12:   0b01100
5:   0b00101 13:   0b01101
6:   0b00110 14:   0b01110
7:   0b00111 15:   0b01111
             16:   0b10000
```

### 1.5.2 Octal

```
#include <stdio.h>

int main() {
    printf("%d\n", 037);
}
```

```
31
```

Base 8: powers of 8

3 bits

Digits: 0,1,2,3,4,5,6,7

```
0:  000  8: 010
1:  001  9: 011
2:  002 10: 012
3:  003 11: 013
4:  004 12: 014
5:  005 13: 015
6:  006 14: 016
7:  007 15: 017
         16: 020
```

```
07:                                  7 =    7
077:                        7*8 + 7 =   63
0777:            7*8*8 + 7*8 + 7 =  511
07777: 7*8*8*8 + 7*8*8 + 7*8 + 7 = 4095
```

### 1.5.3 Base10 Review

Base 10: power of 10

   Digits: 0,1,2,3,4,5,6,7,8,9

~4 bits - not a power of 2

```
0:  0  8:  8
1:  1  9:  9
```

```
2:   2 10: 10
3:   3 11: 11
4:   4 12: 12
5:   5 13: 13
6:   6 14: 14
7:   7 15: 15
       16: 16
```

### 1.5.4   Hex Review

```
Base 16: power of 16

   Digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
or Digits: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f

0:  0x00  8: 0x08
1:  0x01  9: 0x09
2:  0x02 10: 0x0A
3:  0x03 11: 0x0B
4:  0x04 12: 0x0C
5:  0x05 13: 0x0D
6:  0x06 14: 0x0E
7:  0x07 15: 0x0F
         16: 0x10
```

$$0xF \quad = \quad 15 = \quad 15 = 2^4 - 1$$
$$0xFF \quad = \quad 15*16 + 15 = \quad 255 = 2^8 - 1$$
$$0xFFF \quad = \quad 15*16*16 + 15*16 + 15 = \quad 4095 = 2^{12} - 1$$
$$0xFFFF = 15*16*16*16 + 15*16*16 + 15*16 + 15 = 65535 = 2^{16} - 1$$

```
Digit Lookup

0:  0x0 0b0000     8: 0x8 0b1000
1:  0x1 0b0001     9: 0x9 0b1001
2:  0x2 0b0010    10: 0xA 0b1010
3:  0x3 0b0011    11: 0xB 0b1011
4:  0x4 0b0100    12: 0xC 0b1100
5:  0x5 0b0101    13: 0xD 0b1101
6:  0x6 0b0110    14: 0xE 0b1110
7:  0x7 0b0111    15: 0xF 0b1111
```

```
1 nibble = 4 bits = 1 hex digit
```

### 1.5.5   Bitwise Operators

Bitwise operators operate at the bit-level for variables such as integers.

Typically try to avoid anything signed or floating point when doing bit-wise operations unless you are sure.

Also be aware that structs have padding so you might get unexpected results.

### 1.5.6   Left Shift

0b000001 « 1 = 0b000010 0b000001 « 2 = 0b000100 0b000001 « 3 = 0b001000
0b101011 « 3 = 0b011000

Integer Multiplication by 2. Shifts bits to the left. Do not use negative values.

WARNING: Shifting on signed integers is not safe or portable.

```c
#include <stdio.h>
#include <inttypes.h>
int main() {
    uint64_t u64 = 1;
    uint32_t u32 = 1;
    uint16_t u16 = 1;
    int64_t  i64 = 1;
    int32_t  i32 = 1;
    int16_t  i16 = 1;
    for (int i = 0 ; i < 64; i++) {
        u64 = u64 << 1;
        i64 = i64 << 1;
        u32 = u32 << 1;
        i32 = i32 << 1;
        u16 = u16 << 1;
        i16 = i16 << 1;
        printf("64: %20llu %20lld\n", u64, i64);
        printf("32: %20lu %20ld\n",   u32, i32);
        printf("16: %20hu %20hd\n",   u16, i16);
    }
}
```

```
64:                    2                    2
```

```
32:                 2                 2
16:                 2                 2
64:                 4                 4
32:                 4                 4
16:                 4                 4
64:                 8                 8
32:                 8                 8
16:                 8                 8
64:                16                16
32:                16                16
16:                16                16
64:                32                32
32:                32                32
16:                32                32
64:                64                64
32:                64                64
16:                64                64
64:               128               128
32:               128               128
16:               128               128
64:               256               256
32:               256               256
16:               256               256
64:               512               512
32:               512               512
16:               512               512
64:              1024              1024
32:              1024              1024
16:              1024              1024
64:              2048              2048
32:              2048              2048
16:              2048              2048
64:              4096              4096
32:              4096              4096
16:              4096              4096
64:              8192              8192
32:              8192              8192
16:              8192              8192
64:             16384             16384
32:             16384             16384
```

```
16:           16384           16384
64:           32768           32768
32:           32768           32768
16:           32768          -32768
64:           65536           65536
32:           65536           65536
16:               0               0
64:          131072          131072
32:          131072          131072
16:               0               0
64:          262144          262144
32:          262144          262144
16:               0               0
64:          524288          524288
32:          524288          524288
16:               0               0
64:         1048576         1048576
32:         1048576         1048576
16:               0               0
64:         2097152         2097152
32:         2097152         2097152
16:               0               0
64:         4194304         4194304
32:         4194304         4194304
16:               0               0
64:         8388608         8388608
32:         8388608         8388608
16:               0               0
64:        16777216        16777216
32:        16777216        16777216
16:               0               0
64:        33554432        33554432
32:        33554432        33554432
16:               0               0
64:        67108864        67108864
32:        67108864        67108864
16:               0               0
64:       134217728       134217728
32:       134217728       134217728
16:               0               0
```

```
64:         268435456              268435456
32:         268435456              268435456
16:                 0                      0
64:         536870912              536870912
32:         536870912              536870912
16:                 0                      0
64:        1073741824             1073741824
32:        1073741824             1073741824
16:                 0                      0
64:        2147483648             2147483648
32:        2147483648             2147483648
16:                 0                      0
64:        4294967296             4294967296
32:                 0                      0
16:                 0                      0
64:        8589934592             8589934592
32:                 0                      0
16:                 0                      0
64:       17179869184            17179869184
32:                 0                      0
16:                 0                      0
64:       34359738368            34359738368
32:                 0                      0
16:                 0                      0
64:       68719476736            68719476736
32:                 0                      0
16:                 0                      0
64:      137438953472           137438953472
32:                 0                      0
16:                 0                      0
64:      274877906944           274877906944
32:                 0                      0
16:                 0                      0
64:      549755813888           549755813888
32:                 0                      0
16:                 0                      0
64:     1099511627776          1099511627776
32:                 0                      0
16:                 0                      0
64:     2199023255552          2199023255552
```

```
32:                    0                    0
16:                    0                    0
64:        4398046511104        4398046511104
32:                    0                    0
16:                    0                    0
64:        8796093022208        8796093022208
32:                    0                    0
16:                    0                    0
64:       17592186044416       17592186044416
32:                    0                    0
16:                    0                    0
64:       35184372088832       35184372088832
32:                    0                    0
16:                    0                    0
64:       70368744177664       70368744177664
32:                    0                    0
16:                    0                    0
64:      140737488355328      140737488355328
32:                    0                    0
16:                    0                    0
64:      281474976710656      281474976710656
32:                    0                    0
16:                    0                    0
64:      562949953421312      562949953421312
32:                    0                    0
16:                    0                    0
64:     1125899906842624     1125899906842624
32:                    0                    0
16:                    0                    0
64:     2251799813685248     2251799813685248
32:                    0                    0
16:                    0                    0
64:     4503599627370496     4503599627370496
32:                    0                    0
16:                    0                    0
64:     9007199254740992     9007199254740992
32:                    0                    0
16:                    0                    0
64:    18014398509481984    18014398509481984
32:                    0                    0
```

```
16:                           0                          0
64:       36028797018963968       36028797018963968
32:                           0                          0
16:                           0                          0
64:       72057594037927936       72057594037927936
32:                           0                          0
16:                           0                          0
64:     144115188075855872     144115188075855872
32:                           0                          0
16:                           0                          0
64:     288230376151711744     288230376151711744
32:                           0                          0
16:                           0                          0
64:     576460752303423488     576460752303423488
32:                           0                          0
16:                           0                          0
64:   1152921504606846976   1152921504606846976
32:                           0                          0
16:                           0                          0
64:   2305843009213693952   2305843009213693952
32:                           0                          0
16:                           0                          0
64:   4611686018427387904   4611686018427387904
32:                           0                          0
16:                           0                          0
64:   9223372036854775808  -9223372036854775808
32:                           0                          0
16:                           0                          0
64:                           0                          0
32:                           0                          0
16:                           0                          0
```

### 1.5.7  Right Shift

Integer Division by 2. Shifts bits to the right. Do not use negative values.
WARNING: Shifting on signed integers is not safe or portable.

