CS107, Lecture 10
Floating Point

Reading: B&O 2.4
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

• Recap: Generics with Function Pointers
• Representing real numbers
• Fixed Point
• Break: Announcements
• Floating Point
Plan For Today

• Recap: Generics with Function Pointers
• Representing real numbers
• Fixed Point
• Break: Announcements
• Floating Point
Function Pointers

• In C, there is a variable type for functions!
• We can pass functions as parameters, store functions in variables, etc.
• Why is this useful?
Generics Limitations

Sometimes, there is functionality that *cannot* be made generic.

```c
void bubble_sort(void *arr, int n, int elem_size_bytes) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (curr_elem should come before prev_elem) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```
Generics Limitations

Sometimes, there is functionality that cannot be made generic. The caller can pass in a function to perform that functionality for the data they are providing.

```c
void bubble_sort(void *arr, int n, int elem_size_bytes,
                 bool (*cmp_fn)(const void *, const void *)) {
    while (true) {
        bool swapped = false;
        for (int i = 1; i < n; i++) {
            void *prev_elem = (char *)arr + (i-1)*elem_size_bytes;
            void *curr_elem = (char *)arr + i*elem_size_bytes;
            if (cmp_fn(prev_elem, curr_elem) > 0)) {
                swapped = true;
                swap(prev_elem, curr_elem, elem_size_bytes);
            }
        }
        if (!swapped) {
            return;
        }
    }
}
```
Generic C Standard Library Functions

- **qsort** – I can sort an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to sort.

- **bsearch** – I can use binary search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

- **lfind** – I can use linear search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

- **lsearch** - I can use linear search to search for a key in an array of any type! I will also add the key for you if I can’t find it. In order to do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
Generic C Standard Library Functions

• **scandir** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether or not you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering of two given directory entries.
Here’s the variable type syntax for a function:

```
[return type] (*[name])([[parameters]])
```
Function Pointers

```c
int do_something(char *str) {
    ...
}

int main(int argc, char *argv[]) {
    ...
    int (*func_var)(char *) = do_something;
    ...
    func_var("testing");
    return 0;
}
```
void bubble_sort(void *arr, int n, int elem_size_bytes,
        int (*cmp_fn)(const void *, const void *)) {

...}

int cmp_double(const void *, const void *) {...}

int main(int argc, char *argv[]) {

    ...
    double values[] = {1.2, 3.5, 12.2};
    int n = sizeof(values) / sizeof(values[0]);
    bubble_sort(values, n, sizeof(*values), cmp_double);

    ...}
Comparison Functions

• Comparison functions are a common use of function parameters, because many generic functions must know how to compare elements of your type.

• Comparison functions always take *pointers to the data they care about*, since the data could be any size!

When writing a comparison function callback, use the following pattern:
1) Cast the void *argument(s) and set typed pointers equal to them.
2) Dereference the typed pointer(s) to access the values.
3) Perform the necessary operation.

(steps 1 and 2 can often be combined into a single step)
Comparison Functions

• It should return:
  • < 0 if first value should come before second value
  • > 0 if first value should come after second value
  • 0 if first value and second value are equivalent

• This is the same return value format as strcmp!

```
int (*compare_fn)(const void *a, const void *b)
```
int integer_compare(void *ptr1, void *ptr2) {
    // cast arguments to int *s and dereference
    int num1 = *(int *)ptr1;
    int num2 = *(int *)ptr2;

    // perform operation
    return num1 - num2;
}

... qsort(mynums, count, sizeof(*mynums), integer_compare);
int string_compare(void *ptr1, void *ptr2) {
  // cast arguments and dereference
  char *str1 = *(char **)ptr1;
  char *str2 = *(char **)ptr2;

  // perform operation
  return strcmp(str1, str2);
}

... qsort(mysorts, count, sizeof(*mysorts), string_compare);
Generics Wrap-Up

• We use void * pointers and memory operations like `memcpy` and `memmove` to make data operations generic.

• We use function pointers to make logic/functionality operations operations generic.
**memset**

**memset** is a function that sets a specified amount of bytes at one address to a certain value.

```c
void *memset(void *s, int c, size_t n);
```

It fills n bytes starting at memory location `s` with the byte `c`. (It also returns `s`).

