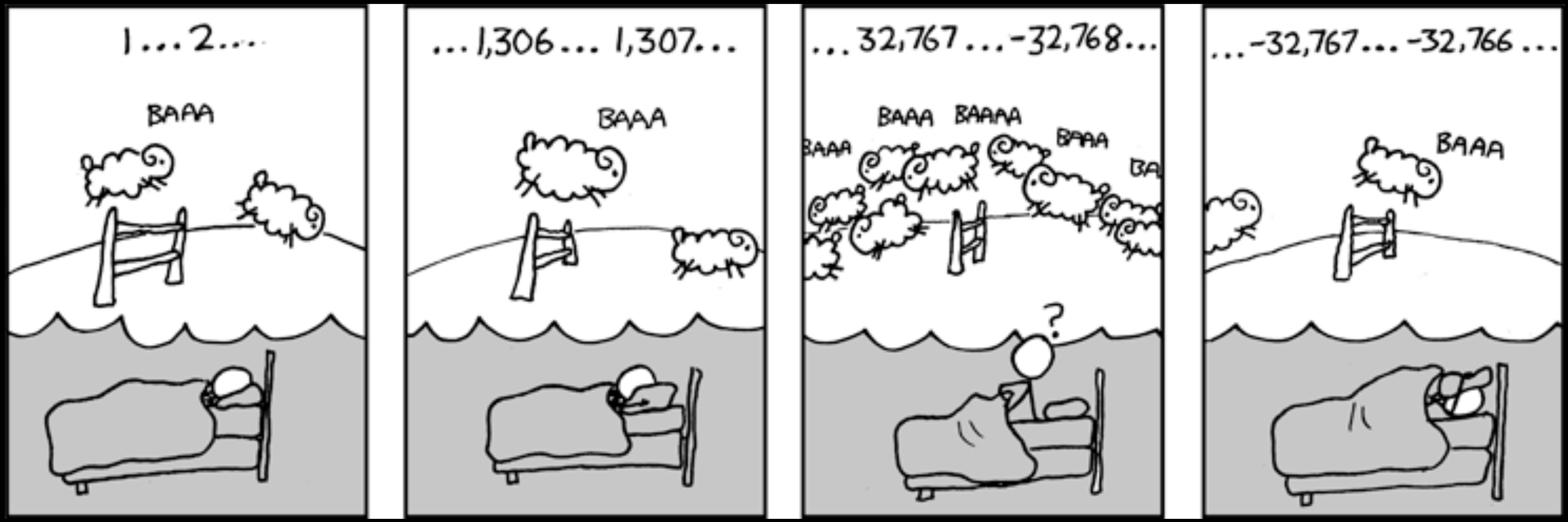
Week 2

- Data types and Input/Output
- Basic Arithmetic and Flow Control



What do we want to do?

- We want to give C numbers and do calculations on them
- Check whether a number is prime
- Check the price per square cm for pizza
- Calculate mortgage rate
- Find out how high your blood alcohol level is after some beers
- Find out how many seconds you have to work to buy a beer

What do we need?

- Some way to store numbers
- Do some quick maths on them
- Print the result

Storing numbers in C

- Different datatypes for different uses
- You want to store an Integer or Real Number?
- C knows three fundamental types:
  Integer (0,255, -24, 8, …)
  Float (2.5, 3.9, -215.6,…)
  Character (‚a‘, ‚L‘, ‚;‘, ‚m’)

To be precise, C knows integers, floats and complex numbers.

More on that topic later

What does an Integer represent?

- A Number from: $\mathbb{Z}$ or $\mathbb{N}_0$
- What range?
- How is it stored in memory?

