CSCI 210 Systems Programming

Week 6

Introduction to C

Compiling and Running Programs from the Command Line

Data Representation

Reminder: Why C?

- C provides low-level powerful functions that allow interacting with the Operating System parts such as the memory, directly
- Most system software including OS kernels are written in C/C++

Effective use of the C language

- You need to master your understanding of various OS and hardware components
- You must be aware of the system architecture and details of operation
- We will be using C in Linux on the x86-64 architecture
- We will use the gcc compiler and use the C11 version:
  - Replaced C99 in 2011 with standardized support for multi-threaded programs, memory alignment control, Unicode support, type-generic macros, etc.
  - C17 fixed minor defects in C11 without adding new features.

A simple computer model

- Data in memory is stored at accessible addresses
- CPU is able to manipulate data stored in memory and access I/O
- Program code is executed as a series of instructions. The instructions can:
  - Manipulate memory
  - Interact with input/output devices (examples?)
  - Display results to the user
- The program code is also stored in memory (von Neumann architecture), but possibly not accessible

Modern Multi-Tasking OS

- Most modern OSes (including Unix/Linux) provide a particular computer model for memory/program storage
- Each *process* has its own dedicated resources, i.e., each process appears to have:
  - A dedicated CPU
  - Private, dedicated memory
  - Private I/O
- OS provides mechanisms to share existing resources among all active processes

Program execution

- C programs (and all other programs, too) are translated into machine instructions
- Computer executes these instructions in order
- Instruction examples:
    - Add two numbers together
    - Store a number to a location in memory
    - Retrieve a sensor reading
    - Display a result
- The instructions, operands, results are all numbers!

Imperative Programming

- C is an imperative language
- It consists of a list of statements
- Each statement is an instruction to the computer to do something
- Statements can be grouped into functions
- The computer executes the program from beginning to end (roughly) – i.e., imperative
- Modern systems (especially interactive systems such as phones/robots) allow for event-driven programming

C programs

- Every C program starts with the function `main()`

```
int main(int argc, char **argv) {
  return 0;
}
```

- Every C function takes zero or more arguments
- Every C function can return a single value
  - You can use some of the arguments as placeholders for additional return values (you will do this in Project #2)
- Every statement ends with a semi-colon (;)
- C programs are stored in files that end with .c extension

main()

```
int main(int argc, char *argv[]) {
   return 0;
}
```

return type

arguments

Variable categories in C

```
                        |- non-static-|
           |- global-|                |- initialized or uninitialized
           |            |- static  ---|
Variables -|
           |- local  -|- register---|- in CPU register (if possible)
                      |- automatic -|- allocated on stack
                      |- static  -- |- initialized or uninitialized
```

Fig. 2.3 Variables in C

- Global variables are defined outside of any functions
- Local variables are defined inside functions
- Static globals are visible only to the file they are defined in
- The initial values of global and static local variables are stored inside the compiled executable. Non-initialized global variables are cleared to 0 at the start of the program execution.
- Static local variable are like global variable whose scope is the function they are defined in.

Compiling a C Program

- Assume the program is saved in a file named prog.c
- We can compile it to an executable program as follows