```c
int counts[5];
memset(counts, 0, 3);  // zero out first 3 bytes at counts
memset(counts + 3, 0xff, 2)  // set last 2 bytes to 0xff
```
Plan For Today

• **Recap**: Generics with Function Pointers
• Representing real numbers
• Fixed Point
• **Break**: Announcements
• Floating Point
We previously discussed representing integer numbers using two’s complement.

However, this system does not represent real numbers such as $\frac{3}{5}$ or 0.25.

How can we design a representation for real numbers?
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.*
Problem: every number base has un-representable real numbers.

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

Base 2: $1/10_{10} = 0.000110011001100110011..._2$

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:** Like in base 10, let’s add binary decimal places to our existing number representation.
Plan For Today

• Recap: Generics with Function Pointers
• Representing real numbers
• Fixed Point
• Break: Announcements
• Floating Point
Fixed Point

• **Idea:** 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.
**Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

\[
\begin{array}{ccccccc}
. & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
1/2s & 1/4s & 1/8s & \ldots \\
\end{array}
\]

\[
\begin{array}{ccccccc}
1 & 0 & 1 & 1 & 0 & . & 1 & 1 \\
16s & 8s & 4s & 2s & 1s & 1/2s & 1/4s \\
\end{array}
\]
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. $1.2 \times 10^6$
- Still be able to compare quickly
- Have more predictable over/under-flow behavior
Plan For Today

• **Recap:** Generics with Function Pointers
• Representing real numbers
• Fixed Point

• **Break:** Announcements
• Floating Point
Announcements

• Functions like `versionsort` and `alphasort` prohibited on assign4
• Brown Institute “Magic Grants” application open! See brown.stanford.edu
• CURIS undergraduate summer research applications open! See curis.stanford.edu. Due 2/10.
Midterm Exam

• The midterm exam is **Fri. 2/15 12:30-2:20PM in Hewlett 200 and Hewlett 201**
  • Last names A-G: Hewlett 201
  • Last Names H-Z: 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 2/10** if possible.
• If you do not have a workable laptop for the exam, you **must** let us know by **Sunday 2/10**. Limited charging outlets will be available for those who need them.
Plan For Today

• Recap: Generics with Function Pointers
• Representing real numbers
• Fixed Point
• Break: Announcements
• Floating Point
Let’s Get Real

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 x 10^6
• Still be able to compare quickly
• Have more predictable over/under-flow behavior
Let’s aim to represent numbers of the following scientific-notation-like format:

\[ x \times 2^y \]

With this format, 32-bit floats represent numbers in the range \(-3.4E+38\) to \(3.4E+38\)! Is every number between those representable? No.
IEEE Single Precision Floating Point

31 30 23 22 0

s exponent (8 bits) fraction (23 bits)

Sign bit (0 = positive)

\[ x \times 2^y \]
## Exponent

<table>
<thead>
<tr>
<th>Exponent (Binary)</th>
<th>Exponent (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>?</td>
</tr>
<tr>
<td>00000001</td>
<td>?</td>
</tr>
<tr>
<td>00000010</td>
<td>?</td>
</tr>
<tr>
<td>00000011</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td>?</td>
</tr>
<tr>
<td>11111100</td>
<td>?</td>
</tr>
<tr>
<td>11111101</td>
<td>?</td>
</tr>
<tr>
<td>11111110</td>
<td>?</td>
</tr>
<tr>
<td>11111111</td>
<td>?</td>
</tr>
</tbody>
</table>
## Exponent

<table>
<thead>
<tr>
<th>Exponent (Binary)</th>
<th>Exponent (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00000001</td>
<td>?</td>
</tr>
<tr>
<td>00000010</td>
<td>?</td>
</tr>
<tr>
<td>00000011</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td>?</td>
</tr>
<tr>
<td>11111100</td>
<td>?</td>
</tr>
<tr>
<td>11111101</td>
<td>?</td>
</tr>
<tr>
<td>11111110</td>
<td>?</td>
</tr>
<tr>
<td>11111111</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
### Exponent

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

• **Actual value = binary value – 127**

<table>
<thead>
<tr>
<th>Binary</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>1 – 127 = -126</td>
</tr>
<tr>
<td>00000010</td>
<td>2 – 127 = -125</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11111101</td>
<td>253 – 127 = 126</td>
</tr>
<tr>
<td>11111110</td>
<td>254 – 127 = 127</td>
</tr>
</tbody>
</table>
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

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

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

Observation: in base 2, this means there is always a 1 to the left of the decimal point!
• We can adjust this value to fit the format described previously. Then, \( x \) will always be in the format \( 1.xxx\ldots \).

• 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

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

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?
Skipping Numbers

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

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

• https://twitter.com/D_M_Gregory/status/1044008750162604032
Let’s Get Real

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. $1.2 \times 10^6$
• Still be able to compare quickly
• Have more predictable over/under-flow behavior
Representing Zero

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

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>All zeros</td>
<td>All zeros</td>
</tr>
</tbody>
</table>

• This means there are two representations for zero! 😞
Representing Small Numbers

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

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>All zeros</td>
<td>Any</td>
</tr>
</tbody>
</table>

• 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.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>All ones</td>
<td>All zeros</td>
</tr>
</tbody>
</table>

• The sign bit indicates whether it is positive or negative infinity.
• Floats have built-in handling of over/underflow!
  • Infinity + anything = infinity
  • Negative infinity + negative anything = negative infinity
  • Etc.
Representing Exceptional Values

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

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any nonzero</td>
</tr>
</tbody>
</table>

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

• 32-bit integer (type `int`):
  › -2,147,483,648 to 2,147,483,647
  › Every integer in that range can be represented

• 64-bit integer (type `long`):
  › -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

• 32-bit floating point (type `float`):
  • ~1.7 x10^{-38} to ~3.4 x10^{38}
  • Not all numbers in the range can be represented (obviously—uncountable)
  • 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 x10^{-308} to ~2 x10^{308}
Floating Point Arithmetic

You might be thinking: oh, this is just overflowing. But it is more subtle than that.

