Last class ← bin to decimal.

Converting decimal to bin :-

let $d = a_{n-1}2^{n-1} + \dots + 2a_1 + a_0$

$\Rightarrow d\%2 = a_0$

and $\underbrace{\text{round}(d/2)}_{\text{same as C0 division}} = a_{n-1}2^{n-2} + \dots + a_1$

Algorithm :-

input : d

output :- binary representation of d in reverse

① if d = 0, then output 0 & exit

② while (d > 0) {

output (d%2);

d ← d/2;

}

Exercise :- use an array to collect the bits.

Hexadecimal :

Base 16 ; has 16 symbols to represent 0 to 15

Hex digits are $\{0, 1, 2, \ldots, 9\} \cup \{A, B, C, D, E, F\}$.

1 hex digits = 4 bits.

base \ number							
10	0	1	2	3	4	5	6
16	0	1	2	3	4	5	6
2	0000	0001	0010	0011	0100	0101	0110
10	7	8	9	10	11	12	13	14	15
16	7	8	9	A	B	C	D	E	F
2	0111	1000	1001	1010	1011	1100	1101	1110	1111

$(CODE)_{16} = (1100\ 0000\ 1101\ 1110)_2$

C O D E

In Co: we can directly enter numbers in hex by prefixing with 0x.

coni will convert a hex to dec and then display it

function :- int2hex
library :- util #use <util>
input :- int
output :- hex conversion output as a string.

Fixed size Number Representation

Machine words :-

saying that a machine is 32 bit or 64 bit means that it can process those many bits in parallel.

In C++ int are 32 bits long.

In computer memory $(1)_{10}$ is stored as

00 ... 00 1

31 times

Using 32 bits we can represent 2^{32} distinct nos.

When some mathematical operation results in nos. bigger than 32 bits then an overflow occurs.

Edit 11. How many distinct nos. are possible with 32 bits?.

We have 32 places which can be filled with 2 choices. This problem is the same as filling 32 labelled boxes with one of the two choices 0 or 1. This is equal to $\underbrace{2*2*2\ldots*2}_{32 \text{ times}} = 2^{32}$

Assume int has only 2 bits.