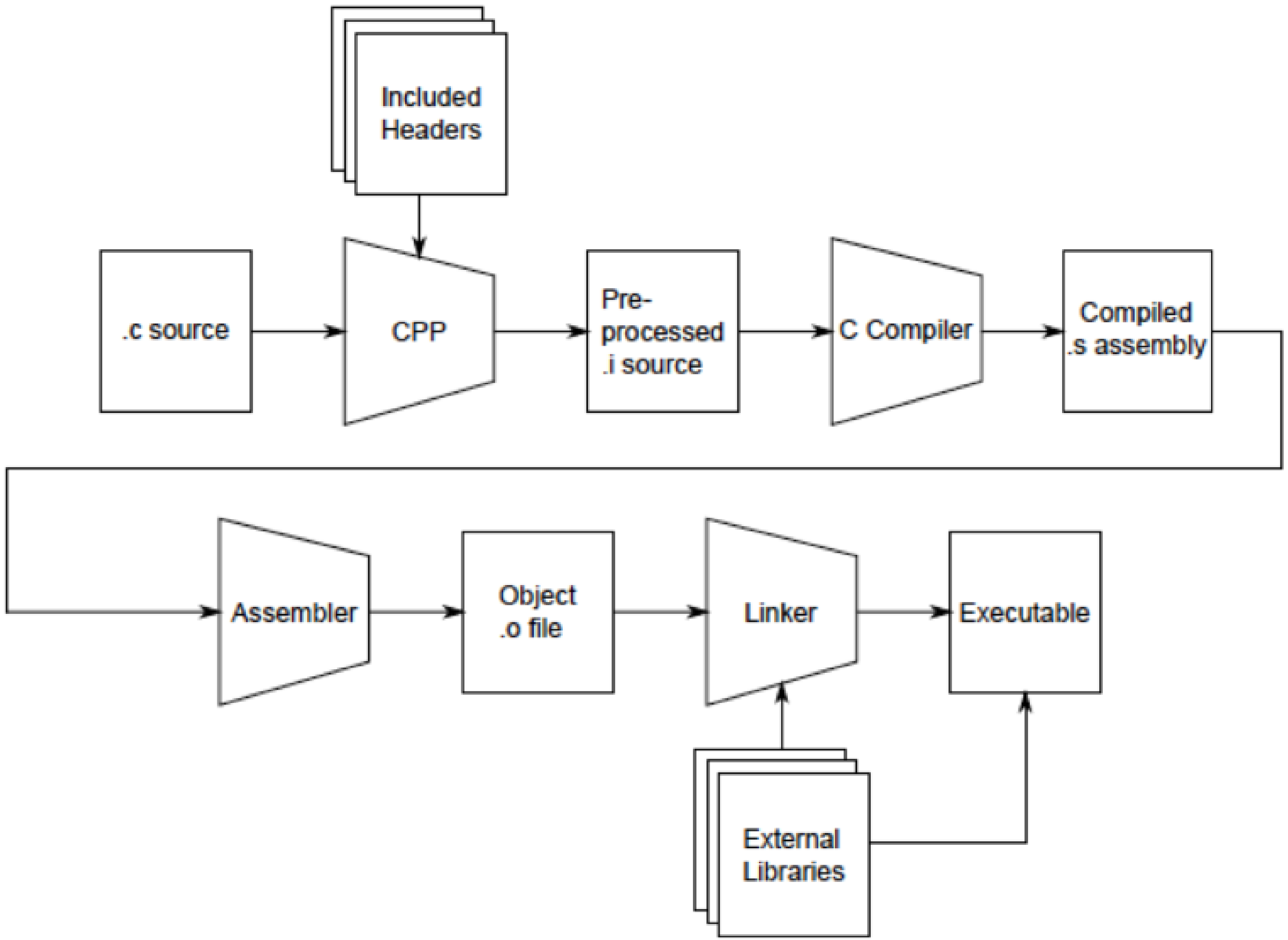
```
$ gcc prog.c
```

- This produces a file named a.out, which is a native binary and can be run as follows

```
./a.out
```

- The program `gcc` actually invokes the compiler toolchain, which is a sequence of many tools

Complete compiler toolchain



The compiled object/executable file

- Contains a header with the size of CODE, DATA, and BSS sections
  - Process creation uses this information to determine the size of execution image in memory
- CODE section contains machine instructions
- DATA section contains the initial values of initialized global and static local variables.
- BSS (Block Starting Symbol) contains the list of uninitialized global and static local variables
- It also has relocation information for pointers and offsets and also a Symbol Table containing a list of non-static globals and function signatures (which are available for cross-reference by other object files).
  - Non-static globals can be accessed from other files with the “extern” modifier.
- We will discuss “Linking” next week in detail.

