Prof. Darrell Long

CSE 13S

On the Nature of Numbers

Srinivasa Ramanujan
Numbers were his friends

On the nature of *Numbers*

We all think that we know what a number is:

- 1, 3, 5, 7, 11, … are all numbers, right?
- Not quite, they are the *representation* of numbers.
  - You cannot hold a number in your hand.

Numbers exist independently of their representation.

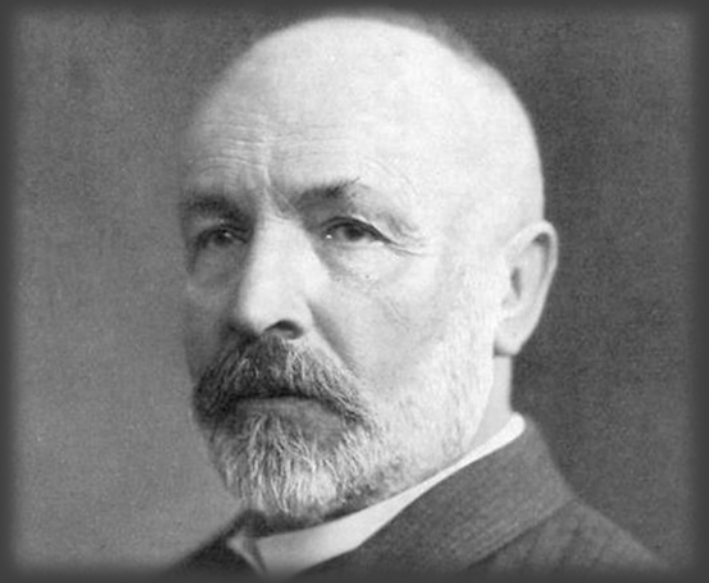
- There are many ways to write the same number.

Our method for writing numbers derives from the Hindu-Arabic numeral system:

- Aryabhata (आर्यभट) introduced positional notation, and Brahmagupta introduced the notion of zero.

We write our numbers base 10 since most of us have 10 fingers.

- The ancient Babylonians used base 60, which is why we have 60 seconds in a minute, 60 minutes in an hour.



Georg Cantor

The *Continuum Hypothesis* drove him mad

Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk. —Leopold Kronecker

Kinds of Numbers

$\mathbb{N} = \{1, 2, 3, \dots\}$

$\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$

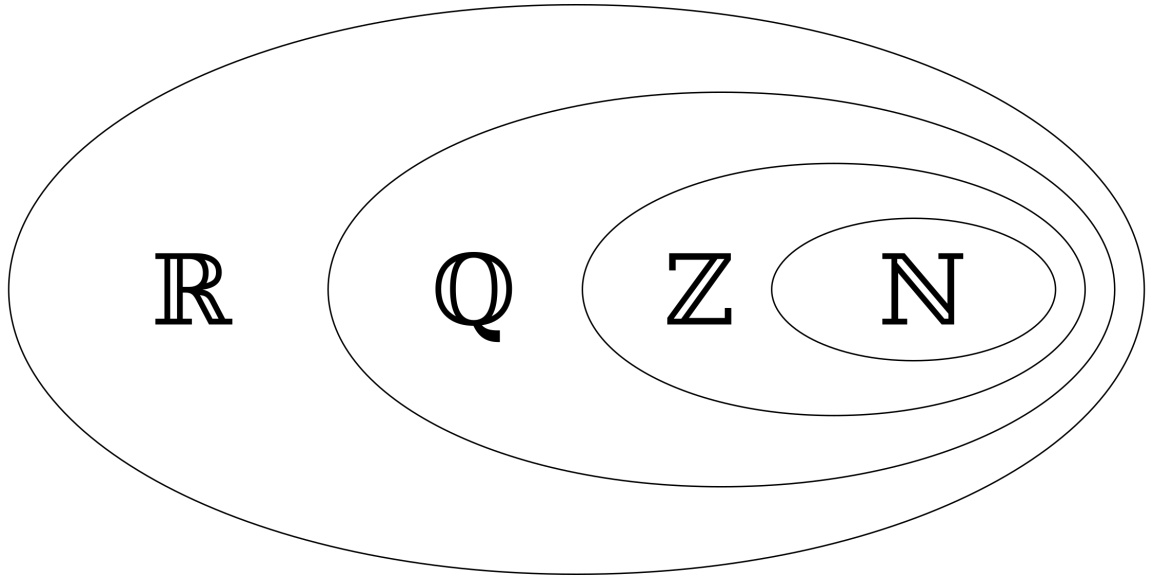
$\mathbb{Q} = \{\frac{a}{b} : a, b \in \mathbb{Z}\}$