0	00
1	01
2	10
3	11

$2^2 = 4$

$(1)_{int2} + (2)_{int2}$

$= (01 + 10)_{int2}$

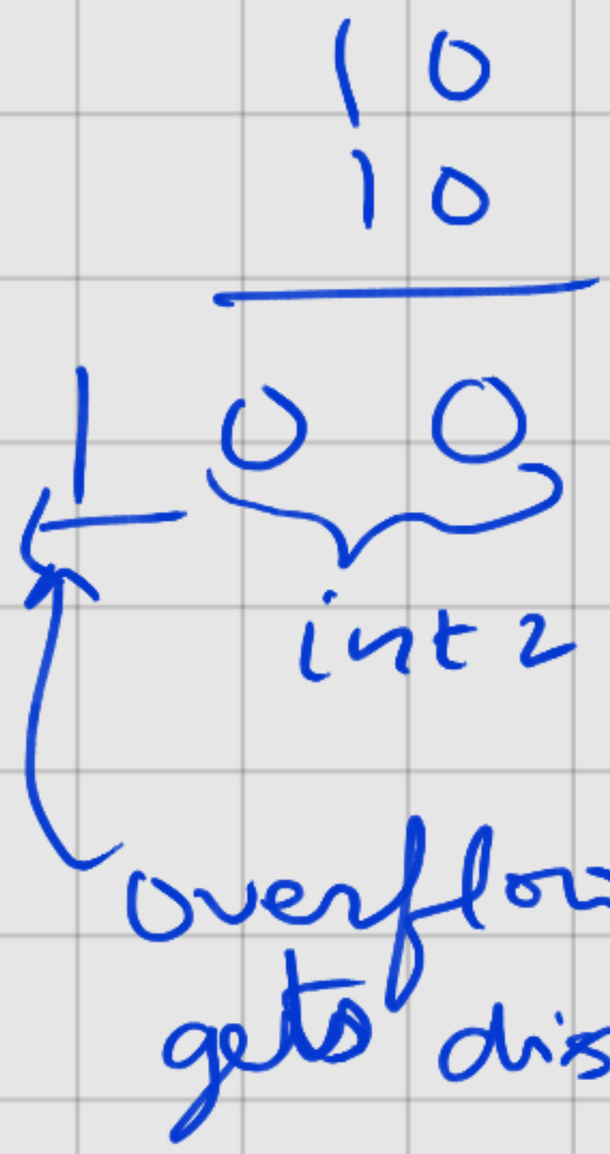
$= (11)_{int2}$

$= 3 \leftarrow$ output

$(2)_{int2} + (2)_{int2} = (10)_{int2} + (10)_{int2}$

$= (00)_{int2}$

$= 0 \leftarrow$ output.



0 ← output.