```
#include <stdio.h>
#include <inttypes.h>
int main() {
```

```
    // 0xF = 0b1111
    // 0x7 = 0b0111
    uint64_t u64 = 0x7FFFFFFFFFFFFFFF;
    uint32_t u32 = 0x7FFFFFFF;
    uint16_t u16 = 0x7FFF;
    int64_t  i64 = 0x7FFFFFFFFFFFFFFF;
    int32_t  i32 = 0x7FFFFFFF;
    uint16_t i16 = 0x7FFF;
    for (int i = 0 ; i < 64; i++) {
        u64 = u64 >> 1;
        i64 = i64 >> 1;
        u32 = u32 >> 1;
        i32 = i32 >> 1;
        u16 = u16 >> 1;
        i16 = i16 >> 1;
        printf("64: %20llu %20lld\n", u64, i64);
        printf("32: %20lu %20ld\n",   u32, i32);
        printf("16: %20zu %20zd\n",   u16, i16);
    }
}
```

```
64:   4611686018427387903   4611686018427387903
32:            1073741823            1073741823
16:                 16383                 16383
64:   2305843009213693951   2305843009213693951
32:             536870911             536870911
16:                  8191                  8191
64:   1152921504606846975   1152921504606846975
32:             268435455             268435455
16:                  4095                  4095
64:    576460752303423487    576460752303423487
32:             134217727             134217727
16:                  2047                  2047
64:    288230376151711743    288230376151711743
32:              67108863              67108863
16:                  1023                  1023
64:    144115188075855871    144115188075855871
32:              33554431              33554431
16:                   511                   511
64:     72057594037927935     72057594037927935
```

```
32:             16777215                   16777215
16:                   255                        255
64:     36028797018963967          36028797018963967
32:               8388607                    8388607
16:                   127                        127
64:     18014398509481983          18014398509481983
32:               4194303                    4194303
16:                    63                         63
64:      9007199254740991           9007199254740991
32:               2097151                    2097151
16:                    31                         31
64:      4503599627370495           4503599627370495
32:               1048575                    1048575
16:                    15                         15
64:      2251799813685247           2251799813685247
32:                524287                     524287
16:                     7                          7
64:      1125899906842623           1125899906842623
32:                262143                     262143
16:                     3                          3
64:       562949953421311            562949953421311
32:                131071                     131071
16:                     1                          1
64:       281474976710655            281474976710655
32:                 65535                      65535
16:                     0                          0
64:       140737488355327            140737488355327
32:                 32767                      32767
16:                     0                          0
64:        70368744177663             70368744177663
32:                 16383                      16383
16:                     0                          0
64:        35184372088831             35184372088831
32:                  8191                       8191
16:                     0                          0
64:        17592186044415             17592186044415
32:                  4095                       4095
16:                     0                          0
64:         8796093022207              8796093022207
32:                  2047                       2047
```

```
16:                        0                        0
64:            4398046511103            4398046511103
32:                     1023                     1023
16:                        0                        0
64:            2199023255551            2199023255551
32:                      511                      511
16:                        0                        0
64:            1099511627775            1099511627775
32:                      255                      255
16:                        0                        0
64:             549755813887             549755813887
32:                      127                      127
16:                        0                        0
64:             274877906943             274877906943
32:                       63                       63
16:                        0                        0
64:             137438953471             137438953471
32:                       31                       31
16:                        0                        0
64:              68719476735              68719476735
32:                       15                       15
16:                        0                        0
64:              34359738367              34359738367
32:                        7                        7
16:                        0                        0
64:              17179869183              17179869183
32:                        3                        3
16:                        0                        0
64:               8589934591               8589934591
32:                        1                        1
16:                        0                        0
64:               4294967295               4294967295
32:                        0                        0
16:                        0                        0
64:               2147483647               2147483647
32:                        0                        0
16:                        0                        0
64:               1073741823               1073741823
32:                        0                        0
16:                        0                        0
```

```
64:            536870911              536870911
32:                    0                      0
16:                    0                      0
64:            268435455              268435455
32:                    0                      0
16:                    0                      0
64:            134217727              134217727
32:                    0                      0
16:                    0                      0
64:             67108863               67108863
32:                    0                      0
16:                    0                      0
64:             33554431               33554431
32:                    0                      0
16:                    0                      0
64:             16777215               16777215
32:                    0                      0
16:                    0                      0
64:              8388607                8388607
32:                    0                      0
16:                    0                      0
64:              4194303                4194303
32:                    0                      0
16:                    0                      0
64:              2097151                2097151
32:                    0                      0
16:                    0                      0
64:              1048575                1048575
32:                    0                      0
16:                    0                      0
64:               524287                 524287
32:                    0                      0
16:                    0                      0
64:               262143                 262143
32:                    0                      0
16:                    0                      0
64:               131071                 131071
32:                    0                      0
16:                    0                      0
64:                65535                  65535
```

```
32:                    0                    0
16:                    0                    0
64:                32767                32767
32:                    0                    0
16:                    0                    0
64:                16383                16383
32:                    0                    0
16:                    0                    0
64:                 8191                 8191
32:                    0                    0
16:                    0                    0
64:                 4095                 4095
32:                    0                    0
16:                    0                    0
64:                 2047                 2047
32:                    0                    0
16:                    0                    0
64:                 1023                 1023
32:                    0                    0
16:                    0                    0
64:                  511                  511
32:                    0                    0
16:                    0                    0
64:                  255                  255
32:                    0                    0
16:                    0                    0
64:                  127                  127
32:                    0                    0
16:                    0                    0
64:                   63                   63
32:                    0                    0
16:                    0                    0
64:                   31                   31
32:                    0                    0
16:                    0                    0
64:                   15                   15
32:                    0                    0
16:                    0                    0
64:                    7                    7
32:                    0                    0
```

```
16:                        0                    0
64:                        3                    3
32:                        0                    0
16:                        0                    0
64:                        1                    1
32:                        0                    0
16:                        0                    0
64:                        0                    0
32:                        0                    0
16:                        0                    0
64:                        0                    0
32:                        0                    0
16:                        0                    0
```

### 1.5.8  Could we access the bits with shifts?

```
#include <stdio.h>
#include <inttypes.h>
#define BIT(x,b,y) ((x<<(b-y-1))>>(b-1))
#define BITS 32
int main() {
    // 0x7 = 0b0111
    uint32_t u32 = 0xFF770ABE;
    char str[BITS+1] = { '\0' };
    for (int i = 0 ; i < BITS; i++) {
        str[BITS-1-i] = (BIT(u32,BITS,i))?'1':'0';
    }
    printf("u32: %s\n", str);
}
```

```
u32: 11111111011101110000101010111110
```

1. Bitprinter.h

   ```
   #ifndef _BITPRINTER_H_
   #define _BITPRINTER_H_
   #include <inttypes.h>
   // Warning some bad hacks here

   #define BIT(x,b,y) ((x<<(b-y-1))>>(b-1))
   #define MAXBITSTRBITS 129
   ```

```
static char _bitstr[MAXBITSTRBITS] = { '\0' };
static char * bitString(uint64_t value, const unsigned int bits) {
    for (unsigned int i = 0 ; i < bits; i++) {
        _bitstr[bits-1-i] = (BIT(value,bits,i))?'1':'0';
    }
    _bitstr[bits] = '\0';
    return _bitstr;
}
static char * bitString64(uint64_t value) {
    return bitString(value,64);
}
static char * bitString32(uint32_t value) {
    return bitString(value,32);
}
static char * bitString16(uint16_t value) {
    return bitString(value,16);
}
static char * bitString8(uint8_t value) {
    return bitString(value,8);
}
#endif
```

### 1.5.9   bitwise and

The & operator is bitwise complement. It means every bit of an integer is and'd.

The & operator is a binary operator.

- 0 & 0 -> 0

- 0 & 1 -> 0

- 1 & 0 -> 0

- 1 & 1 -> 1

0b11111111011101110000101010111110 & 0b1111111111111111111111111111111111
0b11111111011101110000101010111110 & 0b00000000000000000000000000000000
0b0

0b011 & 0b010 == 0b010
0x00000001 & 0xFFFFFFFF = 0x1
if (0x10 & input) {

19

