CS107 Lecture 10
Floating Point

reading:

*B&O 2.4*
CS107 Topic 5: How can a computer represent real numbers in addition to integer numbers?
Learning Goals

Understand the design and compromises of the floating point representation, including:

• Fixed point vs. floating point
• How a floating point number is represented in binary
• Issues with floating point imprecision
• Other potential pitfalls using floating point numbers in programs
Plan For Today

• Representing real numbers and (thought experiment) fixed point
• Floating Point: Normalized values
• **Break**: Announcements
• Floating Point: Special/denormalized values
• Floating Point Arithmetic
Review: Function pointers

fn_ptr_review.c
Warm-up (?): Function pointers

```c
int compare_strings(const void *p, const void *q) {
    char *str1 = *(const char **) p;
    char *str2 = *(const char **) q;
    return strcmp(str1, str2);
}

void test_bsearch() {
    char *words[] = {"aardvark", "beaver", "capybara"}; // sorted
    int n = sizeof(words) / sizeof(*words);
    char *searchkey = strdup("beaver");
    char **found = bsearch(&searchkey, words, n, sizeof(words[0]), compare_strings);
    printf("found %s\n", found ? *found : "none");
    free(searchkey);
}
```

1. What are types are the parameters passed into `bsearch`?
2. Let's change line 3 to:
```c
    char *str1 = (const char *) p;
```

What happens?

Tips: (1) draw pictures, and (2) read `man` pages carefully.
1. Bits and Bytes - *How can a computer represent integer numbers?*
2. Chars and C-Strings - *How can a computer represent and manipulate more complex data like text?*
3. Pointers, Stack and Heap – *How can we effectively manage all types of memory in our programs?*
4. Generics - *How can we use our knowledge of memory and data representation to write code that works with any data type?*
5. Floats - *How can a computer represent floating point numbers in addition to integer numbers?*
6. Assembly - *How does a computer interpret and execute C programs?*
7. Heap Allocators - *How do core memory-allocation operations like malloc and free work?*
Plan For Today

- Representing real numbers and (thought experiment) fixed point
- Floating Point: Normalized values
- **Break:** Announcements
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic
## Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

<table>
<thead>
<tr>
<th>Number/fraction</th>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/10 (bonus)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

<table>
<thead>
<tr>
<th>Number/fraction</th>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>5</td>
<td>5.0</td>
<td>101.0</td>
</tr>
<tr>
<td>9/8</td>
<td>1.125</td>
<td>1.001</td>
</tr>
<tr>
<td>1/3</td>
<td>0.3333…</td>
<td>0.0101010101…</td>
</tr>
<tr>
<td>1/10 (bonus)</td>
<td>0.1</td>
<td>0.00011001100…</td>
</tr>
</tbody>
</table>

Conceptual goal: How can we represent real numbers with a **fixed** number of bits?  
**Learning goal:** Appreciate the IEEE floating point format!
Approximating real numbers

How can we represent real numbers with a *fixed* number of bits?

In the world of real numbers:

- The real number line extends forever (infinite *range*).
- Real numbers have infinite resolution (infinite *precision*).

In the base-2 world of computers, we must *approximate*:

- Each variable type is fixed width (*float*: 32 bits, *double*: 64 bits).
- Compromises are inevitable (*range* and *precision*). Like with *int*, we need to make choices about which numbers make the cut and which don’t.
Thought experiment: Fixed point

Base 10, decimal case:

5934.216123121..._{10}

Base 2, binary case:

1011.0101010101..._{2}

- Decide on fixed granularity, e.g., 1/16
- Assign bits to represent powers from $2^{-3}$ to $2^{-4}$
Thought experiment: Fixed point

Strategy evaluation

What values can be represented?
• Largest magnitude? Smallest? To what precision?

How hard to implement?
• How to convert int into 32-bit fixed point? 32-bit fixed point to int?
• Can existing integer ops (add, multiply, shift) be repurposed?

How well does this meet our needs?

Base 2, binary case:

1011.0101010101…₂

1 0 1 1 . 0 1 0 1 1

• Decide on fixed granularity, e.g., 1/16
• Assign bits to represent powers from 2³ to 2⁻⁴
The problem with fixed point

**Problem:** We must fix where the decimal point is in our representation. This fixes our *precision*.

\[
6.022 \times 10^{23} = 11 \ldots 0.0 \quad (\text{79 bits})
\]

\[
6.626 \times 10^{-34} = 0.0 \ldots 01 \quad (\text{113 bits})
\]

To store both these numbers in the same fixed-point representation, the bit width of the type would need to be at least 192 bits wide!
We have a need for relative rather than absolute precision.