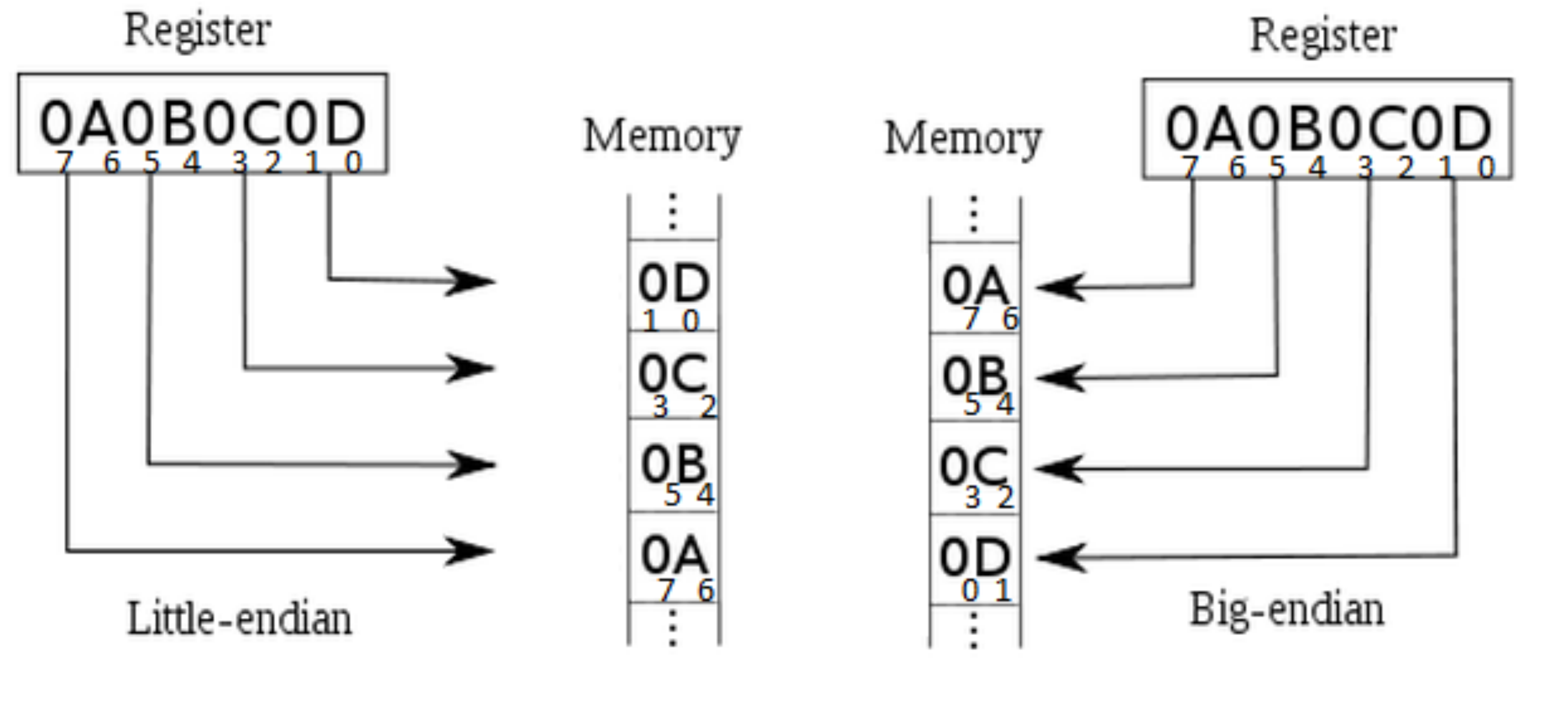
Which numbers?

- Use signed or unsigned to specify whether it is positive only or positive and negative

What range?

- The size of an int is specified as follows: At least 2 bytes
- Usually, but not always 4 bytes
  - 2 Bytes: -32,768 to 32,767
    - (-2^15 to 2^15 -1)
  - 4 Bytes-2,147,483,648 to 2,147,483,647
    - (-2^31 to 2^31 -1)

How is it aligned in memory?