$\mathbb{R} = \mathbb{Q} \cup$ the $irrational$ numbers

$\mathbb{C} = \mathbb{R} \cup$ the *imaginary* numbers

π, e, γ, φ are all *irrational*, and are much more numerous than $\mathbb{Q}$.

Imaginary allows $i = \sqrt{-1}$, and so *complex* is $(a + bi)$, $a, b \in \mathbb{R}$.



Computers can represent *none* of those sets

- Computers do arithmetic in *finite fields*.
  - Which means they have a large, but finite, set of numbers that they can represent.
- It's like saying you can only do arithmetic up to 10 digits.
- A digit for a computer is usually called a *bit*.
  - A bit is either 0 or 1.

Bits	Minimum	Maximum
8	-128	127
16	-32768	32767
32	-2147483648	2147483647
64	-9223372036854775808	9223372036854775807

Bits	Minimum	Maximum
8	0	255
16	0	65535
32	0	4294967295
64	0	18446744073709551615

Positional Number Systems

Aryabhata (आर्यभट) taught us that every member of $\mathbb{N}$ can be written:

- $a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b^1 + a_0 b^0$
- b is the base, be it 2, 8, 10 (our usual base), 16, …
- $a_i \in \{0, \ldots, b-1\}$

$$1962 = 1 \times 10^3 + 9 \times 10^2 + 6 \times 10^1 + 2 \times 10^0$$
$$= 7\mathrm{AA}_{16} = 3652_8 = 2200200_3$$
$$= 11110101010_2$$

Those all represent the same number.

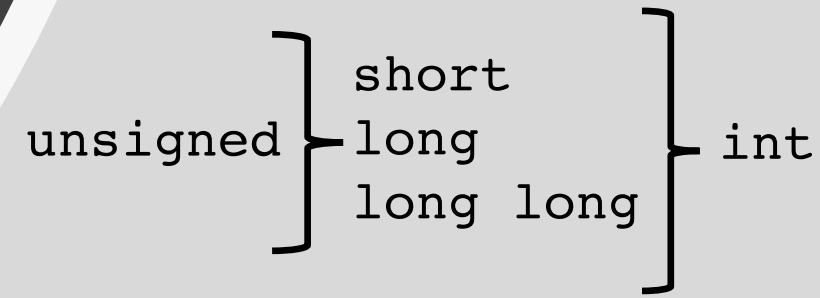
The answer to the ultimate question is 42.

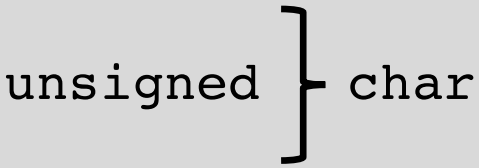
The ultimate question is: What is 6×9?

It's true, since $6 \times 9 = 54 = 42_{13}$.