```c
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %f\n", (a + b) - b);
printf("3.14 + (1e20 - 1e20) = %f\n", a + (b - b));
```

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

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14:</td>
<td>0</td>
<td>10000000</td>
<td>100100011110101111000011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1e20:</td>
<td>0</td>
<td>11000001</td>
<td>010110101111000111101100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Floating Point Arithmetic

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000000</td>
<td></td>
<td>10010001111010111000011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11000001</td>
<td></td>
<td>01011010111100011101100</td>
<td></td>
</tr>
</tbody>
</table>

You cannot simply add the two significands together, you have to align their binary points. If we wanted to add the decimal values, it would look like this:

\[
3.14 + 1000000000000000000000.00 = 10000000000000000000003.14
\]

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

Step 1: convert from base 10 to binary

What is 10000000000000000003.14 in binary? Let’s find out!


101011011100001101011110001011010110011001000000000000000000011.0010001111010111000010100011…
Step 2: find most significant 1 and take the next 23 digits for the fractional component, rounding if needed.

1 0101101011110001110110010110101100100000000000000011.001000111101011100001010011...
Step 3: find how many places we need to shift left to put the number in 1.xxx format. This fills in the exponent component.

1010110101111000111010111100101101011000110001011010110001100010000000000000000011.001000111101011100001010011…

66 shifts -&gt; 66 + 127 = 193
Floating Point Arithmetic

Step 4: if the sign is positive, the sign bit is 0. Otherwise, it’s 1.

Sign bit is 0.
So, we are left with the following for 1000000000000000000000000.14 decimal:

```
  31  30  23  22
   0 11000001 01011010111100011101100
```

Let's compare this to 1e20 that we had before:

```
  31  30  23  22
   0 11000001 01011010111100011101100
```

**Identical!** We didn't have enough bits to differentiate between 1e20 and 1000000000000000000000000.14
Floating Point Arithmetic

Back to our original example:

```c
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %f\n", (a + b) - b);
printf("3.14 + (1e20 - 1e20) = %f\n", a + (b - b));
```

```
$ ./floatMultiTest
(3.14 + 1e20) - 1e20 = 0.000000
3.14 + (1e20 - 1e20) = 3.140000
```

Clearly, \texttt{1e20 - 1e20} will produce 0 (no need to shift the binary points). What this really means is that \textbf{floating point arithmetic is not associative}. In other words, the order of operations matters.
Floating Point Arithmetic

Here is another example:

```c
int main()
{
    double a = 0.1;
    double b = 0.2;
    double c = 0.3;
    double d = a + b;
    printf("0.1 + 0.2 == 0.3 ? %s\n", a + b == c ? "true" : "false");
    return 0;
}
```

$ ./floatEquality
0.1 + 0.2 == 0.3 ? false
The rounding that happens during the calculation of 0.1 + 0.2 produces a different number than 0.3!
Floating Point Arithmetic

- http://geocar.sdf1.org/numbers.html
Let’s Get Real

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. $1.2 \times 10^6$
• Still be able to compare quickly
• Have more predictable over/under-flow behavior
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.
Recap

• **Recap:** Generics with Function Pointers
• Representing real numbers
• Fixed Point
• **Break:** Announcements
• Floating Point

Next time: assembly language