Endian-ness

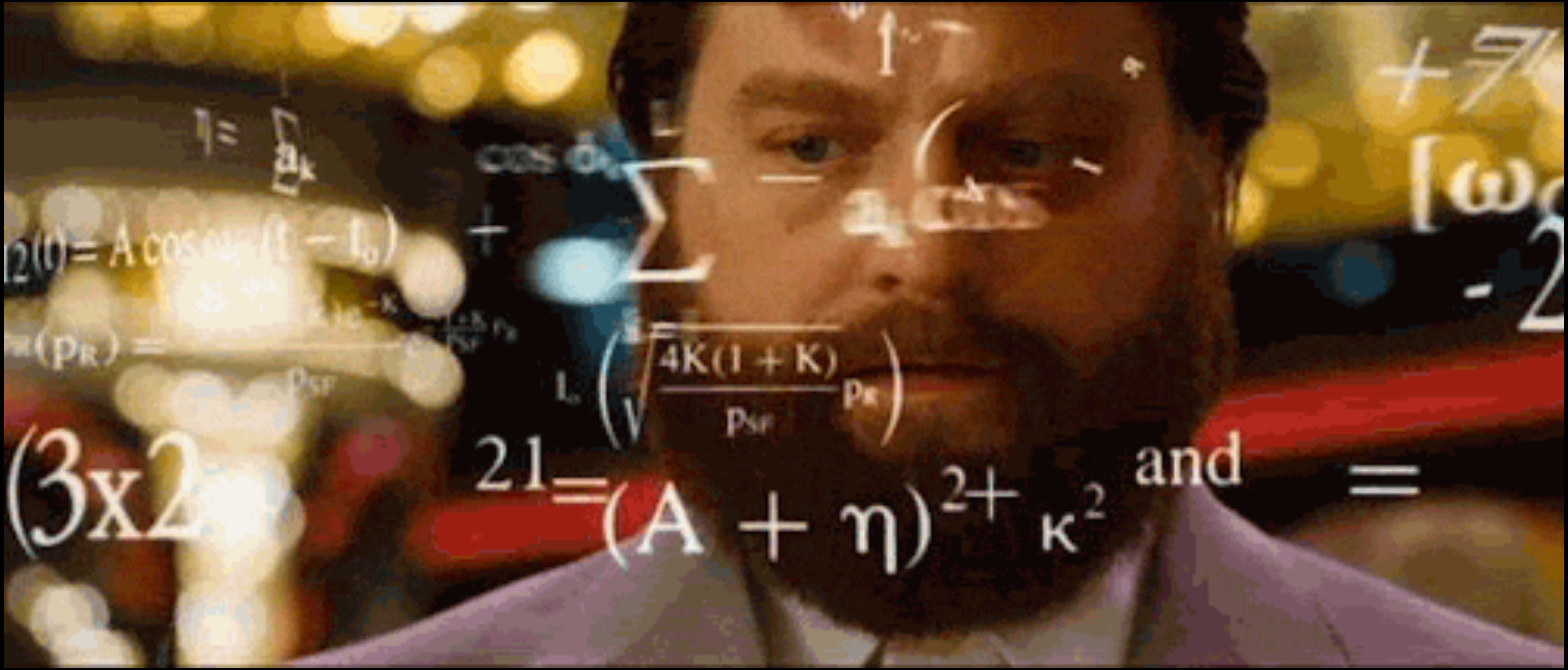
Potato or Potato?

- People can’t agree if a number should start with the highest or lowest byte
- Memory: |12|34|56|78
- Big-Endian people: “it’s clearly 0x12345678”
- Little-Endian people: “Fools! It’s clearly 0x78563412!”
- You must know what endianness is used, perhaps you must re-interpret some numbers..

What if I need other sizes?

- short: at least 2 bytes
- long: at least 4 bytes
- long and short can be used before types

- E.g. if you need a unsigned int with at least 8 bytes of memory: simply use a unsigned long long int


cos
Σ
+
A cos
(p_R) =
p_SF
4K(1 + K)
p_SF
p_R
(3x2
21
= (A + η)^2+ κ^2
and
=
[ω
-2

How to deal with other platforms?

Careful with int

- Int is fine for most use cases
- Int is guaranteed to hold at least 2 Bytes
- If your calculations assume 4 Bytes, your code is wrong on some systems

Rule 1:

Friends don’t let friends use standard integer data types for all problems

What do other languages do?