Specifying an Integer in C

```
char
unsigned char
int
unsigned int
short int
unsigned short int
long int
unsigned long int
long long int
unsigned long long int
```





C Integer Data Types

Name	Size
`char`	Usually 8 bits
`short`	Usually 16 bits
`int`	At least 16 bits
`long`	At least 32 bits
`long long`	Probably 64 bits

#include <stdint.h>

Signed	Unsigned	Size
int8_t	uint8_t	8 bits
int16_t	uint16_t	16 bits
int32_t	uint32_t	32 bits
int64_t	uint64_t	64 bits

Let's ask Linux!

darrell — ssh unix.ucsc.edu — 80×24

```
[unix3.lt.ucsc.edu [101]% vi sizes.c
[unix3.lt.ucsc.edu [102]% cat sizes.c
#include <stdio.h> // The I/O library

int main(void)
{
  printf("char is %d bits\n", sizeof(char) * 8);
  printf("short is %d bits\n", sizeof(short) * 8);
  printf("int is %d bits\n", sizeof(int) * 8);
  printf("long is %d bits\n", sizeof(long) * 8);
  printf("long long %d bits\n", sizeof(long long) * 8);
  return 0; // Success
}
[unix3.lt.ucsc.edu [103]% gcc sizes.c
[unix3.lt.ucsc.edu [104]% ./a.out
char is 8 bits
short is 16 bits
int is 32 bits
long is 64 bits
long long 64 bits
unix3.lt.ucsc.edu [105]%
```

Standard I/O

Print to the screen

How many bytes?

Bytes have 8 bits

Binary Arithmetic

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

Binary arithmetic is just like normal arithmetic, except you have only two digits (bits): 0 and 1.

Addition works as you would expect:

- $0 + 0 = 0$
- $1 + 0 = 1$
- $1 + 1 = 10$ or we might say, “0 carry the 1”

Multiplication is even simpler:

- $0 \times 0 = 0$
- $1 \times 0 = 0$
- $1 \times 1 = 1$

- $101 + 11 = 1000$
- $101 \times 101 = 11001$

Arithmetic in a Finite Field

- Computers have fixed width integers, typically 8, 16, 32, and 64 bits.
  - You've seen the limits of those sizes.
- How do we implement negative numbers?
  - We could do arithmetic like humans do, but there's an easier way.
- In a finite field we can define for every number its *additive inverse*.
  - If i is a number in a finite field, and $\bar{\imath}$ its additive inverse then $i + \bar{\imath} = 0$.
  - Suppose that we have k bits, then we can have every positive integer from 0 to $2^k - 1$.
  - We can use one half of those as additive inverses and so our numbers range from -2^{k-1} to $2^{k-1} - 1$.

Positive	Negative
0000	0000
0001	1111
0010	1110
0011	1101
0100	1100
0101	1011
0110	1010
0111	1001

Two's Complement Arithmetic

To get the additive inverse of a number *m*,

- Flip the bits in *m*, $0 \to 1$ and $1 \to 0$ (in **C** this is the ~ operator), and then
- Add one to the result.

Let's try $5 = 0101_2$ using 4 bit integers:

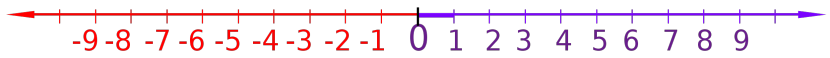
- $\sim 0101_2 + 1 = 1010_2 + 1 = 1011_2$
- $0101_2 + 1011_2 = 10000_2$ but we only have four bits, so we drop the high-order (carry) bit and get 0000_2.

Real Numbers

- Real numbers ($\mathbb{R}$) are:
  - *Continuous*
  - *Uncountably* infinite

There are just as many numbers between any x and y as there in all of $\mathbb{R}$.

- Include all of the:
  - Integers ($\mathbb{Z}$)
  - Rational numbers ($\mathbb{Q}$)
  - Irrational numbers ($\mathbb{R} - \mathbb{Q}$)



René Descartes

Konrad Zuse

2 April 2021

Floating Point Numbers

- Are a proper subset of real numbers:
  - $\mathbb{F} \subsetneq \mathbb{R}$
- Are a proper subset of rational numbers:
  - $\mathbb{F} \subsetneq \mathbb{Q}$
- Are a proper subset of the integers:
  - $\mathbb{F} \subsetneq \mathbb{Z}$
- It's a mistake to think of them as *reals*.
- It is a mistake to think of them as *rationals*.
- They are an *approximation*.

