This Week

Monday: A Note on Assignment 2 and Preparation for Assignment 3

1. Final Thoughts on Floating-point Numbers: Round-off Errors
2. Pointers
3. Dynamic Memory

Wednesday: Intro to Assignment 3

4. Operator Precedence and Supporting Sets
5. Sorting Algorithms

Friday: Help with All Assignments

6. Debugging Using LLDB

This Week

Monday: A Note on Assignment 2 and Preparation for Assignment 3

1. Final Thoughts on Floating-point Numbers: Round-off Errors
2. Pointers
3. Dynamic Memory

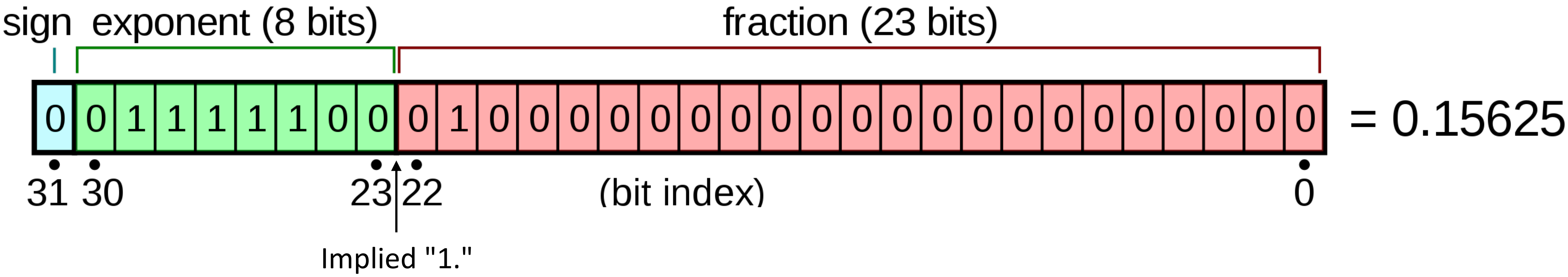
Wednesday: Intro to Assignment 3

4. Operator Precedence and Supporting Sets
5. Sorting Algorithms

Friday: Help with All Assignments

6. Debugging Using LLDB

Floating Point Numbers are Binary

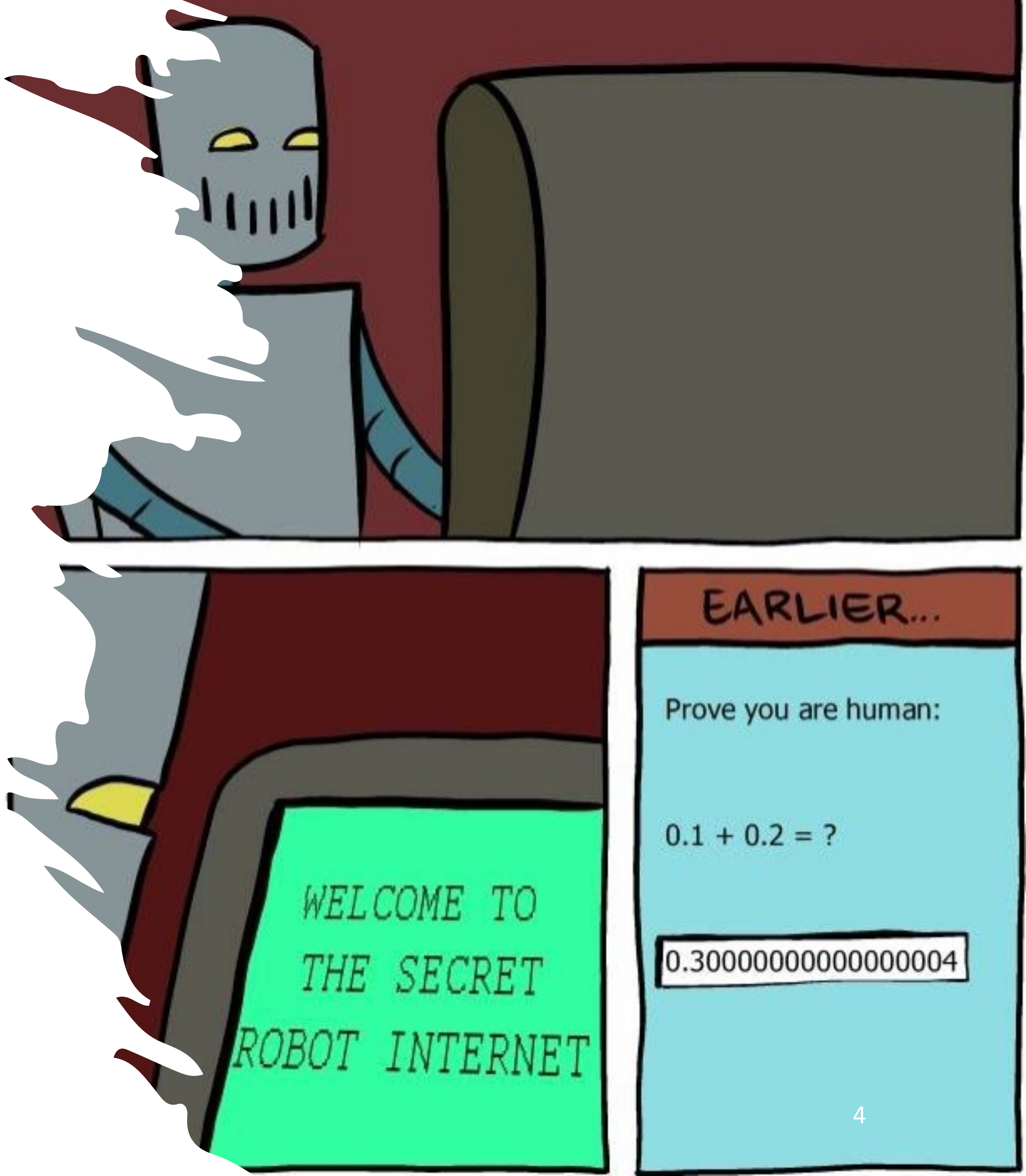


$$(-1)^{b_{31}} \times 2^{(b_{30} \ldots b_{23}) - 127} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

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

Floating-point Arithmetic

- Normal mathematics:
  - Real numbers are exact.
- Computer arithmetic:
  - Numbers are approximated.
- Floating-point arithmetic results in rounding errors.
  - Calculations can require rounding to fit into a finite representation.
  - Results of floating-point arithmetic are also approximations.



Round-Off Errors

- Errors occur when doing any calculation with floating-point.
  - Because rounding occurs to fit values into a finite representation.
  - Floating point $\neq \mathbb{Q}$
- IEEE 754 defines 5 standard rounding modes:
  - Round to nearest, ties rounded to nearest value with even significant digit (most common).
  - Round to nearest, ties rounded to value furthest from zero.
  - Round up (ceiling)
  - Round down (floor)
  - Round to zero (truncation)

Round-Off Errors

```c
#include <stdio.h>

int main(void) {