This same as working mod $4 \equiv$ mod 2^2 ← bits

$$(a+b)-b = a$$

overflows

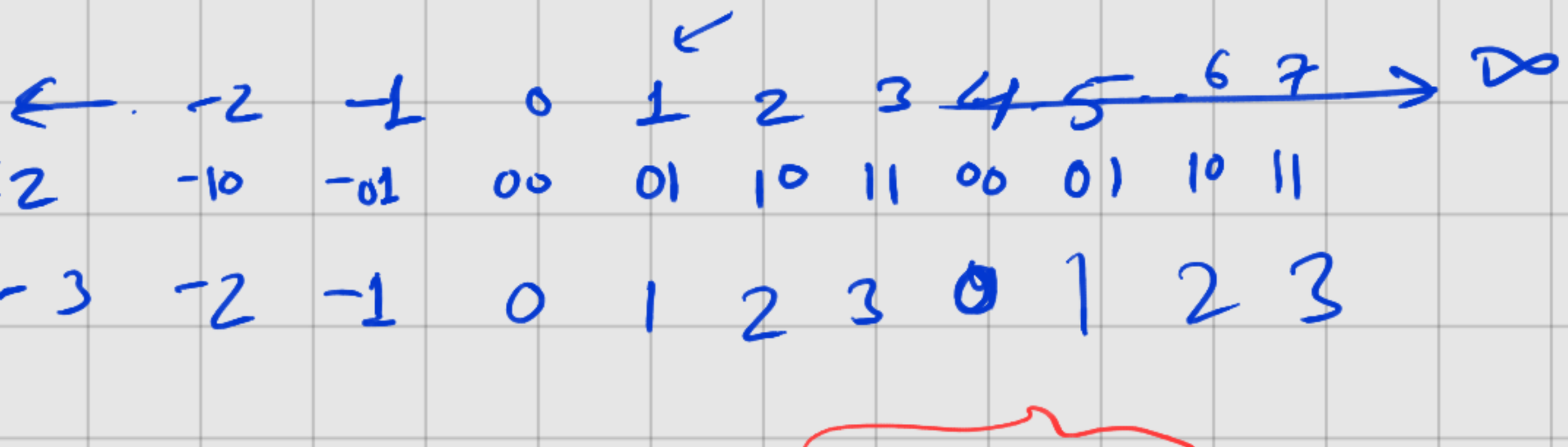
↓ mathematically

↓ Co int

$$((a+b)-b) \bmod 2^{32}$$

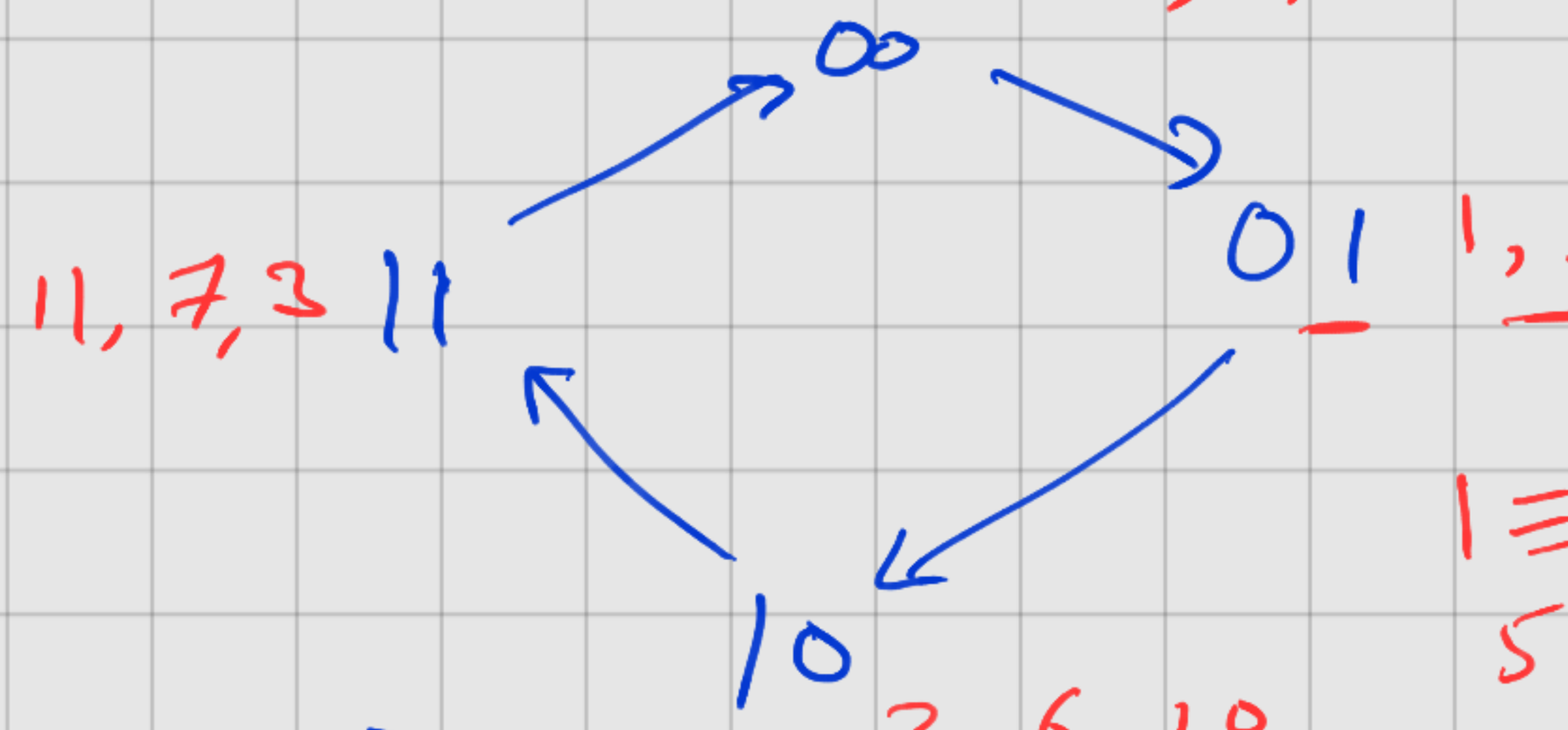
$$= a \bmod 2^{32}.$$

integer line :-



int	2	-10	-01	00	01	10	11	00	01	10	11
0	-3	-2	-1	0	1	2	3	0	1	2	3

0, 4, 8, 12



$1 \equiv 5 \bmod 4$

$5 \equiv 1 \bmod 4$

go clockwise when adding

go counterclockwise when subtracting.

$a \equiv b \bmod n$

$\downarrow$

$(a-b)$ is divisible by n

$a \% n = b \% n$

Laws of modular Arithmetic :-

$a = b \bmod n \equiv a =_n b$

Commutativity :-

$$x + y =_n y + x$$

associativity :-

$$(x+y)+z =_n x+(y+z)$$

additive unit :-

$$x + 0 =_n x$$

} +

Commutativity :

$$x * y =_n y * x$$

Assoc.

$$(xy)z =_n x(yz)$$

multiplicative unit :-

$$x * 1 =_n x$$

Distributivity $x^*(y+z) =_n x^*y + x^*z$

Annihilation: $x * 0 =_n 0$

In essence modular arithmetic behaves like "normal" arithmetic.

```
int z = 1+2;
int y = 2+1;
if ( z == y )
```