Floating Point Numbers

C Specification	C Name	Implementation	Bits
`float`	Single Precision	IEEE 754 Single	32
`double`	Double Precision	IEEE 754 Double	64
`long double`	Extended Precision	IEEE 754 Quad	128

Decimal and Binary Fractions

$\frac{2}{5} = \frac{4}{10} = 0.4 = 4\times10^{-1}$ which is exact.

$\frac{1}{3} = 0.333333\ldots$ which repeats forever.

There is no power of 10 that divides evenly by 3.

The *fundamental theorem of arithmetic* states every $n \in \mathbb{N}$ has a unique prime factorization.

$10^k = 2^k 5^k$ and none of those factors is 3.

Can we write $\frac{1}{10}$ exactly using binary fractions?

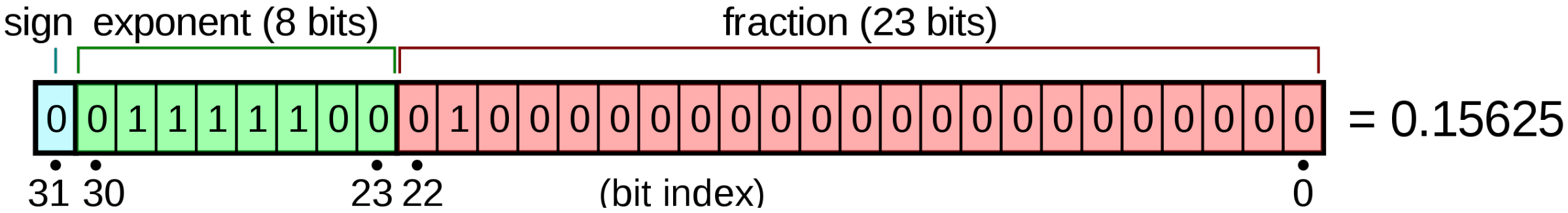
$$a_1 2^{-1} + \cdots + a_k 2^{-k} = 2^{-k} \sum_{i=1}^{k} a_i 2^{k-i}$$

$2^k = 2\times\cdots\times2$, none of which is 5, and when we factor $10 = 2\times5$.

So, 0.1 *cannot* be represented exactly on a digital computer in floating point format.