    // Loop with d from 0.0 to 9.9 stepping by 0.1
    //
    //                      vvvv--- What's this 9.95 ???
    for (double d = 0.0; d <= 9.95; d += 0.1) {
        // ...
    }

    return 0;
}
```

Round-off Errors in `for` Loops

Want to loop through range [0, 10) with steps of 0.1

	Source	Internal Representation
• First value	0.0	0.00000000000000000000000000000
• Step value	0.1	0.10000000000000000555111512...
• Step 100x		9.99999999999998046007476659...
• Last value	10.0	10.00000000000000000000000000000

Some Dangers of Floating-point Arithmetic

- Equality comparisons often can fail

  In Assignment 2, your test of `sqrt_newton()` should do this:

```
for (double d = 0; d <= 9.95; ++d) ...
```

- Why?
  - Choose midway between the last desired value and the next one

9.90 ← Last desired value

9.95 ← Check for < at the midway point to next value

10.00 ← Don't check at the exact upper limit!

Round-Off Errors

```c
#include <stdio.h>

int main(void) {
    // Loop with d from 0.0 to 9.9 stepping by 0.1
    //
    //                              vvvv--- What's this 9.95 ???
    for (double d = 0.0; d <= 9.95; d += 0.1) {
        // Run a test with 100 floating-point equality comparisons.

        if (d == 0.0) printf("d is 0.0\n");
        if (d == 0.1) printf("d is 0.1\n");
        if (d == 0.2) printf("d is 0.2\n");
        if (d == 0.3) printf("d is 0.3\n");
        if (d == 0.4) printf("d is 0.4\n");
        if (d == 0.5) printf("d is 0.5\n");
        if (d == 0.6) printf("d is 0.6\n");
        if (d == 0.7) printf("d is 0.7\n");
        if (d == 0.8) printf("d is 0.8\n");
        if (d == 0.9) printf("d is 0.9\n");

        if (d == 1.0) printf("d is 1.0\n");
        if (d == 1.1) printf("d is 1.1\n");
        if (d == 1.2) printf("d is 1.2\n");
```

