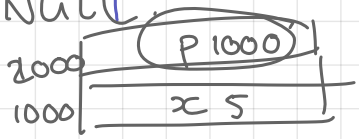
Week 1

Question 9.

int x ; // variable at address 1000 with initial value 0.
int *p; // variable at address 2000 with initial value 0.

	statement	x value	p value ("p points to")	x address	p address ("address of")
initial.	-	0	0	1000	2000
a.	p = &x;	0	1000	1000	2000
b.	x = 5;	5	1000	1000	2000
c.	*p = 3;	3	1000	1000	2000
d	x = (int) p; (typecast)	1000	1000	1000	2000
e.	x = (int) &p;	2000	1000	1000	2000
f.	p = NULL;	?2000	NULL	1000	2000
g.	*p = 1;	fail because &p = 2000	p is NULL		



Question 6.

when to use * and/or malloc for structs?

struct node a;

See code q6.

struct node *b;

what does malloc do?

Struct can hold different variable types	array only one type
fields are accessed by names	fields are accessed by indexes

Week 2

int → any # of bytes (at least 2 bytes)

1. When should the types in **stdint.h** be used?

=> what is the type **uint8_t**? (unsigned, 8 bits)
unsigned int that is 8 bits

how is it different to **int8_t**?
↑ not unsigned

2. How are the bases [decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2)] denoted in C?

- hexadecimal: starts with 0x
- octal: starts with 0
- decimal: any other case
- binary: → 0b → not standard → don't use !!

	decimal	binary	octal	hexadecimal
a.	1	0 0 0 0 0 0 0 1	0 0 1	0 1
b.	8	0 0 0 0 1 0 0 0	0 1 0	0 8
c.	10			
d.	15	0 0 0 0 1 1 1 1 ($1\cdot2^0 + 1\cdot2^2 + 1\cdot2^1 + 1\cdot2^0$)	0 1 7	0 F
e.	16			
f.	100			
g.	127			
h.	200			

15
7 r 1
3 r 1
1 r 1
0 r 1

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

decimal → binary (by hand)

(÷2) 8
4 r 0
2 r 0
1 r 0
0 r 1

$8_{10} = 001000_2 = 8_{16} = 10_8$

$_{10}$ means base 10

1

$_2$ means base 2

binary → hex

4 digits of binary are equal to 1 digit in hex. (hexadecimal)

1 1 1 1
$2^3+2^2+2^1+2^0$ (bin ⇒ dec)
15 = F

1 0 1 1
$2^3\times1+2^2\times0+2^1\times1+2^0\times1$

binary → octal

same as bin → hex but 3 digits

$111_2 = 7$

3. Bitwise Operations

	A	B	OR |	AND &	XOR ^
1.	0	0	0	0	0
2.	0	1	1	0	1
3.	1	0	1	0	1
4.	1	1	1	1	0
	True = 1 False = 0		at least one is 1	need both to be 1	only one can be 1

A	NOT ~
0	1
1	0
	invert the input

a) $0x5555 \mid 0xAAAA$

0101 0101 0101 0101
1010 1010 1010 1010
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

$0xFFFF$

b) $0x5555 \,\&\, 0xAAAA$

0101 0101 0101 0101
1010 1010 1010 1010
&
0 0 0 0

$0x0000$

c) $0x5555 \,\hat{}\, 0xAAAA$

0101 0101 0101 0101
1010 1010 1010 1010
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

$0xFFFF$

d) $0x5555 \,\&\, \sim 0xAAAA$

0101 0101 0101 0101
AAAA: 1010 1010 1010 1010 &
$\sim$AAAA = 0101 0101 0101 0101
0101 0101 0101 0 0 0 0

$0x5555$

e) $0x0001 << 6$

bitwise shift (to) left

0000 0000 0000 0001
00 0000 0001 00 0000

pad w/ 0's

f) $0x5555 >> 4$ (shift right)

always use unsigned vars!!

0101 0101 0101 0101
0000 0101 0101 0101

g) $0x5555 \,\&\, (0xAAAA << 1)$

h) $0xAAAA \mid 0x0001$

i) $0xAAAA \,\&\, \sim 0x0001$

1000
2^3

Given a variable **X**...