- How much error/approximation is tolerable? Radius of atom, bowling ball, planet?

Consider for decimal values:

\[ 3,650,123 \rightarrow 3.65 \times 10^6 \]

\[ 0.0000072491 \rightarrow 7.25 \times 10^{-6} \]

- As a datatype, store mantissa and exponent separately
- Division of digits into exponent/mantissa determines range and precision
IEEE floating point

IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
- Supported by all major systems today
  Hardware: specialized co-processor vs. integrated into main chip

Driven by numerical concerns

- Behavior defined in mathematical terms
- Clear standards for rounding, overflow, underflow
- Support for transcendental functions (roots, trig, exponentials, logs)
- Hard to make fast in hardware

Numerical analysts predominated over hardware designers in defining standard

I think that it is nice to have at least one example—and the floating-point standard is one—where sleaze did not triumph.

— Will Kahan
(chief architect of standard)
Plan For Today

• Representing real numbers and fixed point
• Floating Point: Normalized values
• Break: Announcements
• Floating Point: Special/denormalized values
• Number representations in C
• Floating point arithmetic
Let's aim to represent numbers of the following scientific-notation-like format:

\[ V = (-1)^s \times M \times 2^E \]

- **Sign bit, \( s \):**
  - negative (\( s = 1 \))
  - positive (\( s = 0 \))

- **Mantissa, \( M \):**
  - Significant digits, also called significand

- **Exponent, \( E \):**
  - Scales value by (possibly negative) power of 2
IEEE Floating Point

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

$$V = (-1)^s \times M \times 2^E$$

⚠ Quirky! Exponent and fraction are not encoded as $M$ and $E$ in 2’s complement!
What is the number represented by the following 32-bit float?

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000 0000</td>
<td>0100 0000 0000 0000 0000 0000 000</td>
</tr>
</tbody>
</table>
## Exponent

### exponent (8 bits)

<table>
<thead>
<tr>
<th>exponent (Binary)</th>
<th>$E$ (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11111110</td>
<td>127</td>
</tr>
<tr>
<td>11111101</td>
<td>126</td>
</tr>
<tr>
<td>11111100</td>
<td>125</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>00000011</td>
<td>-124</td>
</tr>
<tr>
<td>00000010</td>
<td>-125</td>
</tr>
<tr>
<td>00000001</td>
<td>-126</td>
</tr>
<tr>
<td>00000000</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### fraction (23 bits)

- **Special**
- **Normalized**
- **Denormalized**
## Exponent: Normalized values

<table>
<thead>
<tr>
<th>exponent (Binary)</th>
<th>$E$ (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11111110</td>
<td>127</td>
</tr>
<tr>
<td>11111101</td>
<td>126</td>
</tr>
<tr>
<td>11111100</td>
<td>125</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>00000011</td>
<td>-124</td>
</tr>
<tr>
<td>00000010</td>
<td>-125</td>
</tr>
<tr>
<td>00000001</td>
<td>-126</td>
</tr>
<tr>
<td>00000000</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Based on this table, how do we compute an exponent from a binary value?

Why would this be a good idea? (hint: what if we wanted to compare two floats with $>$, $<$, $=$?)
Exponent: Normalized values

Not 2’s complement

$$E = \text{exponent} - \text{bias},$$ where float bias = $$2^{8-1} - 1 = 127$$

Exponents are sequentially represented starting from 000...1 (most negative) to 111...10 (most positive).

Bit-level comparison is fast!

<table>
<thead>
<tr>
<th>exponent (Binary)</th>
<th>$E$ (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11111110</td>
<td>127</td>
</tr>
<tr>
<td>11111101</td>
<td>126</td>
</tr>
<tr>
<td>11111100</td>
<td>125</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>00000011</td>
<td>-124</td>
</tr>
<tr>
<td>000000010</td>
<td>-125</td>
</tr>
<tr>
<td>00000001</td>
<td>-126</td>
</tr>
<tr>
<td>000000000</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
We could just encode whatever $M$ is in the fraction field ($f$). But there’s a trick we can use to make the most out of the bits we have...

\[ M = 1. [\text{fraction bits}] \]
Correct scientific notation:
In the mantissa, always keep one non-zero digit to the left of the decimal point.