- Python: arbitrary length integer numbers (literally)
- Java: Has a virtual machine and defined sizes for types
- C#: Same as Java
- Rust: u32,i32,u8,i16 or usize/isize
- Go: Same as Rust
- Haskell: Same as Python
- Javascript: “Number” for all number types

Introducing:

<stdint.h>

<stdint.h>

- Header file
- Gives you access to types like
  uint8_t
  int16_t
  uint64_t
- It won’t matter where your code runs, it will always have the same size
- But may be slower than int

Example!

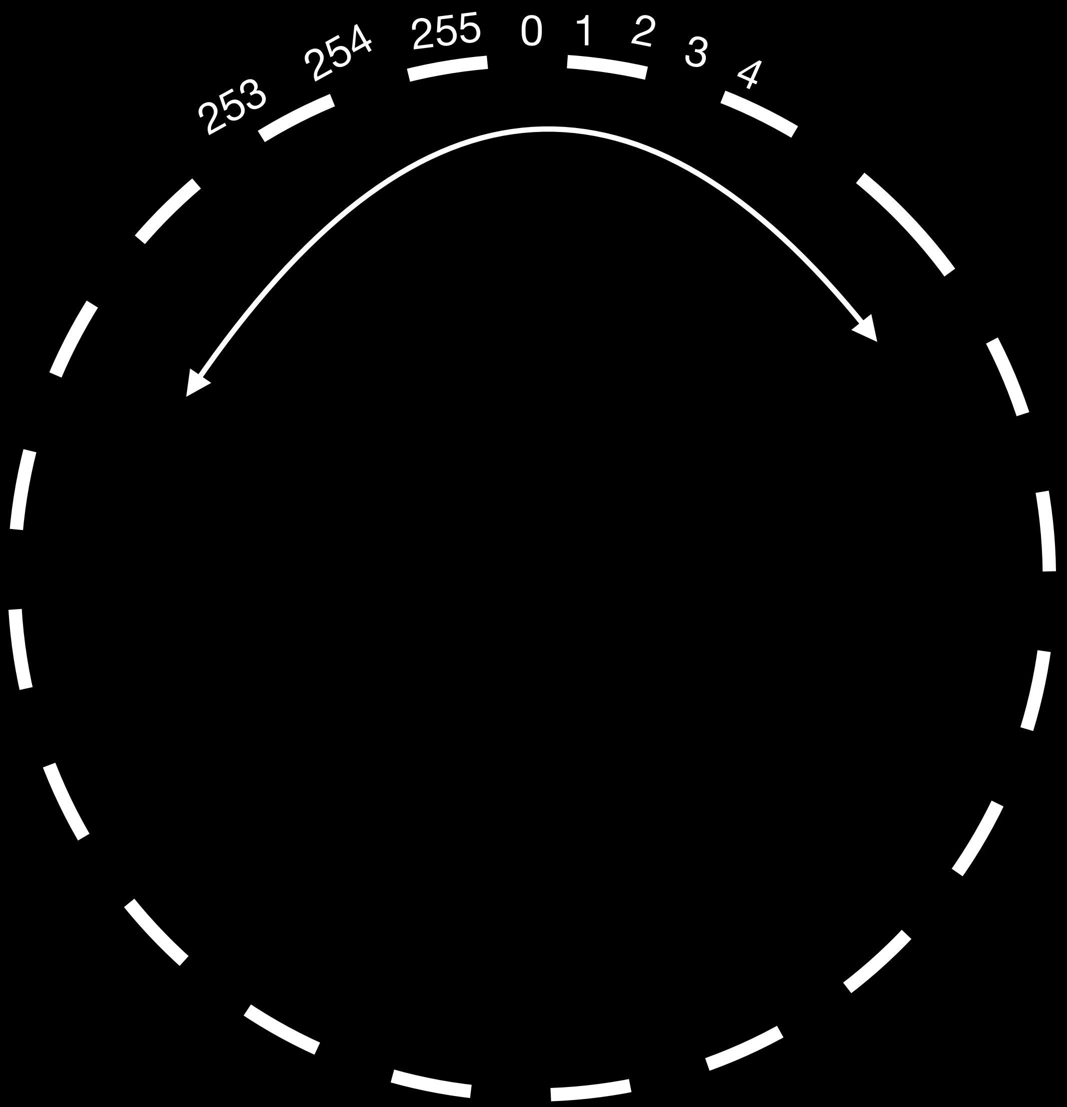
```c
#include <stdint.h>
#include <stdio.h>

int main(){
    int8_t n_12 = 12;
    int8_t n_250 = 120;
    int8_t n_tooMuch = n_12 + n_250;
    printf("%u\n", n_tooMuch);
    return 0;
}
```

Where is undefined behavior?

