

CMPUT201W20B2 Week 11

Abram Hindle

March 24, 2020

Contents

1	Week11	2
1.1	Copyright Statement	2
1.1.1	License	2
1.1.2	Hazel Code is licensed under AGPL3.0+	2
1.2	Alternative version	2
1.3	Init ORG-MODE	2
1.3.1	Org export	3
1.3.2	Org Template	3
1.4	Remember how to compile?	3
1.5	Numbers!	3
1.5.1	Binary	4
1.5.2	Octal	4
1.5.3	Base10 Review	4
1.5.4	Hex Review	5
1.5.5	Bitwise Operators	5
1.5.6	Left Shift	5
1.5.7	Right Shift	11
1.5.8	Could we access the bits with shifts?	16
1.5.9	bitwise and	18
1.5.10	bitwise OR	22
1.5.11	bitwise XOR	25
1.5.12	bitwise complement	28
1.5.13	Flags	29
1.5.14	PRACTICE and More resources	37
1.5.15	BitFields	38
1.5.16	Floating Point Numbers	38

1 Week11

1.1 Copyright Statement

If you are in CMPUT201 at UAlberta this code is released in the public domain to you.

Otherwise it is (c) 2020 Abram Hindle, Hazel Campbell AGPL3.0+

1.1.1 License

CMPUT 201 C Notes Copyright (C) 2020 Abram Hindle, Hazel Campbell

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

1.1.2 Hazel Code is licensed under AGPL3.0+

Hazel's code is also found here <https://github.com/hazelybell/examples/tree/C-2020-01>

Hazel code is licensed: The example code is licensed under the AGPL3+ license, unless otherwise noted.

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 $2 * \text{pow}(16,1)$
- 32 in base 10 is $30 * \text{pow}(10,1) + 2 * \text{pow}(10,0)$
- 32 in base 8 is $4 * \text{pow}(8,1)$

- 32 in base 2 is $1 \cdot \text{pow}(2,5)$
- 31 in base 16 is $1 \cdot \text{pow}(16,1) + 15 \cdot \text{pow}(16,0)$
- 31 in base 10 is $30 \cdot \text{pow}(10,1) + 1 \cdot \text{pow}(10,0)$
- 31 in base 8 is $3 \cdot \text{pow}(8,1) + 7 \cdot \text{pow}(8,0)$
- 31 in base 2 is $1 \cdot \text{pow}(2,4) + 1 \cdot \text{pow}(2,3) + 1 \cdot \text{pow}(2,2) + 1 \cdot \text{pow}(2,1) + 1 \cdot \text{pow}(2,0)$

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

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: $7 \cdot 8 + 7 = 63$ 0777: $7 \cdot 8 \cdot 8 + 7 \cdot 8 + 7 = 511$

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 = 15 = 15 0xFF = 15*16 + 15 = 255 0xFFF = 15*16*16 + 15*16 + 15 = 4095 0xFFFF = 15*16*16*16 + 15*16*16 + 15*16 + 15 = 65535

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

1.5.5 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 bitwise operations unless you are sure.

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

1.5.6 Left Shift

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

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

```
#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;
    uint16_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: %20zu %20zd\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 = 0xFFFFFFFFFFFFFFFF;
    uint32_t u32 = 0xFFFFFFFF;
    uint16_t u16 = 0xFFFF;
    int64_t i64 = 0xFFFFFFFFFFFFFFFF;
    int32_t i32 = 0xFFFFFFFF;
    uint16_t i16 = 0xFFFF;
    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
```

```
#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
```

```
#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

```

#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

```

#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 = 0 ; i < 64; i++) {
        putchar(BIT(interesting, i)?'1':'0');
    }
    puts("");
    for (size_t i = 0 ; i < 64; i++) {
        putchar('0' + i%10);
    }
    puts("");
}

```

```

Lets see some bits!
1111011110110011110101011001000111100110101000101100010010000000
0123456789012345678901234567890123456789012345678901234567890123

```

(b) Bit Printing

& is way safer than shift for bit printing.

```

#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,
        0xFFFFFFFFFFFFFFF,
    };
    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)
    );
}
}

```

```

#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 = ffffffff
ffffffff & 00000002 = ffffffff
ffffffff & 00000004 = ffffffff
ffffffff & 00000008 = ffffffff
ffffffff & 00000010 = ffffffff
ffffffff & 00000020 = ffffffff
ffffffff & 00000040 = ffffffff

```

```

ffffffff & 00000080 = ffffffff7f
ffffffff & 00000100 = fffffffeff
ffffffff & 00000200 = fffffffdff
ffffffff & 00000400 = fffffffbfff
ffffffff & 00000800 = ffffff7fff
ffffffff & 00001000 = fffffefff
ffffffff & 00002000 = fffffdfff
ffffffff & 00004000 = fffffbfff
ffffffff & 00008000 = fffff7fff
ffffffff & 00010000 = fffefffff
ffffffff & 00020000 = fffdfffff
ffffffff & 00040000 = fffbfffff
ffffffff & 00080000 = fff7fffff
ffffffff & 00100000 = ffefffff
ffffffff & 00200000 = ffdfffff
ffffffff & 00400000 = ffbfffff
ffffffff & 00800000 = ff7fffff
ffffffff & 01000000 = fefffff
ffffffff & 02000000 = fdfffff
ffffffff & 04000000 = fbfffff
ffffffff & 08000000 = f7fffff
ffffffff & 10000000 = efffff
ffffffff & 20000000 = dfffff
ffffffff & 40000000 = bfffff
ffffffff & 80000000 = 7fffff
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 .

$$a \wedge b \wedge a = b$$

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:

- `ORDONLY` - Read the file
- `OWRONLY` - Write to the file
- `ORDWR` - Read and Write to the file
- `OCREAT` - CREATE the file if it doesn't exist
- `OAPPEND` - 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!

`OWRONLY | OCREAT` 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-0thHf0mgNZXnTS0vyx5lSduQwsbHYMmVPLtT1/edit#

#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.\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) {
    // 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 spatarn.\n");
    }
}

```



```

int main() {
    printf("OK build some pizza using |\n\n");
    printPizza(CHEESE | CORN);
    printPizza(CHEESE | THINCRUST | BLACKMETALSAUCE | CORN | MYSTERYMEAT);
    printf("\n\nOK build some pizza via iteration\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");
    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);
}

```

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, .
 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 (srsly?), .
 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?), onio
 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?), on
 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: SPAAAAAAAAARTA, or spatarn.

1.5.14 PRACTICE and More resources

- PLEASE Read Hazel's Notes on the subject
 - <https://docs.google.com/document/d/1S-I-0thHf0mgNZXnTS0vyx5lSduQwsbHYMmVPLtT1edit#>
 - 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

1.5.16 Floating Point Numbers