For base 10:

\[42.4 \times 10^5 \rightarrow 4.24 \times 10^6\]
\[324.5 \times 10^5 \rightarrow 3.245 \times 10^7\]
\[0.624 \times 10^5 \rightarrow 6.24 \times 10^4\]

For base 2:

\[10.1 \times 2^5 \rightarrow 1.01 \times 2^6\]
\[1011.1 \times 2^5 \rightarrow 1.0111 \times 2^8\]
\[0.110 \times 2^5 \rightarrow 1.10 \times 2^4\]

Observation: in base 2, this means there is always a 1 to the left of the decimal point!
We could just encode whatever $M$ is in the fraction field ($f$). But there’s a trick we can use to make the most out of the bits we have...

\[ M = 1. \text{[fraction bits]} \]

- An “implied leading 1” representation
- Means: we get one additional bit of precision for free!

Thanks, Will!
What is the number represented by the following 32-bit float?

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000 0000</td>
<td>0100 0000 0000 0000 0000 000</td>
</tr>
</tbody>
</table>

Subtract bias \((2^{8-1} - 1 = 127)\)

\[
E = 128 - 127 = 1
\]

\[
M = (1.01)_2 = 1 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1.25 \quad \text{(base 2)}
\]

Add implicit 1

\[
V = (-1)^0 \times 1.25 \times 2^1 = 2.5
\]
## Practice #1

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0111 1110</td>
<td>0000 0000 0000 0000 0000 000</td>
</tr>
</tbody>
</table>

1. Is this number:  
   A. Greater than 0?  
   B. Less than 0?  

2. Is this number:  
   A. Less than -1?  
   B. Between -1 and 1?  
   C. Greater than 1?  

3. Bonus: What is the number?
Practice #1

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0111 1110</td>
<td>0000 0000 0000 0000 0000 000</td>
</tr>
</tbody>
</table>

1. Is this number:  
A. Greater than 0?  
B. Less than 0?

2. Is this number:  
A. Less than -1?  
B. Between -1 and 1?  
C. Greater than 1?

3. Bonus: What is the number?  
   \[1.0 \times 2^{-1} = 0.5\]
Plan For Today

- Representing real numbers and fixed point
- Floating Point: Normalized values
- **Break:** Announcements
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic
The midterm exam is **Fri. 2/14 12:30PM-2:20PM in Hewlett 200.**

- Covers material through **lab4/assign4** (no floats or assembly language)
- Closed-book, 1 2-sided page of notes permitted, C reference sheet provided

Administered via BlueBook software (on your laptop)

- Practice materials and BlueBook download available on course website
- If you have academic (e.g. OAE) or athletics accommodations, please let us know by **Sunday 2/9** if possible.
- If you do not have a workable laptop for the exam, you **must** let us know by **Sunday 2/9**. Limited charging outlets will be available for those who need them.
Joke break

https://www.smbc-comics.com/comic/2013-06-05

Slightly off from the real float 0.3 😊

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

- Representing real numbers and fixed point
- Floating Point: Normalized values
- **Break:** Announcements
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic
### Reserved Exponent Values

<table>
<thead>
<tr>
<th>Exponent (Binary)</th>
<th>E (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11111110</td>
<td>127</td>
</tr>
<tr>
<td>11111101</td>
<td>126</td>
</tr>
<tr>
<td>11111100</td>
<td>125</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>00000011</td>
<td>-124</td>
</tr>
<tr>
<td>00000010</td>
<td>-125</td>
</tr>
<tr>
<td>00000001</td>
<td>-126</td>
</tr>
<tr>
<td>00000000</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Legend:**
- **special**
- **normalized**
- **denormalized**
Reserved exponent values: All zeros

Zero (+0, -0)

<table>
<thead>
<tr>
<th></th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>0000 0000</td>
<td>all zeros</td>
</tr>
</tbody>
</table>

Denormalized floats:

<table>
<thead>
<tr>
<th></th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>0000 0000</td>
<td>any nonzero</td>
</tr>
</tbody>
</table>

- Smallest normalized exponent: $E = 1 - \text{bias} = -126$
- Mantissa has no leading zero: $M = 0$. [fraction bits]

$v = (-1)^s \times M \times 2^E$

Why would two zeros be okay?