Overflow/Underflow

- Some overflows/underflows on specific types have defined behaviours, others are undefined
- Int types go from ~ -x to x, what if -x-1 or x+1 is reached?
  - Go back to 0?
  - Stay at extreme value?
  - Go to any value?
  - Avoid overflows/underflows as they are really messy

Wrapping



Wrapping

- What if you exceed the boundaries of an unsigned type?
- 0b11111111 + 1 => 100000000
- 255 + 1 => 0 (?)
- Exceeding the maximum value can lead to an overflow and wrapping
- Non-fatal, but leads to funny bugs

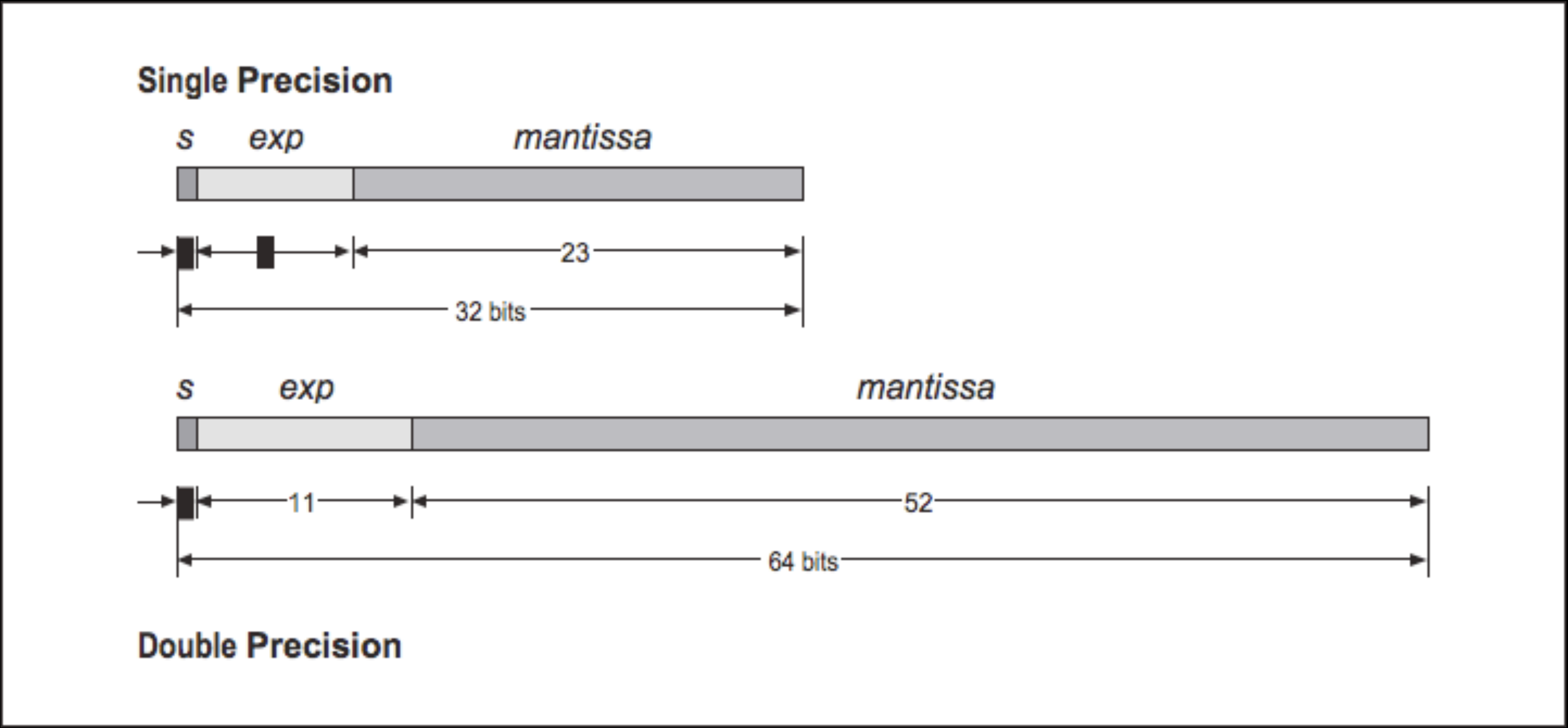
“Nuclear Gandhi”