1+2 = 2+1

← evaluate to true in modular arithmetic

0 true

-ve

0 0

0 1

1, -3

-1, 3

11

10

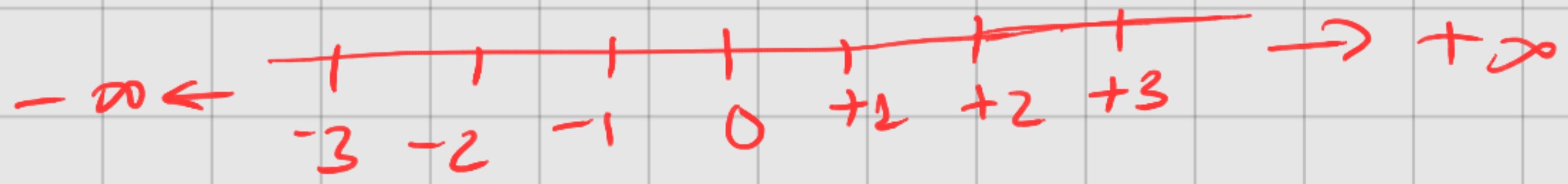
-2, 2

1 - (-3) = 4

$a \equiv b \bmod n$ if $n \mid (a-b)$

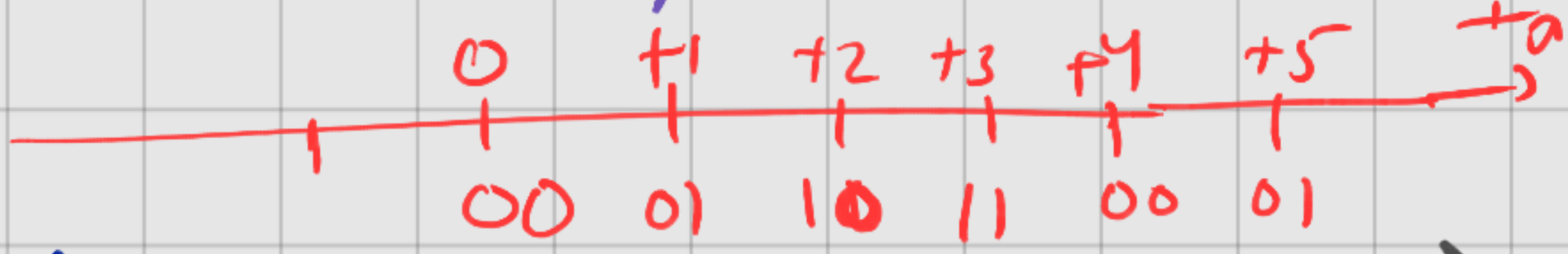
How to represent negative nos in a word having k-bits?

on the integer line going forward by 1 unit add 1 to the current no.