$$\texttt{float} \neq \mathbb{Q}$$

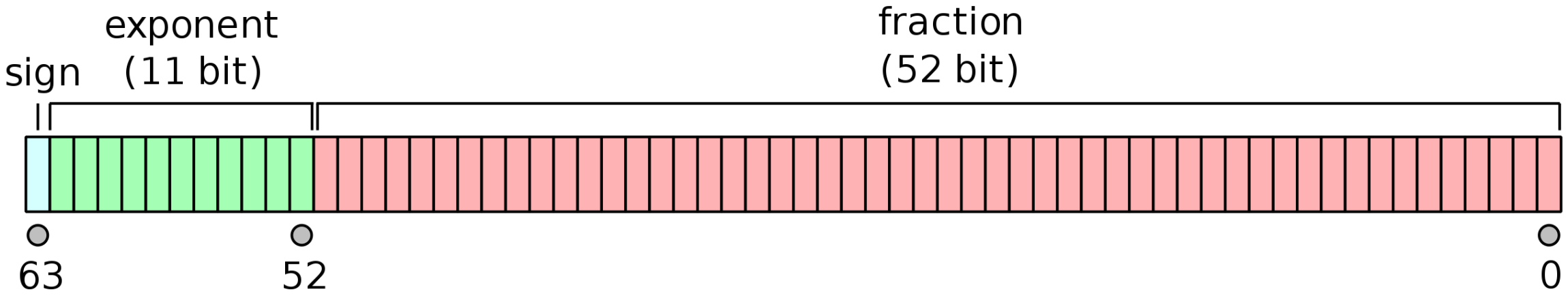
Single Precision



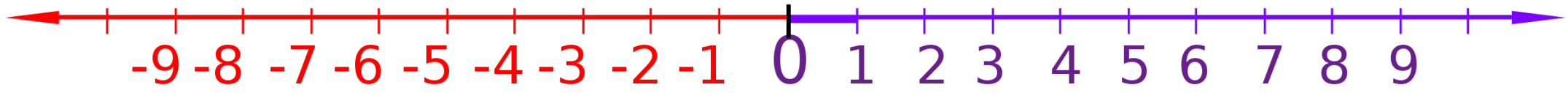
$$(-1)^{b_{31}} \times 2^{(b_{30}\ldots b_{23})-127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

What we are doing is what you learned as *scientific notation* in high school.

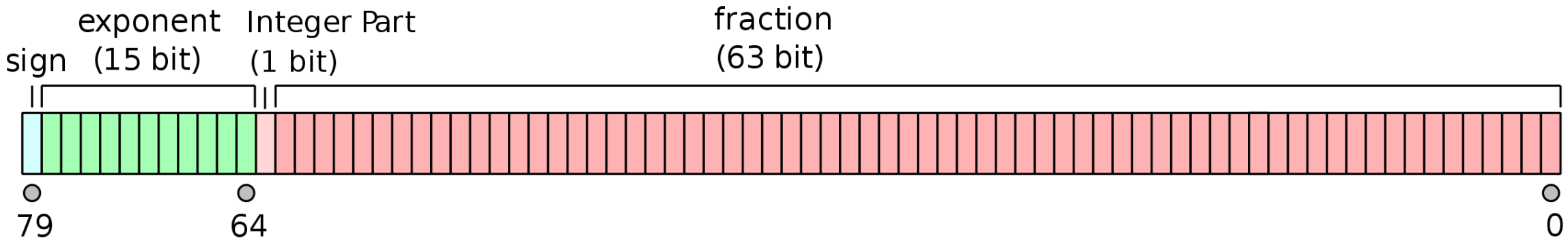
Double Precision



$$(-1)^{b_{63}}\left(1+\sum_{i=1}^{52} b_{52-i}2^{-i}\right)\times 2^{(b_{62}\ldots b_{52})-1023}$$

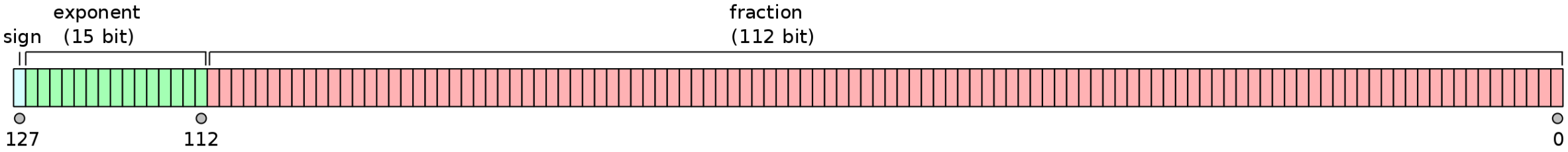


Intel Extended Precision



This format is Intel only, and you should avoid it if you want your results to be portable to other computers.

Quad Precision



$$(-1)^{b_{127}}\left(1+\sum_{i=1}^{112} b_{112-i}2^{-i}\right)\times 2^{(b_{126}\ldots b_{112})-32767}$$

Big endian, little endian

- When we have an integer that requires multiple bytes, what is the address of the byte holding the least significant bit?
  - *Little Endian* means it is the low address byte.
  - *Big Endian* means it is the high address byte.
- You can ask the same question about the most significant bit.
- Most of the time you do not care, unless you are communicating with a computer that uses the other convention.

```
bool isBig()
{
        union {
                uint8_t bytes[2];
                uint16_t word;
        } test;
        test.word = 0xFF00;
        return test.bytes[0];
}

bool isLittle()
{
        union {
                uint8_t bytes[2];
                uint16_t word;
        } test;
        test.word = 0xFF00;
        return test.bytes[1];
}
```

Big Endian versus Little Endian

Big Endian

Little Endian

Random Numbers

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.
—John von Neumann

True random numbers cannot be created using computers.