```
    }

#include <stdio.h>
#include <inttypes.h>
int main() {
    uint32_t pressF = 0xFFFFFFFF;
    printf("Anding 1 bit\n");
    for (uint32_t i = 0 ; i < 32; i++) {
        uint32_t bit = (1 << i);
        printf("%12lu & %12lu = %12lu - 0x%08x\n",
                pressF, bit,  pressF & bit, pressF & bit);
    }
    for (uint32_t i = 0 ; i < 32; i++) {
        uint32_t bit = (1 << i);
        printf("%08x & %08x = %08x\n",
                pressF, bit,  pressF & bit);
    }

}

Anding 1 bit
  4294967295 &            1 =            1 - 0x00000001
  4294967295 &            2 =            2 - 0x00000002
  4294967295 &            4 =            4 - 0x00000004
  4294967295 &            8 =            8 - 0x00000008
  4294967295 &           16 =           16 - 0x00000010
  4294967295 &           32 =           32 - 0x00000020
  4294967295 &           64 =           64 - 0x00000040
  4294967295 &          128 =          128 - 0x00000080
  4294967295 &          256 =          256 - 0x00000100
  4294967295 &          512 =          512 - 0x00000200
  4294967295 &         1024 =         1024 - 0x00000400
  4294967295 &         2048 =         2048 - 0x00000800
  4294967295 &         4096 =         4096 - 0x00001000
  4294967295 &         8192 =         8192 - 0x00002000
  4294967295 &        16384 =        16384 - 0x00004000
  4294967295 &        32768 =        32768 - 0x00008000
  4294967295 &        65536 =        65536 - 0x00010000
  4294967295 &       131072 =       131072 - 0x00020000
  4294967295 &       262144 =       262144 - 0x00040000
```

```
4294967295 &       524288 =       524288 - 0x00080000
4294967295 &      1048576 =      1048576 - 0x00100000
4294967295 &      2097152 =      2097152 - 0x00200000
4294967295 &      4194304 =      4194304 - 0x00400000
4294967295 &      8388608 =      8388608 - 0x00800000
4294967295 &     16777216 =     16777216 - 0x01000000
4294967295 &     33554432 =     33554432 - 0x02000000
4294967295 &     67108864 =     67108864 - 0x04000000
4294967295 &    134217728 =    134217728 - 0x08000000
4294967295 &    268435456 =    268435456 - 0x10000000
4294967295 &    536870912 =    536870912 - 0x20000000
4294967295 &   1073741824 =   1073741824 - 0x40000000
4294967295 &   2147483648 =   2147483648 - 0x80000000
ffffffff & 00000001 = 00000001
ffffffff & 00000002 = 00000002
ffffffff & 00000004 = 00000004
ffffffff & 00000008 = 00000008
ffffffff & 00000010 = 00000010
ffffffff & 00000020 = 00000020
ffffffff & 00000040 = 00000040
ffffffff & 00000080 = 00000080
ffffffff & 00000100 = 00000100
ffffffff & 00000200 = 00000200
ffffffff & 00000400 = 00000400
ffffffff & 00000800 = 00000800
ffffffff & 00001000 = 00001000
ffffffff & 00002000 = 00002000
ffffffff & 00004000 = 00004000
ffffffff & 00008000 = 00008000
ffffffff & 00010000 = 00010000
ffffffff & 00020000 = 00020000
ffffffff & 00040000 = 00040000
ffffffff & 00080000 = 00080000
ffffffff & 00100000 = 00100000
ffffffff & 00200000 = 00200000
ffffffff & 00400000 = 00400000
ffffffff & 00800000 = 00800000
ffffffff & 01000000 = 01000000
ffffffff & 02000000 = 02000000
ffffffff & 04000000 = 04000000
```

```
ffffffff & 08000000 = 08000000
ffffffff & 10000000 = 10000000
ffffffff & 20000000 = 20000000
ffffffff & 40000000 = 40000000
ffffffff & 80000000 = 80000000
```

1. Uses of &

   (a) Checking for bits

```c
#include <inttypes.h>
#include <stdio.h>

// 0x1L MUST be used 0x1 causes bugs
#define BIT(x,y) (x & (0x1L << y))

int main() {
    printf("%lu\n",sizeof(0x1L));
    printf("%lu\n",sizeof(0x1));
}

8
4

#include <inttypes.h>
#include <stdio.h>

// 0x1L MUST be used 0x1 causes bugs
#define BIT(x,y) (x & (0x1L << y))

int main() {
    uint64_t interesting = 0x0123456789ABCDEF;
    puts("Lets see some bits!");
    for (size_t i = 64 ; i > 0; i--) {
        putchar(BIT(interesting, i-1)?'1':'0');
    }
    putchar('\n');
    for (size_t i = 0 ; i < 64; i++) {
        putchar('0' + i%10);
    }
    putchar('\n');
}
```

```
Lets see some bits!
000000010010001101000101011001111000100110101011110011011 1101111
0123456789012345678901234567890123456789012345678901234567890123
```

(b) Bit Printing

    & is way safer than shift for bit printing.

```c
#include <inttypes.h>
#include <stdio.h>

// 0x1L MUST be used 0x1 causes bugs
#define BIT(x,y) (x & (0x1L << y))
#define MAXBITSTRBITS 129
static char _bitstr[MAXBITSTRBITS] = { '\0' };
static char * bitString(uint64_t value, const unsigned int bits) {
    // iterator must be uint64_t
    for (uint64_t i = 0 ; i < bits; i++) {
        char bit = (BIT(value,i))?'1':'0';
        _bitstr[bits-1-i] = bit;
    }
    _bitstr[bits] = '\0';
    return _bitstr;
}

int main() {
    uint64_t interesting[] = {
        0x7F,
        0xFF,
        0xFFF,
        0xABE4BEEF,
        0x7777777777,
        0xFFFFFFFFFF,
        0xABCDEF01234,
        0x7FFFFFFFFFFFFFFF,
        0xFFFFFFFFFFFFFFFF,
    };
    const size_t nints = sizeof(interesting)/sizeof(uint64_t);
    printf("Interesting numbers!\n");
    for (size_t i = 0 ; i < nints; i++) {
        printf("%16lx %20llu %s\n",
```

```
                interesting[i],
                interesting[i],
                bitString(interesting[i],64)
        );
    }
    printf("Interesting[0] & Interesting[1]");
    for (size_t i = 0 ; i < nints; i++) {
        printf("%s\n",
                bitString(interesting[i] & interesting[(i+1)%nints],64)
        );
    }


}
```

```
Interesting numbers!
              7f                        127  00000000000000000000000000000000000
              ff                        255  00000000000000000000000000000000000
             fff                       4095  00000000000000000000000000000000000
        abe4beef                 2883895023  00000000000000000000000000001010101
      7777777777                513105426295  000000000000000000000000011101110111011
      ffffffffff               1099511627775  0000000000000000000000001111111111111111
     abcdef01234              11806310404660  00000000000000000001010101111001101111
7fffffffffffffff          9223372036854775807  0111111111111111111111111111111111111
ffffffffffffffff         18446744073709551615  1111111111111111111111111111111111111
Interesting[0] & Interesting[1]0000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000011111111
00000000000000000000000000000000000000000000000000000000111011101111
0000000000000000000000000000000000010001101100100001101100110011
00000000000000000000000000011101110111011101110111011101110111
00000000000000000000000001011110011011110111100000001001000110100
000000000000000000010101011110011011110111100000001001000110100
01111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000001111111
```

## 1.5.10   bitwise OR

The | operator is bitwise complement. It is the or operator between bits.
    The | operator is a binary operator.

- 0 | 0 -> 0

- 0 | 1 -> 1

- 1 | 0 -> 1

- 1 | 1 -> 1