- In a game (CIV) the peacefulness of a country war represented on a scale from 1-12
- Gandhi, a peaceful person was assigned the lowest value (1)
- When a country becomes a democracy, its peacefulness drops by 2
- Underflow wrapped Gandhi to the maximum value
    - Gandhi became the most aggressive leader

Float

- Computer are not able to properly represent real numbers
- Sign, Exponent, Mantissa
- (+-)1.(Mantissa) * 2^Exponent
- Most real numbers are approximations on the pc

Float vs Double



Why does this never stop?

```c
int main(){
    double d = 0.0;
    while (d != 100.0){
      d += 0.1;
      printf("%f\n", d);
    }

    return 0;
}
```

Rule 2:

Never compare floating point numbers with == or !=

char

- Size: smallest addressable unit
- In most cases 1 Byte
- char can contain an ASCII symbol
- Standard does not specify whether char is signed or unsigned by default

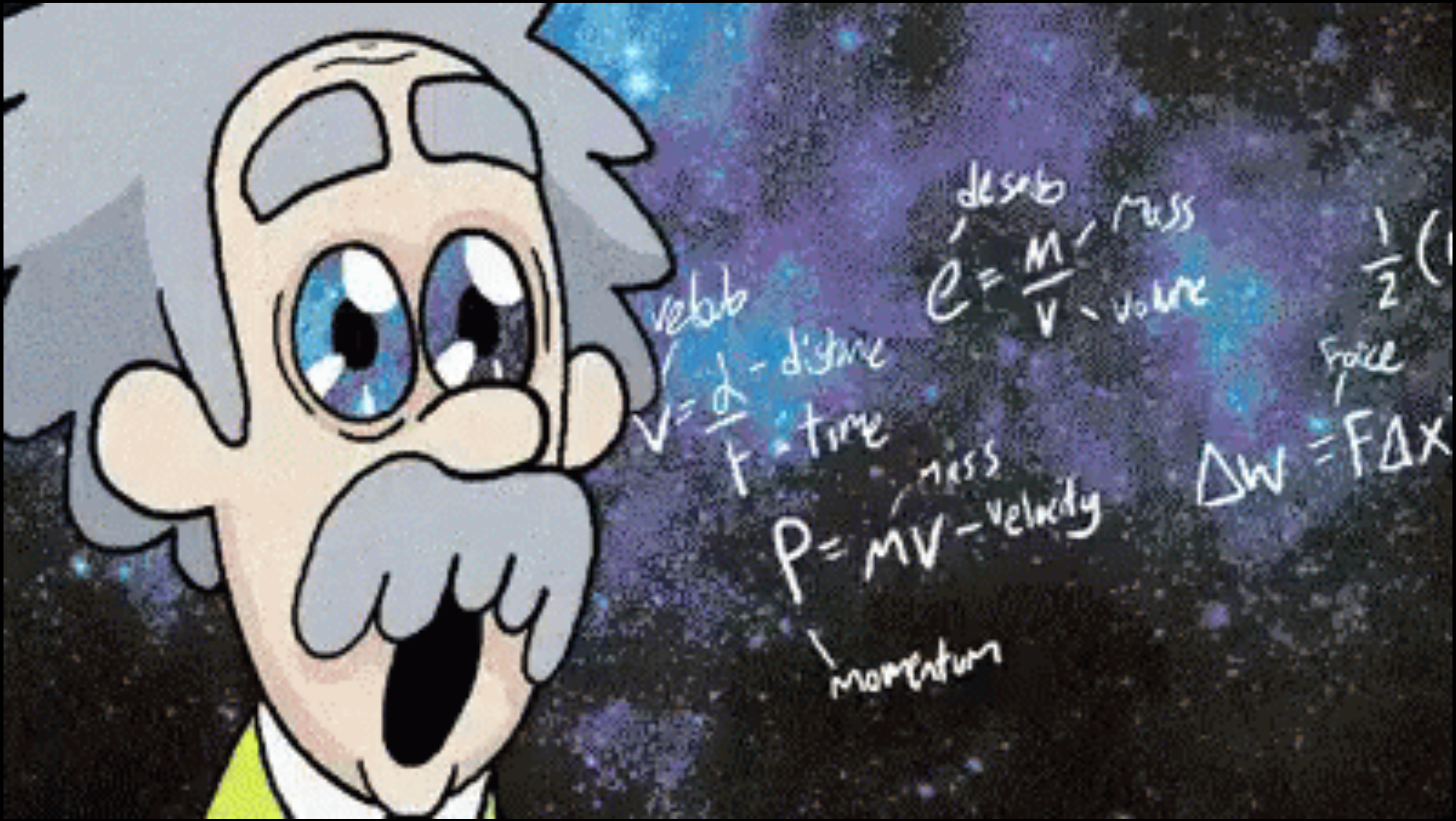
```
char c = 'b';
```

Prepare to be mind blown

- How does a char look like in memory?
- Exactly like a number
- That’s why to be precise, a char is technically just a number that is interpreted differently
- The following is valid C

```c
#include <stdio.h>

int main(){
    char c = 'b';
    printf("%c\n",c);
    c++; // c can be incremented
    printf("%c\n",c);
    c = 67; // ascii code of 'C'
    printf("%c\n",c);
    c = c + 5; // move 5 letters forward
    printf("%c\n",c);
    if('a' > 'A'){ // 'a' == 97, 'A' == 65
        printf("%c\n",'a');
    }
    return 0;
}
```

MATH


desnb
mass
$\rho = \frac{M}{V}$
volume
velob
distance
$V = \frac{d}{t}$
time
mass
$P = MV$
velocity
momentum
$\frac{1}{2}($
force
$\Delta W = F\Delta x$

Operators in C