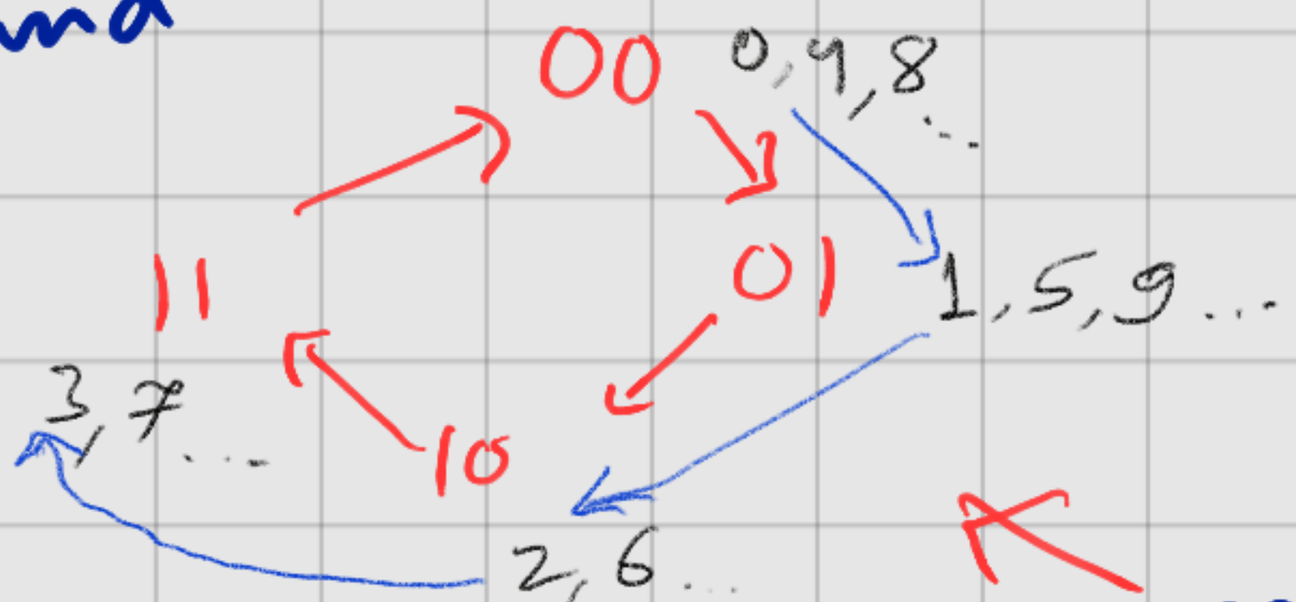
when numbers are represented using only a fixed no. of bits they start repeating:

for 2-bits.



Keep only the last two bits

wrap the number line around



= modular circle

It seems this can represent only the +ve nos.

Lets use intuition from the number line.

going forward on number line ≡ going clockwise on the modular circle.

going backward on number line ≡ going counterclockwise on the modular circle.