```c
#include <stdio.h>
#include <inttypes.h>
int main() {
    uint32_t pressF = 0xFFFFFFFF;
    printf("ORing 1 bit\n");
    for (uint32_t i = 0 ; i < 32; i++) {
        uint32_t bit = (1 << i);
        printf("%08x & %08x = %08x\n",
                pressF, bit,  pressF | bit);
    }
    pressF = 0x11111111;
    for (uint32_t i = 0 ; i < 32; i++) {
        uint32_t bit = (1 << i);
        printf("%08x & %08x = %08x\n",
                pressF, bit,  pressF | bit);
    }
}
```

```
ORing 1 bit
ffffffff & 00000001 = ffffffff
ffffffff & 00000002 = ffffffff
ffffffff & 00000004 = ffffffff
ffffffff & 00000008 = ffffffff
ffffffff & 00000010 = ffffffff
ffffffff & 00000020 = ffffffff
ffffffff & 00000040 = ffffffff
ffffffff & 00000080 = ffffffff
ffffffff & 00000100 = ffffffff
ffffffff & 00000200 = ffffffff
ffffffff & 00000400 = ffffffff
ffffffff & 00000800 = ffffffff
ffffffff & 00001000 = ffffffff
ffffffff & 00002000 = ffffffff
ffffffff & 00004000 = ffffffff
ffffffff & 00008000 = ffffffff
```

```
ffffffff & 00010000 = ffffffff
ffffffff & 00020000 = ffffffff
ffffffff & 00040000 = ffffffff
ffffffff & 00080000 = ffffffff
ffffffff & 00100000 = ffffffff
ffffffff & 00200000 = ffffffff
ffffffff & 00400000 = ffffffff
ffffffff & 00800000 = ffffffff
ffffffff & 01000000 = ffffffff
ffffffff & 02000000 = ffffffff
ffffffff & 04000000 = ffffffff
ffffffff & 08000000 = ffffffff
ffffffff & 10000000 = ffffffff
ffffffff & 20000000 = ffffffff
ffffffff & 40000000 = ffffffff
ffffffff & 80000000 = ffffffff
11111111 & 00000001 = 11111111
11111111 & 00000002 = 11111113
11111111 & 00000004 = 11111115
11111111 & 00000008 = 11111119
11111111 & 00000010 = 11111111
11111111 & 00000020 = 11111131
11111111 & 00000040 = 11111151
11111111 & 00000080 = 11111191
11111111 & 00000100 = 11111111
11111111 & 00000200 = 11111311
11111111 & 00000400 = 11111511
11111111 & 00000800 = 11111911
11111111 & 00001000 = 11111111
11111111 & 00002000 = 11113111
11111111 & 00004000 = 11115111
11111111 & 00008000 = 11119111
11111111 & 00010000 = 11111111
11111111 & 00020000 = 11131111
11111111 & 00040000 = 11151111
11111111 & 00080000 = 11191111
11111111 & 00100000 = 11111111
11111111 & 00200000 = 11311111
11111111 & 00400000 = 11511111
11111111 & 00800000 = 11911111
```

```
11111111 & 01000000 = 11111111
11111111 & 02000000 = 13111111
11111111 & 04000000 = 15111111
11111111 & 08000000 = 19111111
11111111 & 10000000 = 11111111
11111111 & 20000000 = 31111111
11111111 & 40000000 = 51111111
11111111 & 80000000 = 91111111
```

1. Uses of |

   We mostly use | to combine bits together for arguments to functions.

### 1.5.11   bitwise XOR

The ^ operator is bitwise exclusive OR. It is the XOR operator between bits. It differs from or because when both inputs bits are hot the result in 0.

The ^ operator is a binary operator.

- 0 | 0 -> 0

- 0 | 1 -> 1

- 1 | 0 -> 1

- 1 | 1 -> 0

```
#include <stdio.h>
#include <inttypes.h>
int main() {
    uint32_t pressF = 0xFFFFFFFF;
    printf("ORing 1 bit\n");
    for (uint32_t i = 0 ; i < 32; i++) {
        uint32_t bit = (1 << i);
        printf("%08x & %08x = %08x\n",
                pressF, bit,  pressF ^ bit);
    }
    pressF = 0x11111111;
    for (uint32_t i = 0 ; i < 32; i++) {
        uint32_t bit = (1 << i);
        printf("%08x & %08x = %08x\n",
                pressF, bit,  pressF ^ bit);
```

```
    }
}

ORing 1 bit
ffffffff & 00000001 = fffffffe
ffffffff & 00000002 = fffffffd
ffffffff & 00000004 = fffffffb
ffffffff & 00000008 = fffffff7
ffffffff & 00000010 = ffffffef
ffffffff & 00000020 = ffffffdf
ffffffff & 00000040 = ffffffbf
ffffffff & 00000080 = ffffff7f
ffffffff & 00000100 = fffffeff
ffffffff & 00000200 = fffffdff
ffffffff & 00000400 = fffffbff
ffffffff & 00000800 = fffff7ff
ffffffff & 00001000 = ffffefff
ffffffff & 00002000 = ffffdfff
ffffffff & 00004000 = ffffbfff
ffffffff & 00008000 = ffff7fff
ffffffff & 00010000 = fffeffff
ffffffff & 00020000 = fffdffff
ffffffff & 00040000 = fffbffff
ffffffff & 00080000 = fff7ffff
ffffffff & 00100000 = ffefffff
ffffffff & 00200000 = ffdfffff
ffffffff & 00400000 = ffbfffff
ffffffff & 00800000 = ff7fffff
ffffffff & 01000000 = feffffff
ffffffff & 02000000 = fdffffff
ffffffff & 04000000 = fbffffff
ffffffff & 08000000 = f7ffffff
ffffffff & 10000000 = efffffff
ffffffff & 20000000 = dfffffff
ffffffff & 40000000 = bfffffff
ffffffff & 80000000 = 7fffffff
11111111 & 00000001 = 11111110
11111111 & 00000002 = 11111113
11111111 & 00000004 = 11111115
11111111 & 00000008 = 11111119
```

```
11111111 & 00000010 = 11111101
11111111 & 00000020 = 11111131
11111111 & 00000040 = 11111151
11111111 & 00000080 = 11111191
11111111 & 00000100 = 11111011
11111111 & 00000200 = 11111311
11111111 & 00000400 = 11111511
11111111 & 00000800 = 11111911
11111111 & 00001000 = 11110111
11111111 & 00002000 = 11113111
11111111 & 00004000 = 11115111
11111111 & 00008000 = 11119111
11111111 & 00010000 = 11101111
11111111 & 00020000 = 11131111
11111111 & 00040000 = 11151111
11111111 & 00080000 = 11191111
11111111 & 00100000 = 11011111
11111111 & 00200000 = 11311111
11111111 & 00400000 = 11511111
11111111 & 00800000 = 11911111
11111111 & 01000000 = 10111111
11111111 & 02000000 = 13111111
11111111 & 04000000 = 15111111
11111111 & 08000000 = 19111111
11111111 & 10000000 = 01111111
11111111 & 20000000 = 31111111
11111111 & 40000000 = 51111111
11111111 & 80000000 = 91111111
```

1. Uses of ^

   XOR has interesting properties .

   a ^ b ^ a = b

   So if we send a message of $a^b$ we can decode the message of $(a^b)^b$

   ```
   #include <stdio.h>
   #include <inttypes.h>

   void encrypt(char * output, const char * key, char * input) {
       const size_t keylen   = strlen(key);
   ```

```
        const size_t inputlen = strlen(input);
        for (size_t i = 0 ; i < inputlen; i++) {
            output[i] = input[i] ^ key[i%keylen];
        }
        output[inputlen] = '\0';
    }

    int main() {
        const char * key = "EATFOOD";
        const size_t keylen = strlen(key);
        char * input = "I enjoy olive bread";
        char output[1024] = {'\0'};
        encrypt(output, key, input);
        printf("Encrypted: %s\n", output);
        encrypt(output, key, output);
        printf("Encrypted Again: %s\n", output);
    }

    Encrypted:
    a1(% =e.8/9*d'31'+
    Encrypted Again: I enjoy olive bread
```

## 1.5.12  bitwise complement

The ~ operator is bitwise complement. It means every bit is notted The ~ operator is a unary operator.

- 0 -> 1

- 1 -> 0