- Why?
  - Programs are inherently deterministic.
- It has the advantage of *repeatability*, but
- It has the disadvantage of *predictability*.

John von Neumann

Mersenne Twister

A state of the art pseudorandom number generator

❦

But does it generate random numbers?

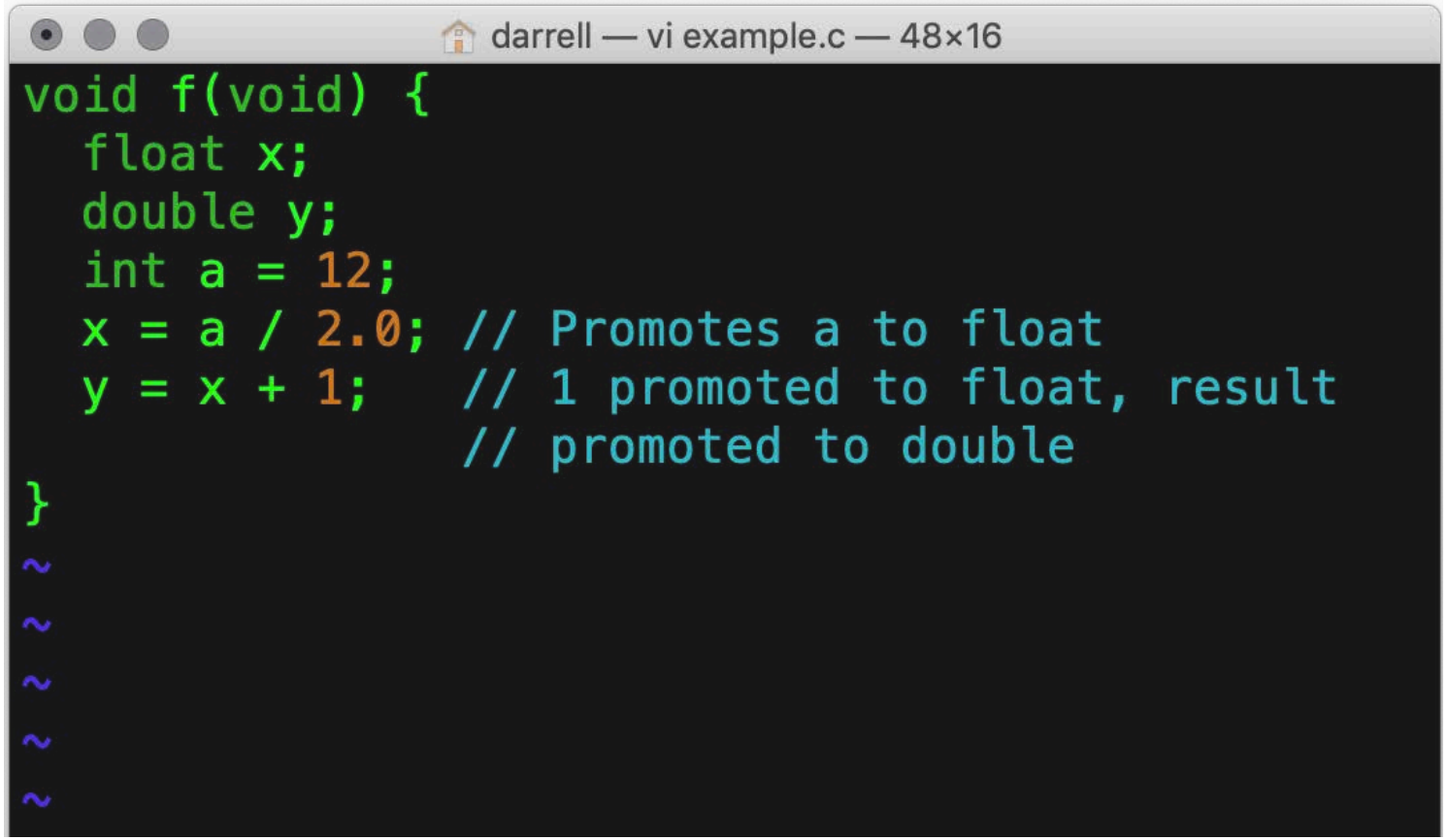
```
unsigned long long genrand64_int64(void) {
  int i;
  unsigned long long x;
  static unsigned long long mag01[2] = {0ULL, MATRIX_A};

  if (mti >= NN) {
    if (mti == NN + 1)
      init_genrand64(5489ULL);

    for (i = 0; i < NN - MM; i++) {
      x = (mt[i] & UM) | (mt[i + 1] & LM);
      mt[i] = mt[i + MM] ^ (x >> 1) ^ mag01[(int)(x & 1ULL)];
    }
    for (; i < NN - 1; i++) {
      x = (mt[i] & UM) | (mt[i + 1] & LM);
      mt[i] = mt[i + (MM - NN)] ^ (x >> 1) ^ mag01[(int)(x & 1ULL)];
    }
    x = (mt[NN - 1] & UM) | (mt[0] & LM);
    mt[NN - 1] = mt[MM - 1] ^ (x >> 1) ^ mag01[(int)(x & 1ULL)];

    mti = 0;
  }

  x = mt[mti++];

  x ^= (x >> 29) & 0x5555555555555555ULL;
  x ^= (x << 17) & 0x71D67FFFEDA60000ULL;
  x ^= (x << 37) & 0xFFF7EEE000000000ULL;
  x ^= (x >> 43);

  return x;
}
```