Example program compilation

- gcc –g –Wall –std=c11 –o example example.c

Data representation

- char
  - Unsigned vs signed
- int
  - Signed integers: Two’s complement
  - Long integers
- float
  - IEEE floating point standards
  - float vs double
- What is the significance of knowing these?
- The following slides are from various publicly available lecture slides (UW, Stanford, METU) and Wiki pages

Notes

- **There are limitations**
  - Memory is finite, numbers/data are not finite
  - We can only represent so much
  - We have $\mathbf{2}^{w}$ distinct bit patterns with **w** bits
- **Design Decisions**
  - **Efficient/Fast and Easy to Implement**
  - **Accuracy**
  - Range
  - Precision

The `char` type

- It is always 1 byte
  - The Unicode character support in the C11 standard have 2 bytes and 4 bytes versions, which are named differently:
    - char16_t, char32_t
- When treated as an integer, can potentially store negative values depending on the machine. Don’t assume it is always positive.
  - Use `unsigned char` if you want positive only values
- Let’s take a look an example: `chartest.c`

Unsigned Integers

- **Unsigned values follow base 2 system**
- Example of converting from base 2 to base 10

$$b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0 2^0$$

- Min value is 0
- Max value is 2^w-1
- Example program to see sizes of various types and the limits
  - limits.h for integer type limits
  - float.h for floating type limits
  - sizeof operator to check the size of a type (or an expression)
    - Syntax: `sizeof( T )`, or `sizeof exp`

Limits

Unsigned Values

UMin = 0

000…0

UMax = $2^{w}-1$

111…1

Two's Complement Values

TMin = -2^{w-1}

100…0

TMax = $2^{w-1}-1$

011…1

Negative one

111…1 0xF…F

Values for W = 32

	Decimal	Hex	Binary
UMax	4,294,967,296	FF FF FF FF	11111111 11111111 11111111 11111111
TMax	2,147,483,647	7F FF FF FF	01111111 11111111 11111111 11111111
TMin	-2,147,483,648	80 00 00 00	10000000 00000000 00000000 00000000
-1	-1	FF FF FF FF	11111111 11111111 11111111 11111111
0	0	00 00 00 00	00000000 00000000 00000000 00000000

Values for W = 64

LONG_MIN = -9223372036854775808

LONG_MAX = 9223372036854775807

ULONG_MAX = 18446744073709551615

Signed integers

- One possible solution:

  - Sign/magnitude notation

    1 101 = - 5

    0 101 = + 5

- Problems:
  - Two different representations for 0:
    - 1 000 = +0
    - 0 000 = -0
  - Addition & subtraction require a watch for the sign! Otherwise, you get wrong results:
    - 0 010 (+2) + 1 010 (-2) = 1 100 (-4)

Alternative solution for signed integers

- Two's complement instead of sign-magnitude representation
  - Positive numbers have a leading 0.
    - 5 => 0101
  - The representation for negative numbers is found by subtracting the absolute value from 2^N for an N-bit system:
    - $-5 \Rightarrow 2^4 - 5 = 16 - 5 = 11_{10} \Rightarrow 1011_2$
- Advantages:
  - 0 has a single representation: +0 = 0000, -0 = 0000
  - Arithmetic works fine without checking the sign bit:
    - 1011 (-5) + 0110 (6) = 0001 (1)
    - 1011 (-5) + 0011 (3) = 1110 (-2)

Two’s complement

- Shortcut to convert from “two’s complement” :
  - If the leading bit is zero, no need to convert.
  - If the leading bit is one, invert the number and add 1.
- What is our range?
  - With 2’s complement we can represent numbers from -2^{N-1} to $2^{N-1} - 1$ using N bits.

All possible values in a 4-bit system

0000: 0	1111: -1
0001: 1	1110: -2
0010: 2	1101: -3
0011: 3	1100: -4
0100: 4	1011: -5
0101: 5	1010: -6
0110: 6	1001: -7
0111: 7	1000: -8

Two’s complement

- Another shortcut to interpret negative numbers in two’s complement:
  - Think of the first bit to have a (-) coefficient in place value representation.
  - 1101: $-1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = -8 + 4 + 0 + 1 = -3$
  - In an 8-bit system:
    - 1101 1001:
    - $-1*2^7 + 1*2^6 + 1*2^4 + 1*2^3 + 1*2^0 = -128 + 64 + 16 + 8 + 1 = -39$.

Two's complement

- Example:
  - We want to compute: 12 – 6
  - 12 => 01100
  - -6 => -(00110) => (11001)+1 => (11010)
- 12 – 6 =

```
  01100
+ 11010
-----------
  00110 => 6
```

So, addition and subtraction operations are the same in the Two's Complement representation

Thanks to these advantages, two's complement is the most common way to represent signed integers on computers.

