Data Types in C

So, somewhat unlike Python, you must tell the compiler the type of every single variable that you use, without exception! This makes C fast, but also somewhat annoying. It also brings up several topics that we need to discuss and understand.

1. Basic / Primitive Data Types.

   a) char - single character

   b) int - integer

   c) float - single precision floating point

   d) double - double precision floating point

There are also modifiers:

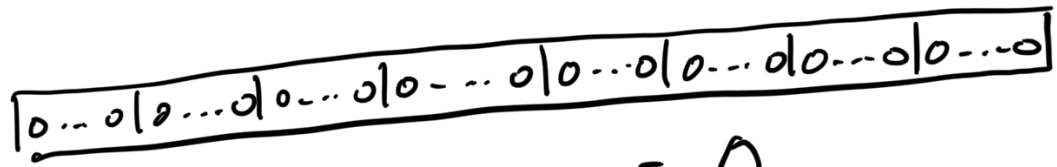
int → short and long

int / char → signed and unsigned

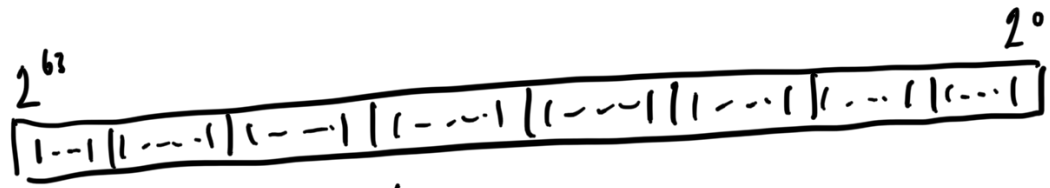
So, ... how? why? → BINARY

Locations in memory → bytes (8 bits)

(a) unsigned long int → 64 bits = 8 bytes.



= 0



$= 2^{64} - 1 =$

1844 674 407 370 955 1615

largest number that can

$\approx 1.84 \times 10^{19}$

be stored as
an integer.

(b) long int → 64 bits

↑
Sign bit

0 = positive
1 = negative

So, you have now only 63 bits to store the actual #, which if all 1's would be $2^{63}-1 = 9.22 \times 10^{18}$

∴ range = -9.22×10^{18} . . . $+9.22 \times 10^{18}$

After this, it actual gets tricky
. . . depends on OS !

See project Primitive Data Types for an example to check your system !

So, what is the point? If we just define a variable as

int i j

what kind of variable is it, by default ???

On most systems, it is probably going to be a 4 byte (32 bit) signed int ... so the range will be:

$$-2147483648 < i < 2147483647$$

... large scale

If you are running [illegible] simulations, this might be too small! Comp. Fluid Dynamics, Particle Physics, IC simulations / Design, etc.
⋮
Need to be careful !!

(c) float and double

Based on scientific notation!!

e.g. -1.235×10^{-61}

How do we store this?

float = 4 byte

double = 8 byte

[long double = 10 byte on some systems]

Let's look at float first!

$$-1.235 \times 10^{-61}$$

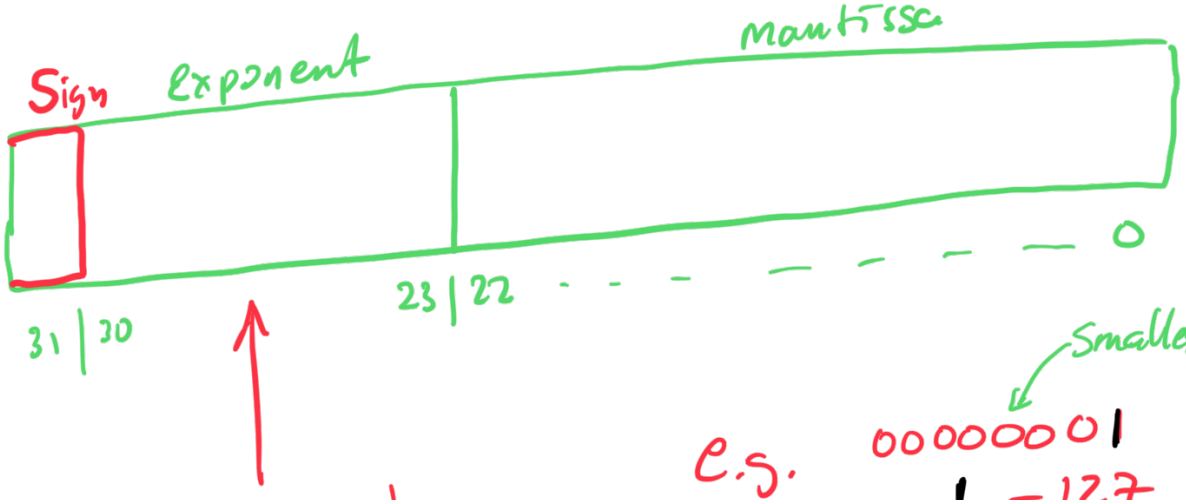
- -1.235: MANTISSA
- 10: base 10
- -61: EXPONENT