```
#include <stdio.h>
#include <inttypes.h>
int main() {
    uint8_t u8 = 0;
    for (uint32_t i = 0 ; i < 256; i+=6) {
        u8 = (uint8_t)i;
        // be careful about how you deal with chars
        printf("%3hhu %3hhu\t-- %3u %3u\n",
                u8, ~u8, u8, ~u8);
    }
}
```

```
  0 255 --   0 4294967295
  6 249 --   6 4294967289
 12 243 --  12 4294967283
 18 237 --  18 4294967277
 24 231 --  24 4294967271
 30 225 --  30 4294967265
 36 219 --  36 4294967259
 42 213 --  42 4294967253
 48 207 --  48 4294967247
 54 201 --  54 4294967241
 60 195 --  60 4294967235
 66 189 --  66 4294967229
 72 183 --  72 4294967223
 78 177 --  78 4294967217
 84 171 --  84 4294967211
 90 165 --  90 4294967205
 96 159 --  96 4294967199
102 153 -- 102 4294967193
108 147 -- 108 4294967187
114 141 -- 114 4294967181
120 135 -- 120 4294967175
126 129 -- 126 4294967169
132 123 -- 132 4294967163
138 117 -- 138 4294967157
144 111 -- 144 4294967151
150 105 -- 150 4294967145
156  99 -- 156 4294967139
162  93 -- 162 4294967133
168  87 -- 168 4294967127
174  81 -- 174 4294967121
180  75 -- 180 4294967115
186  69 -- 186 4294967109
192  63 -- 192 4294967103
198  57 -- 198 4294967097
204  51 -- 204 4294967091
210  45 -- 210 4294967085
216  39 -- 216 4294967079
222  33 -- 222 4294967073
228  27 -- 228 4294967067
234  21 -- 234 4294967061
```

```
240  15 -- 240 4294967055
246   9 -- 246 4294967049
252   3 -- 252 4294967043
```

### 1.5.13  Flags

In C flags are often used. Flags are parameters who's bits indicate if some option is chosen.

'man 2 open' has the open function

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
```

open is part of POSIX systems like linux. open takes a file name (pathname) and takes flags:

- $O_{\text{RDONLY}}$ - Read the file

- $O_{\text{WRONLY}}$ - Write to the file

- $O_{\text{RDWR}}$ - Read and Write to the file

- $O_{\text{CREAT}}$ - CREATE the file if it doesn't exist

- $O_{\text{APPEND}}$ - append to existing file

But how do we open a file for writing that creates a new file with only 1 argument?

The bitwise | operator!

$O_{\text{WRONLY}}$ | $O_{\text{CREAT}}$ will allow us to open a file for writing that will create the file on disk if it doesn't already exist!

```
// in /usr/include/asm-generic/fcntl.h (used by fcntl.h)
// we see the definitions:
#define O_ACCMODE 00000003
#define O_RDONLY 00000000
#define O_WRONLY 00000001
#define O_RDWR 00000002
#ifndef O_CREAT
#define O_CREAT 00000100 /* not fcntl */
```

```
#endif
#ifndef O_EXCL
#define O_EXCL 00000200 /* not fcntl */
#endif
#ifndef O_NOCTTY
#define O_NOCTTY 00000400 /* not fcntl */
#endif
#ifndef O_TRUNC
#define O_TRUNC 00001000 /* not fcntl */
#endif
#ifndef O_APPEND
#define O_APPEND 00002000
#endif

open("~/.bashrc", O_RDONLY); // open ~/.bashrc for read only access
open("/tmp/map.txt", O_WRONLY | O_CREATE); // open /tmp/map.txt to write to it
open("./log.txt", O_WRONLY | O_CREATE | O_APPEND); // open ./map.txt to write to it or

// So
O_WRONLY | O_CREATE            == 00000101 (00000001 | 00000100)
O_WRONLY | O_CREATE | O_APPEND == 00002101 (00000001 | 00000100 | 00002000)
// Question wht number format are they using to define these flags?
```

warning: open is just an example. Please use fopen if you want to read / write files.

warning: flags should only be unsigned integers

warning: flags must be not 0

1. Expected Flag use:

   Use | to set a flag: FLAG1 | FLAG2

   Use & to check a flag: input & FLAG1

   Watch out testing for truth and combining flags

   To check for FLAG1 and FLAG2 being hot you should:

   'if (input & FLAG1 && input & FLAG2) '

   or

   'if (input & (FLAG1 | FLAG2) == (FLAG1 | FLAG2))'

   It's very convoluted so don't get too complicated

This is a bug: 'if (input & (FLAG1 | FLAG2))'

It actually just checks if you have either FLAG1 or FLAG2.

2. Flags Example

   Hazel talks in depth about pizza

   - https://docs.google.com/document/d/1S-I-OthHf0mgNZXnTSOvyx5lSduQwsbHYMmVPLtTl
     edit#

```c
#include <inttypes.h>
#include <stdio.h>

enum pizza_toppings {
    CHEESE          =  0x1,
    THINCRUST       =  0x2,
    THICKCRUST      =  0x4,
    TOMATOSAUCE     =  0x8,
    BLACKMETALSAUCE = 0x10,
    CORN            = 0x20, // Hazel's favourite topping
    ONIONS          = 0x40,
    MYSTERYMEAT     = 0x80,
};

typedef uint32_t PizzaFlags;

void printPizza(PizzaFlags pizza) {
    // Don't do what doctor hindle does.
    // you should use a loop over the constants instead
    printf("A pizza of %s%s%s%s%s%s%s%s.\n",
            (pizza & CHEESE)?"cheese, ":"",
            (pizza & THINCRUST)?"thin crust, ":"",
            (pizza & THICKCRUST)?"thick crust, ":"",
            (pizza & TOMATOSAUCE)?"tomato sauce, ":"",
            (pizza & BLACKMETALSAUCE)?"black metal sauce, ":"",
            (pizza & CORN)?"corn (srsly?), ":"",
            (pizza & ONIONS)?"onions, ":"",
            (pizza & MYSTERYMEAT)?"exotic meats, ":""
    );
}
void violation(PizzaFlags pizza) {
```

34

```c
    // Not all pizzas are created equal
    if (pizza & THINCRUST && pizza & THICKCRUST) {
        printf("WARNING: Impossible to make a thin / thick crust pizza!\n");
    }
    if (pizza & CORN) {
        printf("WARNING: Corn? REALLY?\n");
    }
    if (pizza == (CHEESE | THINCRUST) || pizza == (CHEESE | THICKCRUST)) {
        printf("WARNING: SPAAAAAAAARTA, or spartan.\n");
    }
}
int main() {
    printf("OK build some pizza using |\n\n\n");
    printPizza(CHEESE | CORN);
    printPizza(CHEESE |  THINCRUST | BLACKMETALSAUCE | CORN | MYSTERYMEAT);
    printf("\n\nOK build some pizza via iteration\n\n\n");
    for (uint32_t i = 0; i < 0x8F; i++) {
        printPizza((PizzaFlags)i);
    }
    // Check the pizza
    printf("\n\nOK check some pizzas yo\n\n\n");
    PizzaFlags pizza = CHEESE | CORN | THINCRUST | THICKCRUST;
    printPizza(pizza);
    violation(pizza);
    pizza = CHEESE | CORN | THINCRUST;
    printPizza(pizza);
    violation(pizza);
    pizza = CHEESE | THINCRUST;
    printPizza(pizza);
    violation(pizza);
    // TOGGLE
    printf("\n\nLet's use XOR to toggle flags!\n");
    pizza = CHEESE | CORN | THINCRUST | THICKCRUST;
    pizza = pizza ^ CORN; // NOPE
    pizza = pizza ^ THINCRUST; // NOPE
    printPizza(pizza);
    violation(pizza);
    pizza = pizza ^ CORN; // CORN IS BACK!
    printPizza(pizza);
    violation(pizza);
```

```
}
```

OK build some pizza using |


A pizza of cheese, corn (srsly?), .
A pizza of cheese, thin crust, black metal sauce, corn (srsly?), exotic meats, .