Real Numbers

Problem: unlike with the integer number line, where there are a finite number of values between two numbers, there are an *infinite* number of real number values between two numbers!

Integers between 0 and 2: 1

Real Numbers Between 0 and 2: 0.1, 0.01, 0.001, 0.0001, 0.00001,...

We need a fixed-width representation for real numbers. Therefore, by definition, *we will not be able to represent all numbers*.

Real Numbers

Problem: every number base has un-representable real numbers.

Base 10: $1/6_{10} = 0.16666666....._{10}$

Base 2: $1/10_{10} = 0.00011001100110011001 1..._2$

Therefore, by representing in base 2, *we will not be able to represent all numbers*, even those we can exactly represent in base 10.

Idea: Use Fixed Point

- Like in base 10, let's add binary decimal places to our existing number representation.

5	9	3	4	.	2	1	6
10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
1	0	1	1	.	0	1	1
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}

Fixed Point

- Like in base 10, let's add binary decimal places to our existing number representation.

1	0	1	1	.	0	1	1
8s	4s	2s	1s		1/2s	1/4s	1/8s

- **Pros:** arithmetic is easy! And we know exactly how much precision we have.

Fixed Point

- **Problem:** Where should we put the fixed point? Range versus precision

Base 10		Base 2
5.07E30	=	$10\underbrace{\cdots\cdots}_{\text{100 zeros}}0.1$
9.86E-32	=	$0.0\underbrace{\cdots\cdots}_{\text{100 zeros}}01$

> To be able to store both these numbers using the same fixed point representation, the bitwidth of the type would need to be at least 207 bits wide!

Other ideas, considerations?

What would be nice to have in a real number representation?

- Represent widest range of numbers possible
- Flexible “floating” decimal point
- Represent scientific notation numbers, e.g. 1.2×10^6
- Still be able to compare quickly
- Have more predictable over/under-flow behavior