→ This is a number between 1 and 10 (base 10)

We need to write this in base 2 → binary.

$$\pm \square \times 2^{\square}$$

$\square$: number between 1 and 2

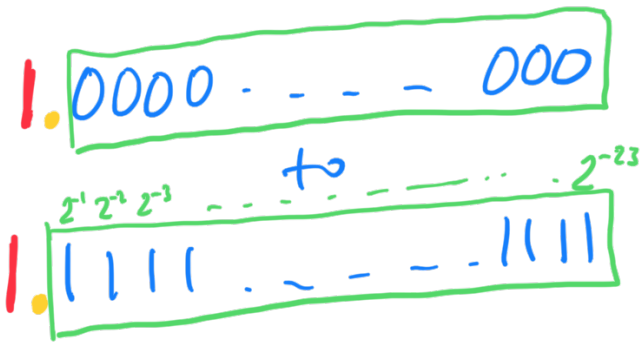


no signbit!
need to subtract 127!
(Excess 128 numbering system)

= −126 ← largest
11111110
= 254 − 127
= 127

The exponents 00000000 and 11111111 are special!

MANTISSA:



↑ extra 1 added in front!

Largest #

0 11111110 1 1

$= 3.403 \times 10^{38}$

Smallest #

0 00000001 0 0

$= 1.175 \times 10^{-38}$

e-127

Number = $(-1)^s$ $2^{\quad}$ 1.m

double → 2.225×10^{-308} to $1.798 \times 10^{+308}$

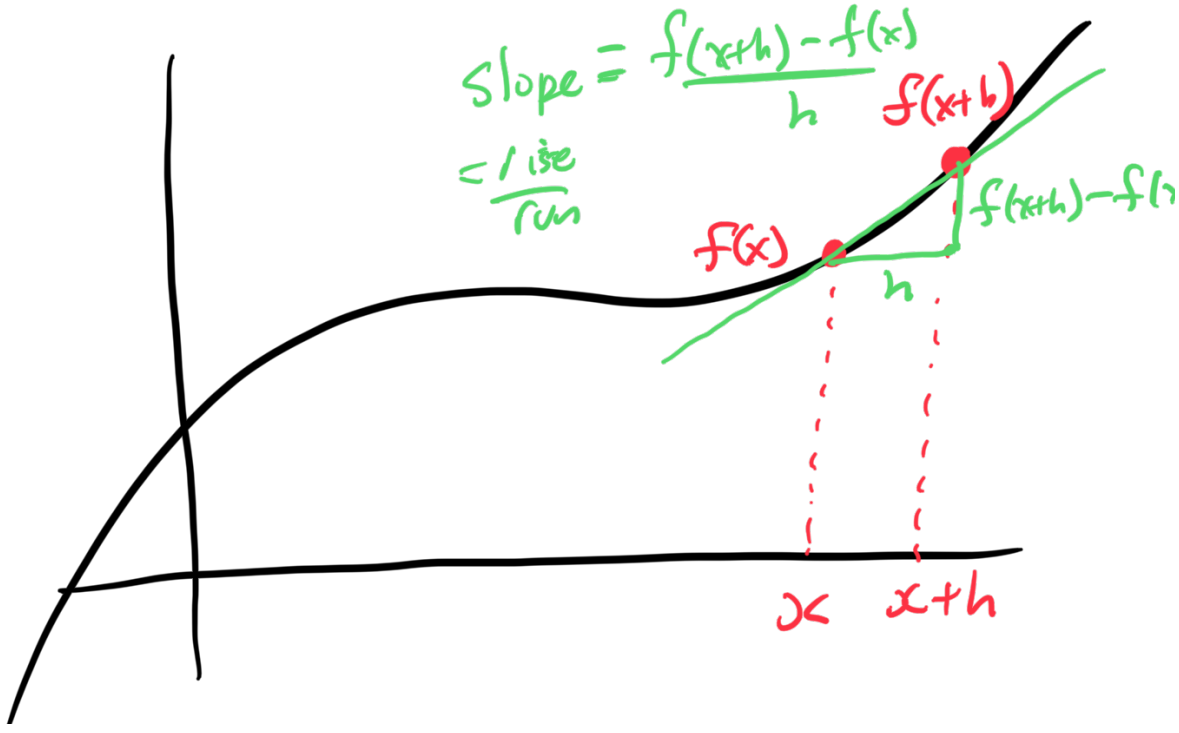
Conclusion: just use double!