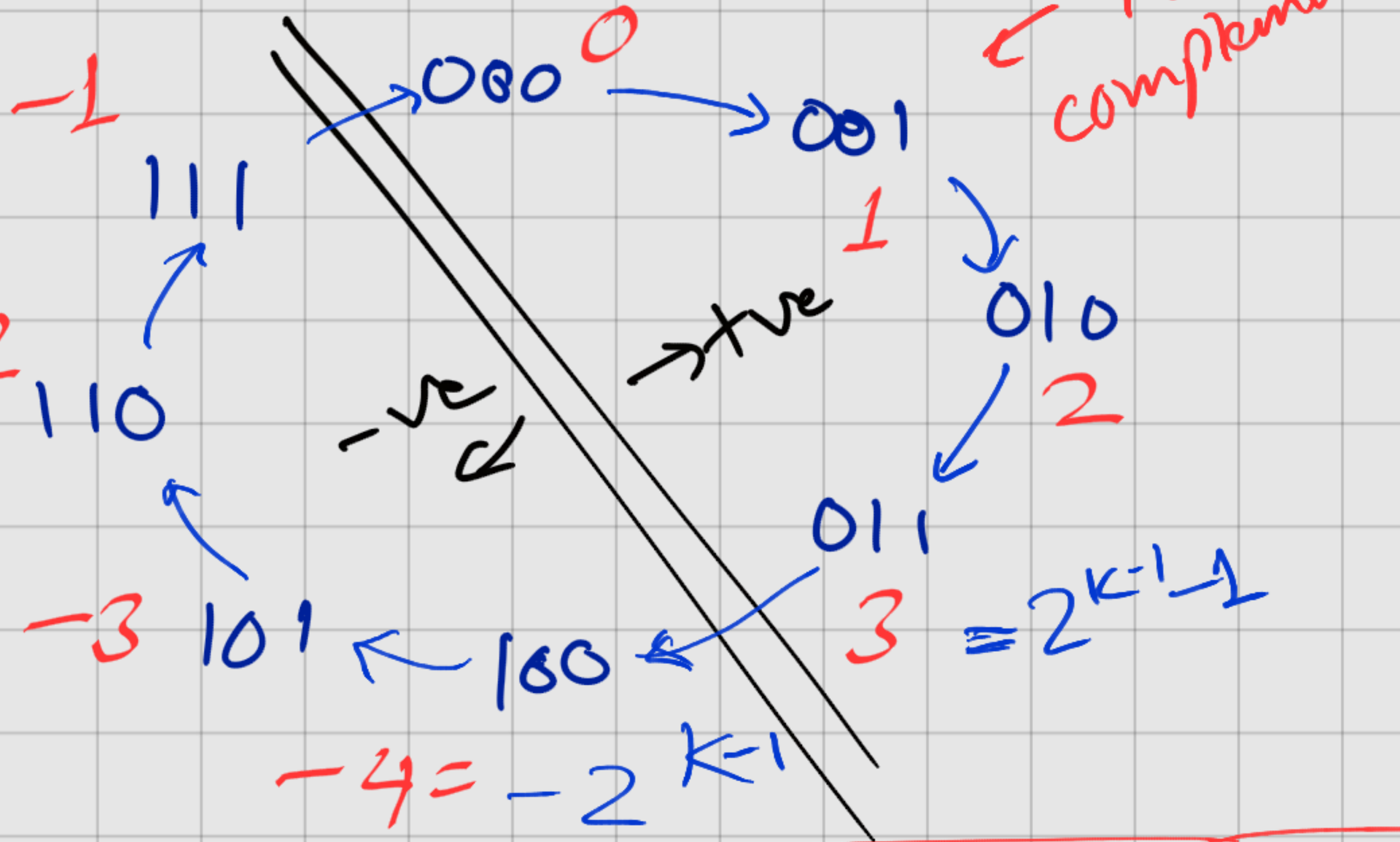
/backward

going left from zero on number line gives us negative numbers

going ↺ counterclockwise gives negative numbers

$k = 3 \quad \therefore 2^k = 2^3 = 8$

3 bits.



← Two's complement

$-4 = -2^{k-1}$

$3 = 2^{k-1} - 1$

for words of 3 bits :

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

So we ended up representing -ve numbers without a negative sign.

Lets check using our table

$(-1)_{int2} + (-3)_{int2}$

$= (111)_{int2} + (101)_{int2}$ } overflow

$= (1100)_{int2}$

$= (100)_{int2}$

$= (-4)_{int2}$

↳ $(-1) + (-3) = -4$ loobs fine here

This is how nos. are represented in C.

int in C0 has k=32 bits

its range is

-2^{31} to $2^{31}-1$

-2^{k-1} to $2^{k-1}-1$

// may overflow but works fine.

$(n+2n) - n \equiv_{int} n$

$n + (n-n)$

no overflow

Division & Modulus :-

In "normal" arithmetic

$$x/y = z$$

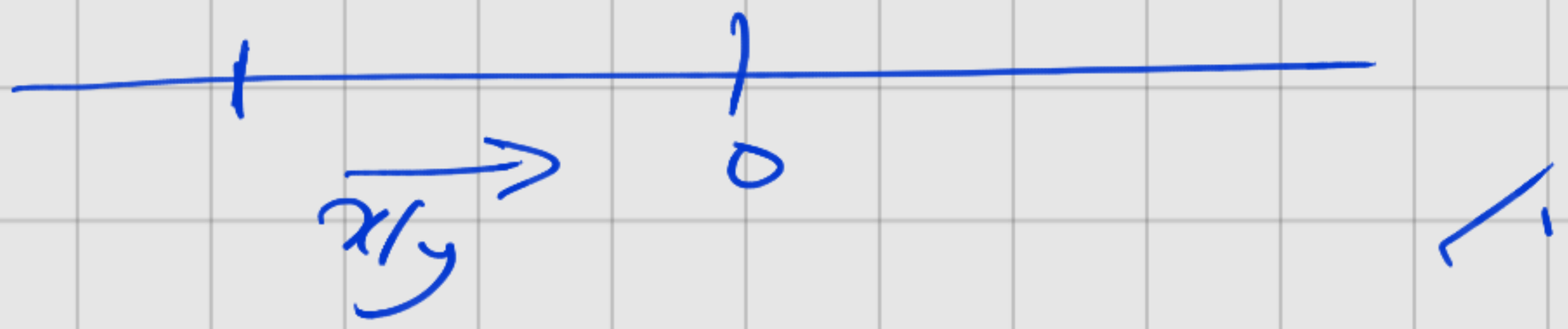
$$y * z = x$$

no longer true in C because of the way division behaves in C.

$$= \text{in C } (x/y) * y + x \% y = x$$

in C x/y gets rounded towards 0.