OK build some pizza via iteration


A pizza of .
A pizza of cheese, .
A pizza of thin crust, .
A pizza of cheese, thin crust, .
A pizza of thick crust, .
A pizza of cheese, thick crust, .
A pizza of thin crust, thick crust, .
A pizza of cheese, thin crust, thick crust, .
A pizza of tomato sauce, .
A pizza of cheese, tomato sauce, .
A pizza of thin crust, tomato sauce, .
A pizza of cheese, thin crust, tomato sauce, .
A pizza of thick crust, tomato sauce, .
A pizza of cheese, thick crust, tomato sauce, .
A pizza of thin crust, thick crust, tomato sauce, .
A pizza of cheese, thin crust, thick crust, tomato sauce, .
A pizza of black metal sauce, .
A pizza of cheese, black metal sauce, .
A pizza of thin crust, black metal sauce, .
A pizza of cheese, thin crust, black metal sauce, .
A pizza of thick crust, black metal sauce, .
A pizza of cheese, thick crust, black metal sauce, .
A pizza of thin crust, thick crust, black metal sauce, .
A pizza of cheese, thin crust, thick crust, black metal sauce, .
A pizza of tomato sauce, black metal sauce, .

36

```
A pizza of cheese, tomato sauce, black metal sauce, .
A pizza of thin crust, tomato sauce, black metal sauce, .
A pizza of cheese, thin crust, tomato sauce, black metal sauce, .
A pizza of thick crust, tomato sauce, black metal sauce, .
A pizza of cheese, thick crust, tomato sauce, black metal sauce, .
A pizza of thin crust, thick crust, tomato sauce, black metal sauce, .
A pizza of cheese, thin crust, thick crust, tomato sauce, black metal sauce, .
A pizza of corn (srsly?), .
A pizza of cheese, corn (srsly?), .
A pizza of thin crust, corn (srsly?), .
A pizza of cheese, thin crust, corn (srsly?), .
A pizza of thick crust, corn (srsly?), .
A pizza of cheese, thick crust, corn (srsly?), .
A pizza of thin crust, thick crust, corn (srsly?), .
A pizza of cheese, thin crust, thick crust, corn (srsly?), .
A pizza of tomato sauce, corn (srsly?), .
A pizza of cheese, tomato sauce, corn (srsly?), .
A pizza of thin crust, tomato sauce, corn (srsly?), .
A pizza of cheese, thin crust, tomato sauce, corn (srsly?), .
A pizza of thick crust, tomato sauce, corn (srsly?), .
A pizza of cheese, thick crust, tomato sauce, corn (srsly?), .
A pizza of thin crust, thick crust, tomato sauce, corn (srsly?), .
A pizza of cheese, thin crust, thick crust, tomato sauce, corn (srsly?), .
A pizza of black metal sauce, corn (srsly?), .
A pizza of cheese, black metal sauce, corn (srsly?), .
A pizza of thin crust, black metal sauce, corn (srsly?), .
A pizza of cheese, thin crust, black metal sauce, corn (srsly?), .
A pizza of thick crust, black metal sauce, corn (srsly?), .
A pizza of cheese, thick crust, black metal sauce, corn (srsly?), .
A pizza of thin crust, thick crust, black metal sauce, corn (srsly?), .
A pizza of cheese, thin crust, thick crust, black metal sauce, corn (srsly?), .
A pizza of tomato sauce, black metal sauce, corn (srsly?), .
A pizza of cheese, tomato sauce, black metal sauce, corn (srsly?), .
A pizza of thin crust, tomato sauce, black metal sauce, corn (srsly?), .
A pizza of cheese, thin crust, tomato sauce, black metal sauce, corn (srsly?), .
A pizza of thick crust, tomato sauce, black metal sauce, corn (srsly?), .
A pizza of cheese, thick crust, tomato sauce, black metal sauce, corn (srsly?), .
A pizza of thin crust, thick crust, tomato sauce, black metal sauce, corn (srsly?)
A pizza of cheese, thin crust, thick crust, tomato sauce, black metal sauce, corn
A pizza of onions, .
```

```
A pizza of cheese, onions, .
A pizza of thin crust, onions, .
A pizza of cheese, thin crust, onions, .
A pizza of thick crust, onions, .
A pizza of cheese, thick crust, onions, .
A pizza of thin crust, thick crust, onions, .
A pizza of cheese, thin crust, thick crust, onions, .
A pizza of tomato sauce, onions, .
A pizza of cheese, tomato sauce, onions, .
A pizza of thin crust, tomato sauce, onions, .
A pizza of cheese, thin crust, tomato sauce, onions, .
A pizza of thick crust, tomato sauce, onions, .
A pizza of cheese, thick crust, tomato sauce, onions, .
A pizza of thin crust, thick crust, tomato sauce, onions, .
A pizza of cheese, thin crust, thick crust, tomato sauce, onions, .
A pizza of black metal sauce, onions, .
A pizza of cheese, black metal sauce, onions, .
A pizza of thin crust, black metal sauce, onions, .
A pizza of cheese, thin crust, black metal sauce, onions, .
A pizza of thick crust, black metal sauce, onions, .
A pizza of cheese, thick crust, black metal sauce, onions, .
A pizza of thin crust, thick crust, black metal sauce, onions, .
A pizza of cheese, thin crust, thick crust, black metal sauce, onions, .
A pizza of tomato sauce, black metal sauce, onions, .
A pizza of cheese, tomato sauce, black metal sauce, onions, .
A pizza of thin crust, tomato sauce, black metal sauce, onions, .
A pizza of cheese, thin crust, tomato sauce, black metal sauce, onions, .
A pizza of thick crust, tomato sauce, black metal sauce, onions, .
A pizza of cheese, thick crust, tomato sauce, black metal sauce, onions, .
A pizza of thin crust, thick crust, tomato sauce, black metal sauce, onions, .
A pizza of cheese, thin crust, thick crust, tomato sauce, black metal sauce, onion
A pizza of corn (srsly?), onions, .
A pizza of cheese, corn (srsly?), onions, .
A pizza of thin crust, corn (srsly?), onions, .
A pizza of cheese, thin crust, corn (srsly?), onions, .
A pizza of thick crust, corn (srsly?), onions, .
A pizza of cheese, thick crust, corn (srsly?), onions, .
A pizza of thin crust, thick crust, corn (srsly?), onions, .
A pizza of cheese, thin crust, thick crust, corn (srsly?), onions, .
A pizza of tomato sauce, corn (srsly?), onions, .
```

A pizza of cheese, tomato sauce, corn (srsly?), onions, .
A pizza of thin crust, tomato sauce, corn (srsly?), onions, .
A pizza of cheese, thin crust, tomato sauce, corn (srsly?), onions, .
A pizza of thick crust, tomato sauce, corn (srsly?), onions, .
A pizza of cheese, thick crust, tomato sauce, corn (srsly?), onions, .
A pizza of thin crust, thick crust, tomato sauce, corn (srsly?), onions, .
A pizza of cheese, thin crust, thick crust, tomato sauce, corn (srsly?), onions, .
A pizza of black metal sauce, corn (srsly?), onions, .
A pizza of cheese, black metal sauce, corn (srsly?), onions, .
A pizza of thin crust, black metal sauce, corn (srsly?), onions, .
A pizza of cheese, thin crust, black metal sauce, corn (srsly?), onions, .
A pizza of thick crust, black metal sauce, corn (srsly?), onions, .
A pizza of cheese, thick crust, black metal sauce, corn (srsly?), onions, .
A pizza of thin crust, thick crust, black metal sauce, corn (srsly?), onions, .
A pizza of cheese, thin crust, thick crust, black metal sauce, corn (srsly?), onic
A pizza of tomato sauce, black metal sauce, corn (srsly?), onions, .
A pizza of cheese, tomato sauce, black metal sauce, corn (srsly?), onions, .
A pizza of thin crust, tomato sauce, black metal sauce, corn (srsly?), onions, .
A pizza of cheese, thin crust, tomato sauce, black metal sauce, corn (srsly?), oni
A pizza of thick crust, tomato sauce, black metal sauce, corn (srsly?), onions, .
A pizza of cheese, thick crust, tomato sauce, black metal sauce, corn (srsly?), on
A pizza of thin crust, thick crust, tomato sauce, black metal sauce, corn (srsly?)
A pizza of cheese, thin crust, thick crust, tomato sauce, black metal sauce, corn
A pizza of exotic meats, .
A pizza of cheese, exotic meats, .
A pizza of thin crust, exotic meats, .
A pizza of cheese, thin crust, exotic meats, .
A pizza of thick crust, exotic meats, .
A pizza of cheese, thick crust, exotic meats, .
A pizza of thin crust, thick crust, exotic meats, .
A pizza of cheese, thin crust, thick crust, exotic meats, .
A pizza of tomato sauce, exotic meats, .
A pizza of cheese, tomato sauce, exotic meats, .
A pizza of thin crust, tomato sauce, exotic meats, .
A pizza of cheese, thin crust, tomato sauce, exotic meats, .
A pizza of thick crust, tomato sauce, exotic meats, .
A pizza of cheese, thick crust, tomato sauce, exotic meats, .
A pizza of thin crust, thick crust, tomato sauce, exotic meats, .

```
OK check some pizzas yo


A pizza of cheese, thin crust, thick crust, corn (srsly?), .
WARNING: Impossible to make a thin / thick crust pizza!
WARNING: Corn? REALLY?
A pizza of cheese, thin crust, corn (srsly?), .
WARNING: Corn? REALLY?
A pizza of cheese, thin crust, .
WARNING: SPAAAAAAAARTA, or spartan.


Let's use XOR to toggle flags!
A pizza of cheese, thick crust, .
WARNING: SPAAAAAAAARTA, or spartan.
A pizza of cheese, thick crust, corn (srsly?), .
WARNING: Corn? REALLY?
```