Why would we want so much precision for tiny numbers?
Reserved exponent values: All zeros

Zero (+0, -0)

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>0000 0000</td>
<td>all zeros</td>
</tr>
</tbody>
</table>

Denormalized floats:

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>0000 0000</td>
<td>any nonzero</td>
</tr>
</tbody>
</table>

- Smallest normalized exponent: $E = 1 - \text{bias} = -126$
- Mantissa has no leading zero: $M = 0$. $V = (-1)^s \times M \times 2^E$
Reserved exponent values: All ones

Infinity (+\text{inf}, -\text{inf})

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>1111 1111</td>
<td>all zeros</td>
</tr>
</tbody>
</table>

Why would we want to represent infinity?

Not a number (\text{NaN}):

Computation result that is an invalid mathematical real number.

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>1111 1111</td>
<td>any nonzero</td>
</tr>
</tbody>
</table>

What kind of mathematical computation would result in a non-real number? (hint: square root)
Reserved exponent values: All ones

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>1111 1111</td>
<td>all zeros</td>
</tr>
</tbody>
</table>

Infinity (+\(\text{inf}\), -\(\text{inf}\))

- floats have built-in handling of overflow: infinity + anything = infinity

Not a number (\(\text{NaN}\)):

- Computation result that is an invalid mathematical real number.

- Examples: \(\sqrt{x}\) (i.e., \(\frac{2}{\sqrt{x}}\)), where \(x\) is negative, \(\frac{\infty}{\infty}\), \(\infty + (\text{-\infty})\), etc.
Questions?
Plan For Today

• Representing real numbers and fixed point
• Floating Point: Normalized values
• **Break:** Announcements
• Floating Point: Special/denormalized values
• **Number representations in C**
• Floating point arithmetic
We said that it’s not possible to represent \textit{all} real numbers using a fixed-width representation. What does this look like?

**Float Converter**
- [https://www.h-schmidt.net/FloatConverter/IEEE754.html](https://www.h-schmidt.net/FloatConverter/IEEE754.html)

**Floats and Graphics**
- [https://www.shadertoy.com/view/4tVyDK](https://www.shadertoy.com/view/4tVyDK)
**float and double**

**float** (32 bits)
- 8-bit exponent ranges from -126 to +127, \(2^{127} = 10^{37}\)

<table>
<thead>
<tr>
<th>S</th>
<th>exponent (8 bits)</th>
<th>Fraction (23 bits)</th>
</tr>
</thead>
</table>

**double** (64 bits)
- 11-bit exponent ranges from -1022 to +1023, \(2^{1023} = 10^{308}\)

<table>
<thead>
<tr>
<th>S</th>
<th>exponent (11 bits)</th>
<th>Fraction (52 bits)</th>
</tr>
</thead>
</table>
float and int

32-bit integer (type \texttt{int}): 
−2,147,483,648 to 2,147,483,647

64-bit integer (type \texttt{long}): 

32-bit floating point (type \texttt{float}): 
\(\sim 1.7 \times 10^{-38}\) to \(\sim 3.4 \times 10^{38}\) (+ negative range)

64-bit floating point (type \texttt{double}): 
\(\sim 9 \times 10^{-307}\) to \(\sim 1.8 \times 10^{308}\) (+ negative range)

(normalized float/double ranges)

All integers in these ranges can be represented.

Not all numbers can be represented. Gaps can get quite large: larger the exponent, larger the gap between successive fraction values.
Plan For Today

• Representing real numbers and fixed point
• Floating Point: Normalized values
• **Break:** Announcements
• Floating Point: Special/denormalized values
• Number representations in C
• Floating point arithmetic
Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)
But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often unwise.
Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative. 
(a + b) equals (b + a)  
But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often unwise.
Lisa’s Official Guide To Making Money

It’s easy!

FAST!

You can lose money, too!
Demo: Float Arithmetic

Try it yourself:
./bank 100 1        # deposit
./bank 100 -1       # withdraw
./bank 100000000 -1 # make bank
./bank 16777216 1   # lose bank

Why is $2^{24}$ special?
Introducing “Minifloat”

For a more compact example representation, we will use an 8 bit “minifloat” with a 4 bit exponent, 3 bit fraction and bias of 7 (note: minifloat is just for example purposes, and is not a real datatype).