-ve +ve.



Safety Requirements:-

1. Division by 0 is undefined.
→ whenever you have x/y in the prog. make sure that $y != 0$.

2. x/y & x%y have preconditions.

```
//@ requires y != 0;
//@ requires !(x == int_min()
                && y == -1);
```

int_min() for C_0 $= -2^{31}$.

For more operation on int :-

Bitwise operations :-

& 0 1

and
0	0	0
1	0	1

| pipe symbol

or ! 0 1

0	0	1
1	1	1

xor
^	0	1
0	0	1
1	1	0

not ~
0	1
1	0

2^2 =2> 10

2->10^
2->10

00

= 0

^ ≡ not the pow operation.

~2 = ~(2)int32.

bitwise operator take
bits (ints) and output
bits (ints) { output is not
bool }

Arrays in C

Memory model in C :-

```
int f(int x){
    x = 10;
    return 1;
}
int g(int){
    int x = 0;
    int y = f(x);
    return x;
}
```

On making a call to g( ):

local memory.

frame g x = 0
         y = 1

pow(x,y)

pow(x,y-1)

if y==0
return 1

Variable, function calls are stored in local memory.

Arrays:-

bool [] b =
alloc_array(bool, 32);

-> int [] A = alloc_array(int, 32);

A is 0xCOFFEE (int[] with 32 elements)

->

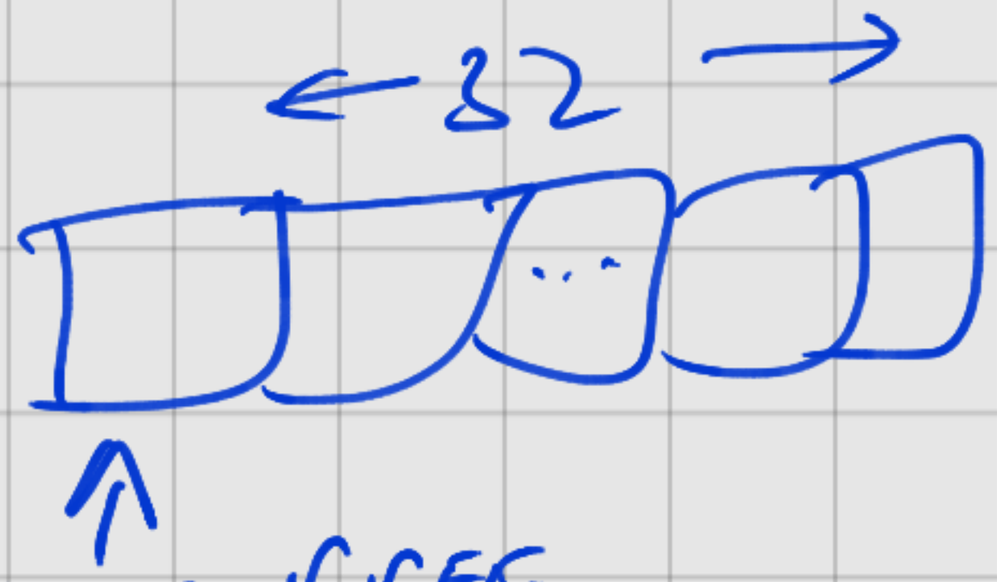
memory address -

local memory (vars) (function frames)