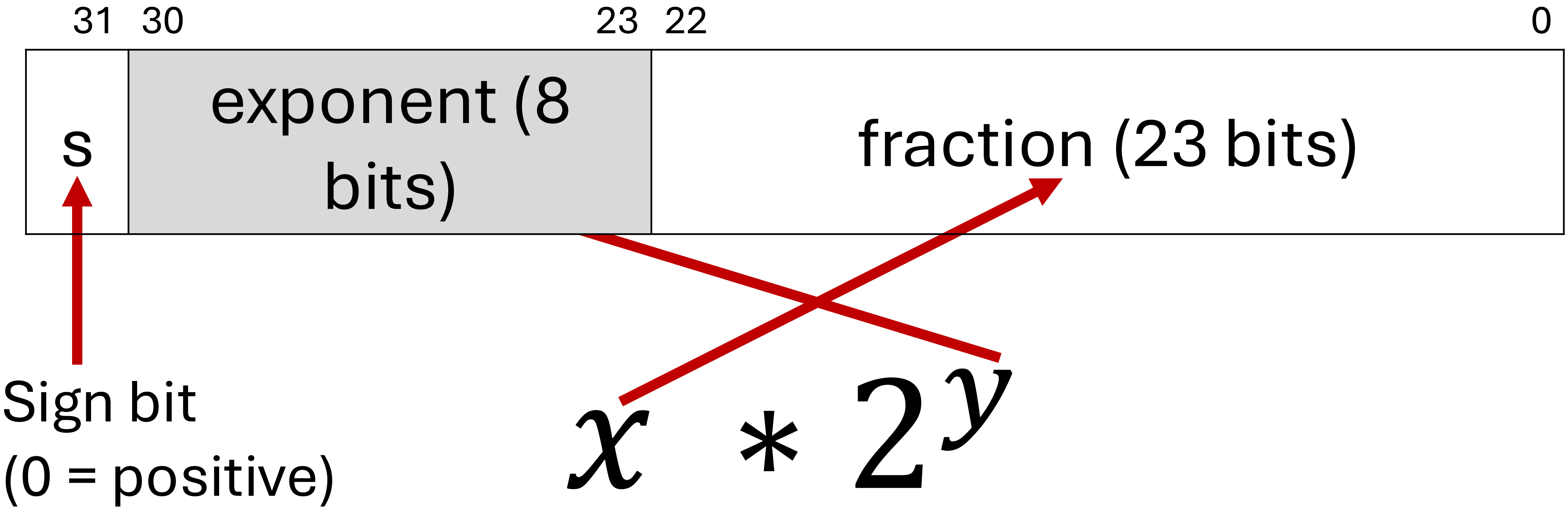
IEEE Floating Point

Let's aim to represent numbers of the following scientific-notation-like format:

$$x \ast 2^{y}$$

With this format, 32-bit floats represent numbers in the range ~1.2 x10^{-38} to ~3.4 x10^{38}! Is every number between those representable? **No**.

IEEE Single Precision Floating Point



Exponent

s	exponent (8 bits)	fraction (23 bits)

Exponent (Binary)	Exponent (Base 10)
11111111	?
11111110	?
11111101	?
11111100	?
…	?
00000011	?
00000010	?
00000001	?
00000000	?

Exponent

s	exponent (8 bits)	fraction (23 bits)

Exponent (Binary)	Exponent (Base 10)
11111111	RESERVED
11111110	?
11111101	?
11111100	?
…	?
00000011	?
00000010	?
00000001	?
00000000	RESERVED

Exponent

s	exponent (8 bits)	fraction (23 bits)

Exponent (Binary)	Exponent (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
…	…
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

Exponent

s	exponent (8 bits)	fraction (23 bits)

- The exponent is **not** represented in two’s complement.
- Instead, exponents are sequentially represented starting from 000…1 (most negative) to 111…10 (most positive). This makes bit-level comparison fast.
- **Actual value = binary value – 127 (“bias”)**

11111110	254 – 127 = 127
11111101	253 – 127 = 126
…	…
00000010	2 – 127 = -125
00000001	1 – 127 = -126

Fraction

s	exponent (8 bits)	$\boxed{\text{fraction (23 bits)}}$

$$x * 2^{y}$$

- We could just encode whatever x is in the fraction field. But there's a trick we can use to make the most out of the bits we have.

An Interesting Observation

In Base 10:

$42.4 \times 10^5 = 4.24 \times 10^6$

$324.5 \times 10^5 = 3.245 \times 10^7$

$0.624 \times 10^5 = 6.24 \times 10^4$

We tend to adjust the exponent until we get down to one place to the left of the decimal point.

In Base 2:

$10.1 \times 2^5 = 1.01 \times 2^6$

$1011.1 \times 2^5 = 1.0111 \times 2^8$

$0.110 \times 2^5 = 1.10 \times 2^4$

Observation: in base 2, this means there is *always* a 1 to the left of the decimal point!

Fraction

s	exponent (8 bits)	fraction (23 bits)

$$x * 2^y$$

- We can adjust this value to fit the format described previously. Then, x will always be in the format **1.XXXXXXXXX...**
- Therefore, in the fraction portion, we can encode just what is *to the right* of the decimal point! This means we get one more digit for precision.

Value encoded = 1._[FRACTION BINARY DIGITS]_

Practice

Sign	Exponent						Fraction			
0	0	…	0	0	0	1	0	1	0	…

Is this number:

A) Greater than 0?

B) Less than 0?

Is this number:

A) Less than -1?

B) Between -1 and 1?

C) Greater than 1?

Representing Zero

The float representation of zero is all zeros (with any value for the sign bit)

Sign	Exponent	Fraction
`any`	`All zeros`	`All zeros`

- This means there are two representations for zero! ☹

Representing Small Numbers

If the exponent is all zeros, we switch into “denormalized” mode.

Sign	Exponent	Fraction
`any`	`All zeros`	`Any`

- We now treat the exponent as -126, and the fraction as *without* the leading 1.
- This allows us to represent the smallest numbers as precisely as possible.

Representing Exceptional Values

If the exponent is all ones, and the fraction is all zeros, we have +-infinity.