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (4 bits)</th>
<th>fraction (3 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7 6</td>
<td>3 2</td>
</tr>
</tbody>
</table>
In minifloat, with a balance of $128, a deposit of $4 would not be recorded at Lisa’s Bank. Why not?

\[
\begin{array}{c}
128 \\
+ \phantom{0}4 \\
\hline \\
128? \\
\end{array}
\]

Let’s step through the calculations to add these two numbers (note: this is just for understanding; real float calculations are more efficient).
Floating Point Arithmetic

128 \[ 0 \quad 1110 \quad 000 \]
+ 4 \[ 0 \quad 1001 \quad 000 \]

Floating Point Arithmetic (we won’t go into details):

- Manipulate significand and exponents independently
- Compute exact result \((x \text{ op } y)\)
- Round and put in floating point

\[ 128.00 + 4.00 = 132.00 \]

Exact result: 132

(Next slide)
### Practice #2

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (4 bits)</th>
<th>fraction (3 bits)</th>
</tr>
</thead>
</table>

What is 132 as a minifloat?

\[
V = (-1)^s \times M \times 2^E
\]

\[
E = \text{exponent} - \text{bias}
\]

\[
M = 1.\text{[fraction bits]}
\]

\[
\text{bias} = 2^{4-1} - 1 = 7
\]

\[
V = (-1)^s \times M \times 2^{E - 2^4-1 + 1}
\]

\[
V = (-1)^s \times M \times 2^7
\]

\[
V = (-1)^s \times 1.???? \times 2^7
\]

\[
V = (-1)^s \times 1011 \times 2^7
\]

\[
V = (-1)^s \times 1011 \times 128
\]

\[
V = (-1)^s \times 134 \times 2
\]

\[
V = (-1)^s \times 268
\]

\[
V = (-1)^s \times 268
\]

\[
V = (-1)^s \times 268
\]
What is 132 as a minifloat?

1. Convert to binary.
2. Convert to scientific notation \( M \times 2^E \).
3. Determine exponent = \( E + \text{bias} \)
4. Determine fraction, with rounding.
5. Determine sign \( s \)

\[
V = (-1)^s \times M \times 2^E
\]

\[
E = \text{exponent} - \text{bias}
\]

\[
M = 1.\ [\text{fraction bits}]
\]

\[
\text{bias} = 2^{4-1} - 1 = 7
\]

\[
V = (-1)^s \times (1.0000100) \times 2^7
\]

\[
(1000 \ 0100)_2 \times (1.0000100)_2 \times 2^7
\]

\[
7 + 7 = 14 = (1110)_2
\]

\[
(1.0000100)_2
\]

\[
(1.0000100)_2
\]

\[
0 \ (\text{positive})
\]
Approximation error is inevitable

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0</td>
<td>1110</td>
<td>000</td>
</tr>
<tr>
<td>+ 4</td>
<td>0</td>
<td>1001</td>
<td>000</td>
</tr>
<tr>
<td>132?</td>
<td>0</td>
<td>1110</td>
<td>000</td>
</tr>
</tbody>
</table>

We didn’t have enough bits to differentiate between 128 and 132.
Approximation error is inevitable

We didn’t have enough bits to differentiate between 128 and 132.

Another way to corroborate this: the *next-largest minifloat* that can be represented after 128 is 144. 132 isn’t representable!

**Key Idea:** the smallest float hop increase we can take is to increase the fractional component by 1.
Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)
But (a + b) + c may not equal a + (b + c)

Approximation and rounding is inevitable.

Equality comparison operations are often inaccurate.
Floating Point Arithmetic

Is this just overflowing? It turns out it’s more subtle.

```c
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
```

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

- 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
Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

\[(a + b)\] equals \[(b + a)\]

But \[(a + b) + c\] may not equal \[a + (b + c)\]

Equality comparison operations are often inaccurate.
Demo: Float Equality

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

- [http://geocar.sdf1.org/numbers.html](http://geocar.sdf1.org/numbers.html)
Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)
But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often unwise inaccurate.
What would be nice to have in a real number representation?

- Represent widest range of numbers possible
- Flexible “floating” decimal point
- Still be able to compare quickly
- Represent scientific notation numbers, e.g. $1.2 \times 10^6$
- Have more predictable overflow behavior