- +, -, *, /, %, = are well known
- ++, -- could be known
- <<, >>, &, | , ~, ^, obscure black magic

Bitwise arithmetic

- Operations on binary representation of numbers
- Really fast
- Can improve speed if used correctly
- Might break your code if wrong

What the *curseword* is bitwise arithmetic?

- Normal arithmetic: Operate on numbers
  5 + 12 = 17

- Bitwise arithmetic: Iterate through bits of numbers and compare bitwise

5 & 6

0000 0101 & 0000 0111

&	7	6	5	4	3	2	1	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
4	0	0	0	0	0	1	0	0

| and &

- Bitwise OR and AND
- 10 ==   0b 0000 1010
- 5 ==   0b 0000 0101
- 5|10 ==   0b 0000 1111
- 5&10 ==   0b 0000 0000

<< and >>

- << and >> shift the binary representation of a number by some amount of digits

- 5 == 0b00000101
  5 >> 1 == 0b00000010 == 2 (Divide by 2^1)
  5 >> 2 == 0b00000001 == 1 (Divide by 2^2)
  5 << 1 == 0b00001010 == 10 (Multiply by 2^1)
  5 << 2 == 0b00010100 == 20 (Multiply by 2^2)
  5 << 3 == 0b00101000 == 40 (Multiply by 2^3)
  5 << 4 == 0b01010000 == 80 (Multiply by 2^4)

What do we want to do?

- Check whether a number is prime
- Check the price per square cm for pizza
- Calculate mortgage rate
- Find out how high your blood alcohol level is after some beers
- Find out how many seconds you have to work to buy a beer

Bitwise FUN

- Print out the binary representation of an integer number
- Print out the binary representation of a floating point number
- Print all uint16_t numbers, which contain exactly 3 ‚1‘ in their binary representation
- Do bitwise arithmetic on signed and unsigned numbers mixed
- Find the biggest floating point number your pc can represent
- Find the biggest int number your pc can represent