Sign	Exponent	Fraction
`any`	`All ones`	`All zeros`

- The sign bit indicates whether it is positive or negative infinity.
- Floats have built-in handling of overflow!
  - Infinity + anything = infinity
  - Negative infinity + negative anything = negative infinity

Representing Exceptional Values

If the exponent is all ones, and the fraction is nonzero, we have **Not a Number.**

Sign	Exponent	Fraction
`any`	`All ones`	`Any nonzero`

- NaN results from computations that produce an invalid mathematical result.
  - Sqrt(negative)
  - Infinity / infinity
  - Infinity + -infinity
  - Etc.

Floating Point Representation Summary

Exponent	Mantissa	Meaning
0x00	0	± 0
0x00	Non-zero	± denorm num
0x01 – 0xFE	Anything	± norm num
0xFF	0	± ∞
0xFF	Non-zero	NaN

Skipping Numbers

- We said that it's not possible to represent *all* real numbers using a fixed-width representation. What does this look like?

Float Converter

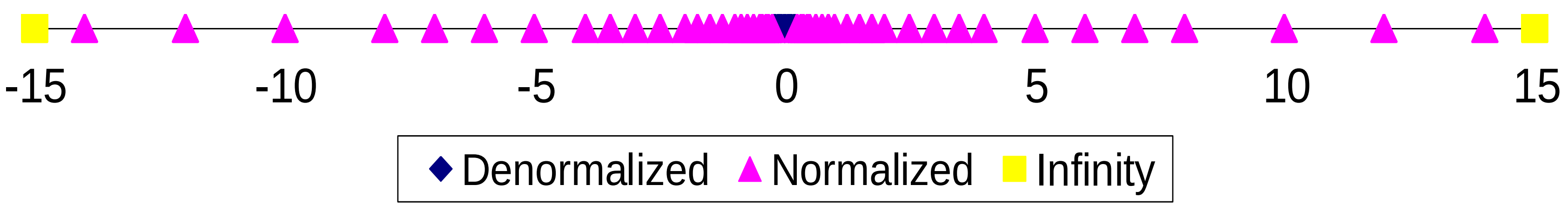
- https://www.h-schmidt.net/FloatConverter/IEEE754.html

Floats and Graphics

- https://www.shadertoy.com/view/4tVyDK

Distribution of Values

- **What can't we get?**
  - Between largest norm and infinity: **Overflow**
  - Between zero and smallest denorm: **Underflow**
  - Between norm numbers?: **Rounding**



Floating Point Ranges

- 32-bit floating point (type **float**):
  - ~1.2 $\times 10^{-38}$ to ~3.4 $\times 10^{38}$
  - Not all numbers in the range can be represented (not even all integers in the range can be represented!)
  - Gaps can get quite large! (larger the exponent, larger the gap between successive fraction values)
- 64-bit floating point (type **double**):
  - ~2.2 $\times 10^{-308}$ to ~1.8 $\times 10^{308}$

- See more types and format specifiers at:
  - https://en.wikipedia.org/wiki/C_data_types

Floating Point Arithmetic

Is this just overflowing? No. It is actually something more subtle.

```
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b);  // prints 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b)); // prints 3.14
```

Let’s look at the binary representations for 3.14 and 1e20:

	31	30 … 23	22 … 0
3.14:	0	10000000	10010001111010111000011
1e20:	0	11000001	01011010111100011101100

Floating Point Arithmetic

	31	30 … 23	22 … 0
3.14:	0	10000000	10010001111010111000011

	31	30 … 23	22 … 0
1e20:	0	11000001	01011010111100011101100

To add real numbers, we must align their binary points:

```
                       3.14
+   100000000000000000000.00
    ------------------------
    100000000000000000003.14
```

What does this number look like in 32-bit IEEE format?

Floating Point Arithmetic

The binary representation for 1e20 + 3.14 equals the following:

31	30 … 23	22 … 0
0	11000001	01011010111100011101100

Which is the **same** as the binary representation for 1e20!

We don't have enough bits to differentiate between 1e20 and 1e20 + 3.14.

Take home message?