Copy/extract the value of a specific bit: X & mask

Set a specific bit to 1: X | mask

Set a specific bit to 0: X & ~mask.

invert a specific bit: X ^ mask

set a specific bit to 1.

	0011	0000
mask:	0000	0100
want:	0011	0100
|:	0011	0100

want to set the original value to 1 @ this digit

^ → good

	0011	0100
	0000	0100
^	0011	0000

00000 0 . 1 0 0 0 1

1000 0000 0

00000 = 0

1000

i = 31

extract = 31

setting = 31 - 31 = 0

i = 28

extract = 28

setting = 31 - 28 = 3

is a^b == a & ~b ?

a	b	~b	a^b	a & ~b
0	0	1	0	0
0	1	0	1	0
1	0	1	1	1
1	1	0	0	0

(2)

no, they're different (2)

set a specific bit to 0.

a	orig	0110	1000
b	mask	0010	0001
		0100	1000
	~b	1101	1110
	a&(~b)	0100	1000

copy specific bit:

orig	0110	1010
mask	0011	0000
want:	0010	0000
try &	0010	0000

Week 3

!! weekly tests start this week !!

Negative Values

How are signed values represented?

X xxx xxxx xxxx xxxx

0 = positive
1 = negative

representing a number with 2's complement

- for positive values:
  we don't use 2's comp.
  same representation (normal binary)
- for negative values:
  use 2's complement

for a int8_t variable

Eg. ^ 5_{10}:

0000 0101 (21)

vs -5_{10}:

Use 2's complement: invert all binary digits and then add 1.

1111 1010 + 1 = 1111 1011

↳ signifying that this is negative.

100_{10}:

0110 0100

vs -100_{10}:

invert 1001 1011

add 1: 1001 1100

-28: 28_{10} = 0001 1100$_2$ (invert; find 1st 1 from RHS; keep the same)

$\therefore -28_{10}$ = 1110 0100$_2$

convert 16-bit representation to the corresponding decimal value:

- 0x0013

0000 0000 0001 0011

+ve

∴ read as normal

$2^4 + 2^1 + 2^0 = 19_{10}$

- 0xffff ⇒ f = 1111

= 1111 1111 1111 1111

-ve

two's complement.

= −1

in order to get the magnitude, apply 2's comp. again:

0000 0000 0000 0001 = 1

0x 800f

$= 2^3$

= 1000 0000 0000 1111

↳ ∴ negative

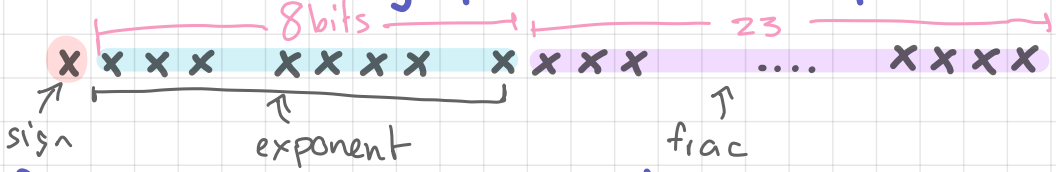
∴ we need 2's complement.

⇒ 0111 1111 1111 0001

= -32753

Floating Point Values

How are floating point values represented? (IEE 754) — 32 bits



from this we can calculate the value with the formula:

$$\text{sign} \times (1 + \text{frac}) \times 2^{\text{exp}-127}$$

note: we don't use two's complement here.

Eg: Convert the following to decimal numbers

convert like normal int (ignore signs)

$2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0 \quad 2^{-1}\ 2^{-2}\ 2^{-3}\ \ldots$
$0\ 1\ 0\ 1\ 1\ 0 \;.\; 0\ 1\ 1\ 0$

f) 0 | 1 0 0 0 0 0 0 0 | 0 1 1 0 ... 0 (all 0's)