Arithmetic Operators

- These operators follow the precedence and associativity rules that you learned in high school algebra.

- Modulo is the remainder when you divide two integers:
  - In *number theory*, mod is always non-negative.
  - For negative integers the sign resulting from % depends on the **C** compiler (the sign is the same as the dividend).

*	**Multiplication**
/	Division
%	Modulo (integers only)
+	Addition
–	Subtraction



Type Promotion

- When you have an expression of mixed types, then C will promote the *lower* type to a *higher* type.
  - Lower means that values of that type are a *subset* of the values that can be held by the higher type.
  - Smaller integers are promoted to bigger integers.
  - Integers are promoted to floating point types.
  - Smaller floating point types are promoted to larger floating point types.

Operators	Associativity
* / %	Left
+ –	Left
<< >>	Left
< <= > >=	Left
== !=	Left
&	Left
^	Left
|	Left
&&	Left
||	Left
?:	Right
= += –= *= /= %= …	Right

Operator Precedence

- Operators fall into equivalence classes: for example multiplicative, additive, logical, Boolean, and bit-wise operators.
  - There are more operators than are shown in this table.
- If you are unsure of precedence or associativity then use ( ) to make your expression unambiguous.

Summary

- Computers work with *bits* (0 and 1), and
- These bits are grouped in *bytes* (8 bits),
  - These bytes group into larger quantities called *words*.
- Computers do integer arithmetic in finite fields,
  - Because the words are of fixed size.
  - This allows us to use two's complement arithmetic.
- Computers can only represent a small subset of the integers.
- Computers give us data types like: `float`, `double`, and `long double`.
  - Do not make the mistake of thinking they are *real numbers* ($\mathbb{R}$),
  - They are really a very small subset of the *rationals* ($\mathbb{Q}$).

Ultimately, do we care *how* computers represent numbers? Usually, we do not. But, we do care when (i) we want to manipulate bits, and (ii) when we care about precision, as in scientific calculations. We must always choose a representation that can hold all the values that we need.