memory addresses

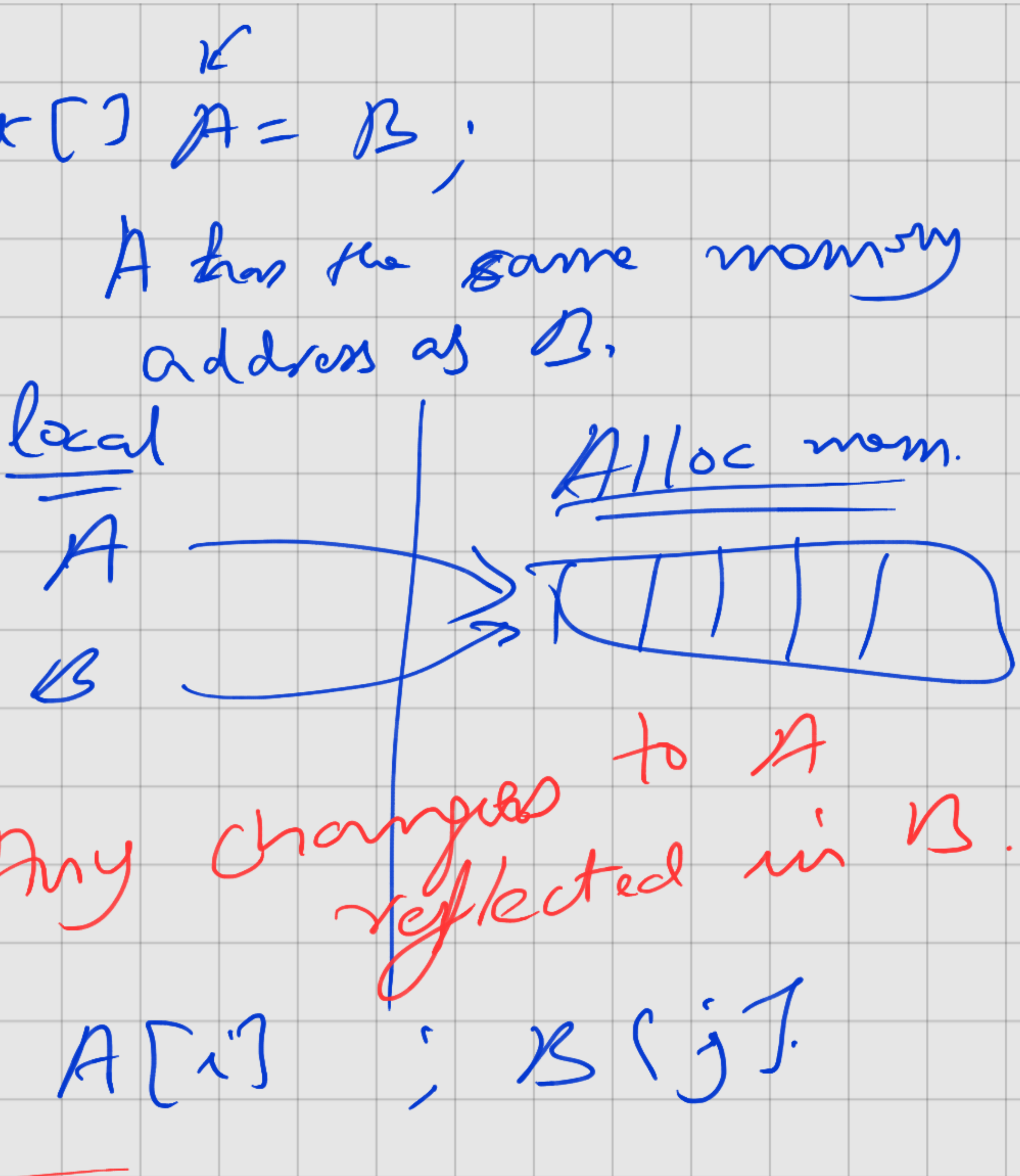
A = 0xCOFFEE

Allocated memory.



int [] A = B;

A has the same memory address as B.



Any changes reflected in B.

A[i] ; B[j].

int [] A = allocarray(int, n);

initially elements of A are zero.

the only valid indices

$0 \leq$ index < 32

preconditions for using arrays :-

prec for index.

A[i]

// @ requires $0 <= i$ &&

$i <$ "length of A".

alloc_array (type, n)

// @ requires $n >= 0$;

always use

coin - d