### 1.5.14   PRACTICE and More resources

- PLEASE Read Hazel's Notes on the subject

    - https://docs.google.com/document/d/1S-I-OthHf0mgNZXnTSOvyx5lSduQwsbHYMmVPLtTl
      edit#
    - UAlberta accounts only

- Please READ CHAPTER 20 OF THE TEXTBOOK. Then practice
  your binary skills!

    - http://www.free-test-online.com/binary/hex2bin.htm
    - http://www.free-test-online.com/binary/binary2hex.htm
    - http://www.free-test-online.com/binary/add_binary.htm
    - https://web.stanford.edu/class/cs107/lab1/practice.html

### 1.5.15   BitFields

```
struct bitfield {
    unsigned int b0:  1;
    unsigned int b1:  2;
```

```
    unsigned int b2:  4;
    unsigned int b3:  8;
    unsigned int b4: 16;
    unsigned int b5:  1;
    //unsigned int b6: 1;
};

int main() {
    struct bitfield b = { 0, 3, 12, 242, 65000 };
    printf("%hhu %hhu %hhu %hhu %hu\n", b.b0, b.b1, b.b2, b.b3, b.b4);
    printf("size: %lu\n", sizeof(b));
}
```

```
0 3 12 242 65000
size: 4
```

1. Chmod example

   Let's try bitfields and unix file permissions!

   - 1 execute
   - 2 write
   - 4 read
   - 7 read | write | execute

```
@piggy:/tmp/coolbears$ touch what
@piggy:/tmp/coolbears$ ls -l what
-rw-r--r-- 1 hindle1 hindle1 0 Mar 26 10:14 what
@piggy:/tmp/coolbears$ echo NEAT > what
@piggy:/tmp/coolbears$ cat what
NEAT
@piggy:/tmp/coolbears$ ls -l what
-rw-r--r-- 1 hindle1 hindle1 5 Mar 26 10:14 what
@piggy:/tmp/coolbears$ chmod 000 what
@piggy:/tmp/coolbears$ cat what
cat: what: Permission denied
@piggy:/tmp/coolbears$ ls -l what
---------- 1 hindle1 hindle1 5 Mar 26 10:14 what
@piggy:/tmp/coolbears$ chmod 444 what
```

```
@piggy:/tmp/coolbears$ ls -l
total 4
-r--r--r-- 1 hindle1 hindle1 5 Mar 26 10:14 what
@piggy:/tmp/coolbears$ cat what
NEAT
@piggy:/tmp/coolbears$ echo echo do I hear an echo > what
-bash: what: Permission denied
@piggy:/tmp/coolbears$ chmod 666 what
@piggy:/tmp/coolbears$ echo echo do I hear an echo > what
@piggy:/tmp/coolbears$ cat what
echo do I hear an echo
@piggy:/tmp/coolbears$ ./what
-bash: ./what: Permission denied
@piggy:/tmp/coolbears$ chmod 777 what
@piggy:/tmp/coolbears$ ./what
do I hear an echo
@piggy:/tmp/coolbears$ cat what
echo do I hear an echo
@piggy:/tmp/coolbears$ chmod 711 what
@piggy:/tmp/coolbears$ ls -l what
-rwx--x--x 1 hindle1 hindle1 23 Mar 26 10:15 what
```

Let's go to that chmod example I gave you where I set the

```c
#include <stdio.h>

struct unixperms {
    unsigned int execute:  1;
    unsigned int write:  1;
    unsigned int read:  1;
    unsigned int: 5; // try filling up to the next storage unit see what happens
};
// these don't stack well
struct fullperms {
   struct unixperms user;
   struct unixperms group;
   struct unixperms all;
};

// GCC is OK with this but clang is not so impressed
```

```c
union unixpun {
    unsigned int uint;
    struct unixperms perm;
};
union fullpun {
    unsigned char bytes[sizeof(struct fullperms)];
    struct fullperms perm;
};

int main() {
    union unixpun nonePun          = { .perm=(struct unixperms){0,0,0} };
    union unixpun readPun          = { .perm=(struct unixperms){0,0,1} };
    union unixpun readWritePun     = { .perm=(struct unixperms){0,1,1} };
    union unixpun readWriteExecPun = { .perm=(struct unixperms){1,1,1} };
    union unixpun readExecPun      = { .perm=(struct unixperms){1,0,1} };
    union unixpun puns[] = { nonePun, readPun, readWritePun,
                             readWriteExecPun, readExecPun };
    char * names[] = { "nonePun", "readPun", "readWritePun",
                             "readWriteExecPun", "readExecPun" };
    size_t npuns = sizeof(puns)/sizeof(puns[0]);
    printf("sizeof(struct unixperms)=%lu\n",sizeof(struct unixperms));
    for (size_t i = 0; i < npuns; i++) {
        printf("Pun:%16s E:%hhu W:%hhu R:%hhu [%4hhu] [%08x]\n", names[i],
                puns[i].perm.execute,
                puns[i].perm.write,
                puns[i].perm.read,
                puns[i].uint,  // here's the undefined behaviour
                puns[i].uint   // here's the undefined behaviour
        );
    }
    struct fullperms aFile = { readExecPun.perm, readExecPun.perm, nonePun.perm };
    union fullpun fullPun = { .perm=aFile };
    printf("sizeof(fullPun)==%lu\n", sizeof(fullPun));
    for (size_t i = 0 ; i < sizeof(struct fullperms); i++) {
        printf("%02x", fullPun.bytes[i]);
    }
    puts("");
    printf("^^^ well that is confusing\n");

    // Let's do this better
```

```
    struct betterunixperms {
        unsigned int ue:  1;
        unsigned int uw:  1;
        unsigned int ur:  1;
        unsigned int ge:  1;
        unsigned int gw:  1;
        unsigned int gr:  1;
        unsigned int ae:  1;
        unsigned int aw:  1;
        unsigned int ar:  1;
        unsigned int: 7; // pad
    };
    struct chmod {
        unsigned int u: 3;
        unsigned int g: 3;
        unsigned int a: 3;
        unsigned int: 7; // pad
    };
    union chmodb {
         struct chmod ch;
         struct betterunixperms bits;
         unsigned short ushort;
         unsigned int uint;
    };
    union chmodb ch = { .ch={7,1,0} };
    printf("sizeof(betterunixperms)==%lu\n",sizeof(struct betterunixperms));
    printf("sizeof(chmod)==%lu\n",sizeof(struct chmod));
    printf("sizeof(chmodb)==%lu\n",sizeof(union chmodb));
    printf("Better? %o %o %o %03o\n", ch.ch.u, ch.ch.g, ch.ch.a,
            ch.uint);
}


sizeof(struct unixperms)=4
Pun:         nonePun E:0 W:0 R:0 [    0] [00000000]
Pun:         readPun E:0 W:0 R:1 [    4] [00000004]
Pun:    readWritePun E:0 W:1 R:1 [    6] [00000006]
Pun:readWriteExecPun E:1 W:1 R:1 [    7] [00000007]
Pun:    readExecPun E:1 W:0 R:1 [    5] [00000005]
sizeof(fullPun)==12
```

```
050000000500000000000000
^^^ well that is confusing
sizeof(betterunixperms)==4
sizeof(chmod)==4
sizeof(chmodb)==4
Better? 7 1 0 017
```

### 1.5.16    Floating Point Numbers

Helpful resources:

- https://en.wikipedia.org/wiki/IEEE_754

- http://steve.hollasch.net/cgindex/coding/ieeefloat.html

Single precision IEEE754 floating point numbers
    Type: float
    -Sign: 0 - positive, 1 negative -Exponent: unsigned 8 bit integer with 127
$(2^7$-1) offset

- -126 to 127

- Fraction: 24 bits (only 23 used). The first bit is implied.