Finally, char ... ask a computer scientist to teach you, if you care. Strings, in C, are arrays of chars. More on this later, but we will use chars minimally in this course, because scientists use numbers

Practical Example of Why any of this matters ☺

Calculus (Fundamental theorem)

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$



This would be a cool way to calculate the derivative of any function, numerically, on like, a computer or something.

Question: how small should we make h? (may 10^{-308}?!)

Let's investigate this....

let $f(x) = x^3$

then $f'(x) = 3x^2$

Let's evaluate at $x = 1$

$f(1) = (1)^3 = 1$

$f(x) = x^3$

$f'(x) = 3(1)^2 = \boxed{3}$ Exact answer.

Now, let's try some small values of h.

Let $h = 0.001$

$f(x+h) = (1.001)^3 =$

$f(x) = (1)^3 = 1.003003001$

$$\frac{f(x+h) - f(x)}{h} = \frac{0.003003001}{.001}$$

$$= 3.003001$$

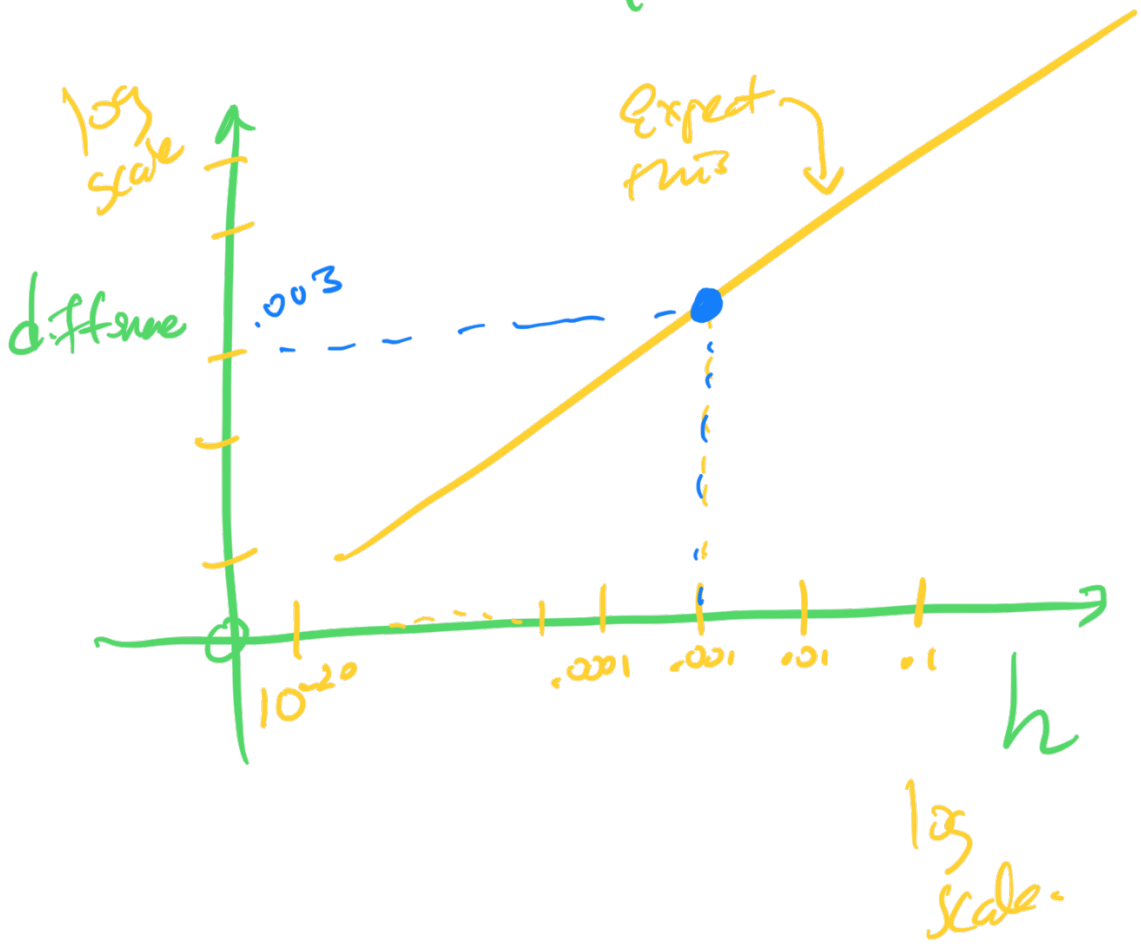
Which is, not quite the same as the exact answer (as expected)