Floating point arithmetic is not associative. The order of operations matters!

```
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b);  // prints 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b)); // prints
3.14
```

- The first line loses precision when first adding 3.14 and 1e20, as we have seen.
- The second line first evaluates 1e20 – 1e20 = 0, and then adds 3.14

Floating Point Arithmetic

Float arithmetic is an issue with most languages, not just C!

- http://geocar.sdf1.org/numbers.html

Floats Summary

- IEEE Floating Point is a carefully-thought-out standard. It’s complicated, but engineered for their goals.
- Floats have an extremely wide range, but cannot represent every number in that range.
- Some approximation and rounding may occur! This means you definitely don’t want to use floats e.g. for currency.
- Associativity does not hold for numbers far apart in the range
- Equality comparison operations are often unwise.

Floating Point Limitations: Math Properties

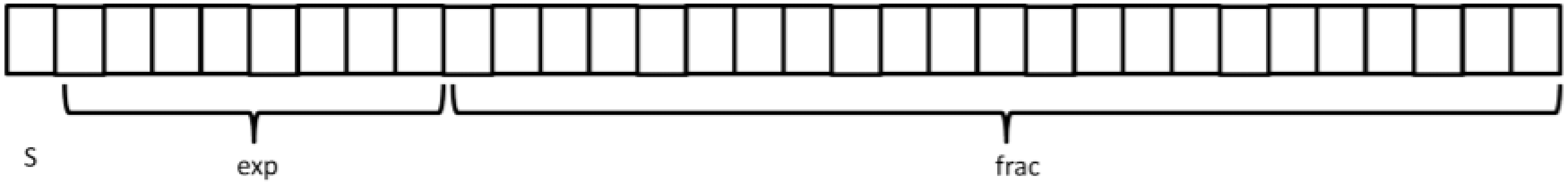
- Exponent overflow yields +∞ or -∞
- Floats with value +∞, -∞, and NaN can be used in operations
  - Result usually still +∞, -∞, or NaN; sometimes intuitive, sometimes not
- Floating point ops do not work like real math, due to rounding!
  - Not associative:
    - (3.14 + 1e100) – 1e100 != 3.14 + (1e100 – 1e100)
  - Not distributive:
    - 100 * (0.1 + 0.2) != 100 * 0.1 + 100 * 0.2
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

Another example

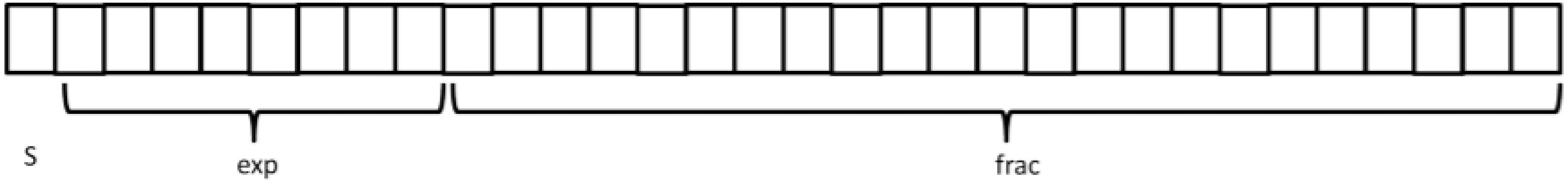
- That demonstrates the advantage of knowing your data representations:
  - https://en.wikipedia.org/wiki/Fast_inverse_square_root
- Try it out by compiling and running the function on isengard

Exercises

- Consider the decimal number **1.25**. Give the IEEE-754 representation of this number as a 32-bit floating-point number.



- Convert the decimal **1.1×2^{-128}** to IEEE 754 single precision



Exercises

- If x and y have type float, give two different reasons that (x+2*y)-y == x+y might evaluate to 0 (i.e., false).

Exercises

- What is the largest positive number we can represent with a 10-bit signed two’s complement integer?

- Bit pattern?
- Decimal value?

Exercises

- Assuming unsigned integers, what is the result when you compute UMAX+1?
- Assuming two's complement signed representation, what is the result when you compute TMAX+1?

Exercises

- Is the ‘==’ operator a good test of equality for floating point values? Why or why not?

Exercises

- Give an example of three floating-point numbers x, y, and z, such that the distributive property $x\,(y + z) = x\,y + x\,z$ does not hold.