Round-Off Errors

```
veenstra@arm128:~/s23/13s-cse/L13$ ./fp1
d is 0.0
d is 0.1
d is 0.2
d is 0.4
d is 0.5
d is 0.6
d is 0.7
d is 1.2
d is 1.3
d is 4.4
d is 4.5
d is 4.6
veenstra@arm128:~/s23/13s-cse/L13$
```

Some Dangers of Floating-point Arithmetic

- Common numbers cannot be represented exactly

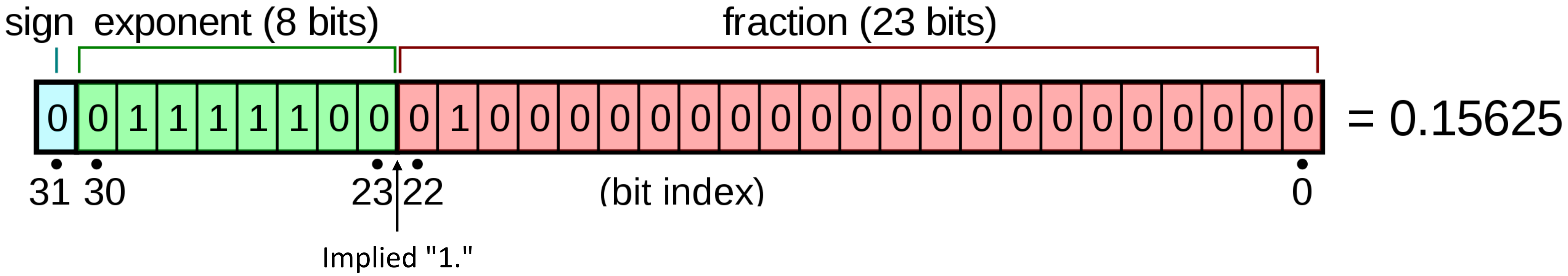
  0.1 cannot be represented exactly in a `float` or a `double` !

Cannot Represent 0.1 or 0.01 Exactly!

```
veenstra@arm128:~/s23/13s-cse/L13$ ./roundoff
float  0.01 is 0.009999999776482582092285156250000000000000000000000000
double 0.01 is 0.010000000000000000208166817117216851329430937767028808593750000

float  0.1  is 0.100000001490116119384765625000000000000000000000000000000
double 0.1  is 0.100000000000000005551115123125782702118158340454101562500000000
```

Floating Point Numbers are <u>Binary</u>



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

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

float 0.1

```
float  0.1  is 0.100000001490116119384765625
0x3dcccccd:
sign is +
exponent contributes 2 to the -4
fraction is
  1
+ 1 / 2         (0.500000000000000000000000000)
+ 1 / 16        (0.062500000000000000000000000)
+ 1 / 32        (0.031250000000000000000000000)
+ 1 / 256       (0.003906250000000000000000000)
+ 1 / 512       (0.001953125000000000000000000)
+ 1 / 4096      (0.000244140625000000000000000)
+ 1 / 8192      (0.000122070312500000000000000)
+ 1 / 65536     (0.000015258789062500000000000)
+ 1 / 131072    (0.000007629394531250000000000)
+ 1 / 1048576   (0.000000953674316406250000000)
+ 1 / 2097152   (0.000000476837158203125000000)
+ 1 / 8388608   (0.000000119209289550781250000)
```

Some Dangers of Floating-point Arithmetic

- Addition and subtraction
  - Adding numbers of very different magnitudes can result in the digits of the number of smaller magnitude rounded away.
  - Subtracting numbers of very different magnitudes can also result in precision error.
    - Truncation errors can occur when shifting the mantissa.

Example: Adding Different Magnitudes (10 digits)

$$
\begin{array}{r}
1.234\,567\,890 \times 10^{+9} \\
+\ \underline{1.234\,567\,890 \times 10^{-7}} \\
1.234\,567\,890 \times 10^{+9}
\end{array}
\qquad
\begin{array}{rr}
 & 1{,}234{,}567{,}890\ .\ 000\ 000\ 000\ 000\ 000\ 000 \\
+ & \underline{0\ .\ 000\ 000\ 123\ 456\ 789\ 000} \\
 & \boxed{1{,}234{,}567{,}890}\ .\ 000\ 000\ \cancel{123\ 456\ 789\ 0}00
\end{array}
$$

1. Align the columns of the *addends*
2. Add
3. Round to 10 significant digits
4. When one addend is too small
    - Rounded sum == larger addend!