So, let's just make h smaller

smaller, ...

and smaller and

$$\text{difference} = \left| f'(x)_{\text{exact}} - f'(x)_{\text{numerical}} \right|$$



(This could be done in any language ... we will do it in C, because that is what the course is about)

We need to know about

Things we

1. arrays
2. functions
3. plotting!
4. math functions (x^2, x^3, 10^x)
5. loops

Single-precision examples [edit]

These examples are given in bit *representation*, in hexadecimal and binary, of the floating-point value. This includes the sign, (biased) exponent, and significand.

$0\ 00000000\ 00000000000000000000001_2 = 0000\ 0001_{16} = 2^{-126} \times 2^{-23} = 2^{-149} \approx 1.4012984643 \times 10^{-45}$
(smallest positive subnormal number)

$0\ 00000000\ 11111111111111111111111_2 = 007f\ ffff_{16} = 2^{-126} \times (1 - 2^{-23}) \approx 1.1754942107 \times 10^{-38}$
(largest subnormal number)

$0\ 00000001\ 00000000000000000000000_2 = 0080\ 0000_{16} = 2^{-126} \approx 1.1754943508 \times 10^{-38}$
(smallest positive normal number)

$0\ 11111110\ 11111111111111111111111_2 = 7f7f\ ffff_{16} = 2^{127} \times (2 - 2^{-23}) \approx 3.4028234664 \times 10^{38}$
(largest normal number)

$0\ 01111110\ 11111111111111111111111_2 = 3f7f\ ffff_{16} = 1 - 2^{-24} \approx 0.999999940395355225$
(largest number less than one)

$0\ 01111111\ 00000000000000000000000_2 = 3f80\ 0000_{16} = 1$ (one)

$0\ 01111111\ 00000000000000000000001_2 = 3f80\ 0001_{16} = 1 + 2^{-23} \approx 1.00000011920928955$
(smallest number larger than one)

$1\ 10000000\ 00000000000000000000000_2 = c000\ 0000_{16} = -2$
$0\ 00000000\ 00000000000000000000000_2 = 0000\ 0000_{16} = 0$
$1\ 00000000\ 00000000000000000000000_2 = 8000\ 0000_{16} = -0$

$0\ 11111111\ 00000000000000000000000_2 = 7f80\ 0000_{16} = \text{infinity}$
$1\ 11111111\ 00000000000000000000000_2 = ff80\ 0000_{16} = -\text{infinity}$

$0\ 10000000\ 10010010000111111011011_2 = 4049\ 0fdb_{16} \approx 3.14159274101257324 \approx \pi$ (pi)
$0\ 01111101\ 01010101010101010101011_2 = 3eaa\ aaab_{16} \approx 0.333333343267440796 \approx 1/3$

$x\ 11111111\ 10000000000000000000001_2 = ffc0\ 0001_{16} =$ qNaN (on x86 and ARM processors)
$x\ 11111111\ 00000000000000000000001_2 = ff80\ 0001_{16} =$ sNaN (on x86 and ARM processors)