    - So 00000000000000000000000 is actually 100000000000000000000000

| Sign (1 bit) | Exponent (8 bit) | b10 | Fraction (23 bit) | Value | base 2 |
|---|---|---|---|---|---|
| 0 | 01111100 | -3 | 01000000000000000000000 | 0.15625 | $1/8 + 1/32$ |
| 1 | 01111100 | -3 | 01000000000000000000000 | -0.15625 | |
| 0 | 01111100 | -3 | 01000000000000000000000 | -0.15625 | |
| 0 | 00000000 | 0 | 00000000000000000000000 | 0 | |
| 1 | 00000000 | 0 | 00000000000000000000000 | -0 | |
| 0 | 11000000 | 65 | 00000000000000000000000 | 3.68e+19 | $2^{65}$ |
| 1 | 11111111 | 0 | 00000000000000000000000 | -inf | |
| 0 | 11111111 | 0 | 00000000000000000000000 | inf | |

- Neat design trick: you can sort them as signed integers (two's complement)!

- -0 is right before 0 for signed integer sorting

0   01111100   -3   01000000000000000000000   0.15625   $1/8 + 1/32$

45

0.15625 to floating point. Implied bit is pow(2,-3)
pow(2,-3) + 0*pow(2,-4) + 1*pow(2,-5) (pow(2,5)==32, pow(2,-5) = 1/32.0)

```c
#include <math.h>
#include <stdio.h>
#include <inttypes.h>
// 0x1L MUST be used 0x1 causes bugs
#define BIT(x,y) (x & (0x1L << y))
#define MAXBITSTRBITS 129
static char _bitstr[MAXBITSTRBITS] = { '\0' };
static char * bitString(uint64_t value, const unsigned int bits) {
    // iterator must be uint64_t
    for (uint64_t i = 0 ; i < bits; i++) {
        char bit = (BIT(value,i))?'1':'0';
        _bitstr[bits-1-i] = bit;
    }
    _bitstr[bits] = '\0';
    return _bitstr;
}


// NOT PORTABLE
struct float_t {
    unsigned int mantissa:23; // LOWEST
    unsigned int exponent:8;
    unsigned int sign:1; // HIGHEST
};
// type pun for fun!
union floatint {
    float f;
    uint32_t i;
    struct float_t t;
};


int main() {
    float floats[] = {
        0.0,
       -0.0,
        INFINITY,
```

```
        -INFINITY,
         0.00001,
         0.0001,
         0.001,
         0.01,
         0.1,
         1.0,
         1/64.0,
         1/32.0,
         1/16.0,
         1/8.0,
         1/4.0,
         1/2.0,
         1.0,
         2.0,
         4.0,
         128.0,
         65536.0,
         0.15625,
         36893488147419103232.0 // 2**65
};
size_t nfloats = sizeof(floats)/sizeof(float);
printf("We're printing floats!\n");
printf("sizeof(floatint) == %lu\n", sizeof(union floatint));
printf("sizeof(float_t) == %lu\n", sizeof(struct float_t));
for (size_t i = 0; i < nfloats; i++) {
    float f = floats[i];
    union floatint fi = {.f=f};
    printf("%12g %08x %s\n", f, fi.i, bitString(fi.i, 32));
    printf("%12g %08x sign: %hhu exponent: %hhu exp-127: %d mantissa: %08x\n",
            f,
            fi.i,
            fi.t.sign,
            fi.t.exponent,
            (int)fi.t.exponent - (int)127,
            fi.t.mantissa
    );
    /* // We don't need bitfields
    printf("%12g %08x sign: %hhu exponent: %hhu exp-127: %d mantissa: %08x\n",
            f,
```

```
                fi.i,
                (fi.i >> 31),
                ((fi.i << 1) >> 24),
                (int)((fi.i << 1) >> 24) - (int)127,
                (fi.i & 0x007FFFFF)
        );
        */


    }
    puts("");
}
```

```
We're printing floats!
sizeof(floatint) == 4
sizeof(float_t) == 4
          0 00000000 00000000000000000000000000000000
          0 00000000 sign: 0 exponent: 0 exp-127: -127 mantissa: 00000000
         -0 80000000 10000000000000000000000000000000
         -0 80000000 sign: 1 exponent: 0 exp-127: -127 mantissa: 00000000
        inf 7f800000 01111111100000000000000000000000
        inf 7f800000 sign: 0 exponent: 255 exp-127: 128 mantissa: 00000000
       -inf ff800000 11111111100000000000000000000000
       -inf ff800000 sign: 1 exponent: 255 exp-127: 128 mantissa: 00000000
      1e-05 3727c5ac 00110111001001111100010110101100
      1e-05 3727c5ac sign: 0 exponent: 110 exp-127: -17 mantissa: 0027c5ac
     0.0001 38d1b717 00111000110100011011011100010111
     0.0001 38d1b717 sign: 0 exponent: 113 exp-127: -14 mantissa: 0051b717
      0.001 3a83126f 00111010100000110001001001101111
      0.001 3a83126f sign: 0 exponent: 117 exp-127: -10 mantissa: 0003126f
       0.01 3c23d70a 00111100001000111101011100001010
       0.01 3c23d70a sign: 0 exponent: 120 exp-127: -7 mantissa: 0023d70a
        0.1 3dcccccd 00111101110011001100110011001101
        0.1 3dcccccd sign: 0 exponent: 123 exp-127: -4 mantissa: 004ccccd
          1 3f800000 00111111100000000000000000000000
          1 3f800000 sign: 0 exponent: 127 exp-127: 0 mantissa: 00000000
   0.015625 3c800000 00111100100000000000000000000000
   0.015625 3c800000 sign: 0 exponent: 121 exp-127: -6 mantissa: 00000000
    0.03125 3d000000 00111101000000000000000000000000
    0.03125 3d000000 sign: 0 exponent: 122 exp-127: -5 mantissa: 00000000
     0.0625 3d800000 00111101100000000000000000000000
```

```
   0.0625 3d800000 sign: 0 exponent: 123 exp-127: -4 mantissa: 00000000
    0.125 3e000000 00111110000000000000000000000000
    0.125 3e000000 sign: 0 exponent: 124 exp-127: -3 mantissa: 00000000
     0.25 3e800000 00111110100000000000000000000000
     0.25 3e800000 sign: 0 exponent: 125 exp-127: -2 mantissa: 00000000
      0.5 3f000000 00111111000000000000000000000000
      0.5 3f000000 sign: 0 exponent: 126 exp-127: -1 mantissa: 00000000
        1 3f800000 00111111100000000000000000000000
        1 3f800000 sign: 0 exponent: 127 exp-127: 0 mantissa: 00000000
        2 40000000 01000000000000000000000000000000
        2 40000000 sign: 0 exponent: 128 exp-127: 1 mantissa: 00000000
        4 40800000 01000000100000000000000000000000
        4 40800000 sign: 0 exponent: 129 exp-127: 2 mantissa: 00000000
      128 43000000 01000011000000000000000000000000
      128 43000000 sign: 0 exponent: 134 exp-127: 7 mantissa: 00000000
    65536 47800000 01000111100000000000000000000000
    65536 47800000 sign: 0 exponent: 143 exp-127: 16 mantissa: 00000000
  0.15625 3e200000 00111110001000000000000000000000
  0.15625 3e200000 sign: 0 exponent: 124 exp-127: -3 mantissa: 00200000
3.68935e+19 60000000 01100000000000000000000000000000
3.68935e+19 60000000 sign: 0 exponent: 192 exp-127: 65 mantissa: 00000000
```

1. How do I avoid typepunning?

   The C99 is union type punning :-/

   The "blessed" way that is C11 compatible is memcpy.

   Bitfields are generally considered CURSED due to platform specific
   behaviour.