- **sign:** = 1 (∵ positive)
- **exp:** = 2^7 = 128
- **frac:** = $2^{-2} + 2^{-3}$ = 0.375
- **value** = $1 \times (1 + 0.375) \times 2^{128-127}$ = 2.75

c) 0 | 0 1 1 1 1 1 1 1 | 1 0 0 ... 0

$2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0 \quad 2^{-1}$

- **sign:** 1
- **exp:** 127 ↪ $10000000 - 1$, $2^7 - 1$
- **frac:** 0.5
- **value** = $1 \times (1 + 0.5) \times 2^{127-127}$ = 1.5

a) 0 | 0 0 0 0 0 0 0 0 | 0 0 0 ... 0

- **sign:** 1
- **exp:** 0
- **frac:** 0
- **value** = $1 \times (1 + 0) \times 2^{0-127}$ = $2^{-127} \simeq 0$

b) 1 | 0 0 0 0 0 0 0 0 | 0 0 0 ... 0

denormal

- **sign:** −1
- **exp:** 0
- **frac:** 0
- **value** = $-1 \times (1 + 0) \times 2^{0-127}$ = $-2^{-127} \simeq -0$

in C we're looking mostly @ **float** not double

Eg 2: Convert the following to IEEE 754 encoded bit strings:

a) $2.5 \xrightarrow{\times 2} 1.25 \xrightarrow{\times 2} \Leftrightarrow 2.5 = 1.25 \times 2^1$

sign: 1x (bit = 0 to show +ve)

exp: $1 = exp - 127 \Rightarrow exp = 128$

frac: 0.25_{10}

bits = 0 1000 0000 01000000000...0

first express the number in the form:

$\pm (1 + frac) \times 2^{(n)}$

$n = exp - 127$

b) $0.375 \times 2^2 = 1.5 \Rightarrow 0.375 = 1.5 \times 2^{-2}$

sign: bit = 0

exp: $-2 = exp - 127 \Rightarrow exp = 125$

frac: $0.375_{10} = 0.011_2$

bits = 0 0111 1101 0110 0

c) 27.0

sign:

exp:

frac:

bits =

next page.

d) 100

sign:

exp:

frac:

bits =

$\div 2$:

128
64 0
32 0
16 0
8 0
4 0
2 0
1 0
0 1

1000 0000

to convert frac to binary

(x2) 0.25
0.5
1.0
.0

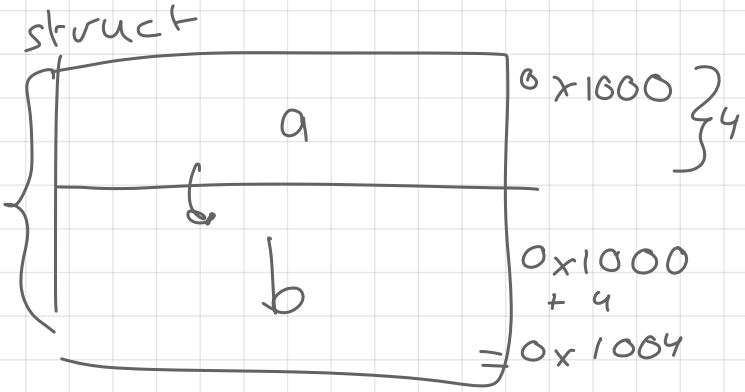
Structs vs Unions

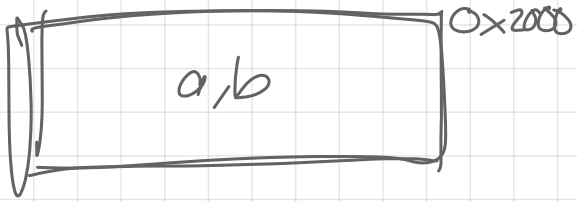
```
struct {
    int a;
    float b;
} x1;
```

&x1 = 0x1000
&(x1.a) = 0x1000
&(x1.b) = 0x1004

```
union {
    int a;
    float b;
} x2;
```

&x2 = 0x2000
&(x2.a) = 0x2000
&(x2.b) = 0x2000





0.375_{10}

(x2) 0.375
0.75
1.5
1.0

$.011$

$27 \div 2^4 = 1.6875$
$\Rightarrow 27 = (1 + 0.6875) \times 2^{(4)}$
$n = 4$ ↳ frac

c) 27.0

sign: bit = 0 ∵ positive

exp: $n = exp - 127$; $n = 4$ ∴ $4 = exp - 127 \Rightarrow exp\ 131_{10} = 1000\ 0011_2$

frac: 0.6875

bits = 0 1000 0011 10110....0

(x2) 0.6875
1.375
0.75
1.5
1.0