Hi there! Welcome back to section, we're happy that you're here 😊

**Goals of Floating Point**

Representation should include: [1] a large range of values (both very small and very large numbers), [2] a high amount of precision, and [3] real arithmetic results (e.g. $\infty$ and NaN).

**IEEE 754 Floating Point Standard**

The value of a real number can be represented in scientific binary notation as:

$$\text{Value} = (-1)^{\text{sign}} \times \text{Mantissa}_2 \times 2^{\text{Exponent}} = (-1)^s \times 1.M_2 \times 2^{E-bias}$$

The binary representation for floating point values uses three fields:

1. **S**: encodes the sign of the number (0 for positive, 1 for negative)
2. **E**: encodes the exponent in biased notation with a bias of $2^{w-1}-1$
3. **M**: encodes the mantissa (or significand, or fraction) – stores the fractional portion, but does not include the implicit leading 1.

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 bit</td>
<td>23 bits</td>
</tr>
<tr>
<td>0</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>0</td>
<td>23 bits</td>
<td>23 bits</td>
</tr>
<tr>
<td>0</td>
<td>52 bits</td>
<td>23 bits</td>
</tr>
</tbody>
</table>

How a float is interpreted depends on the values in the exponent and mantissa fields:

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>anything</td>
<td>denormalized number (denorm)</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>normalized number</td>
</tr>
<tr>
<td>255</td>
<td>zero</td>
<td>infinity ($\infty$)</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>not-a-number (NaN)</td>
</tr>
</tbody>
</table>

**Exercises:**

1. What is the largest, finite, positive value that can be stored using a float?
   
   \[ S = 0, E = 254, M = 0b1...1. \]
   
   Value is then \(1.1...1 \times 2^{127} = (2 - 2^{-23}) \times 2^{127} = 2^{128} - 2^{104}.\)

2. What is the smallest, positive, normalized value that can be stored using float?
   
   \[ S = 0, E = 1, M = 0b0...0. \]
   
   Value is then \(1.0 \times 2^{-126} = 2^{-126}.\)

3. Convert the decimal number 1.25 into single precision floating point representation:
   
   \[ 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 \]
   
   \(1.25 = 2^0 + 2^{-2} = 1.012 \times 2^0.\)
   
   \(S = 0, E = 0 + \text{bias} = 127 = 0b 0111 1111, M = 0b 010...0.\)

4. What are the decimal values of the following floats?
   
   \[ \begin{align*}
   0x80000000 & : -0 \\
   0xFF94BEEF & : \text{NaN} \\
   0x41180000 & : +9.5 \\
   0x41180000 = 0b 0|100 0001 0|001 1000 0...0. \\
   & S = 0, E = 128+2 = 130 \rightarrow \text{Exponent} = E - \text{bias} = 3, \text{Mantissa} = 1.00112 \\
   & 1.00112 \times 2^3 = 1001.12 = 8 + 1 + 0.5 = 9.5
   \end{align*} \]
Floating Point Mathematical Properties

- Not associative: \((2 + 2^{50}) - 2^{50} \neq 2 + (2^{50} - 2^{50})\)
- Not distributive: \(100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2\)
- Not cumulative: \(2^{25} + 1 + 1 + 1 \neq 2^{25} + 4\)

Exercises:

5. Based on floating point representation, explain why each of the three statements above occurs.

- **Associative:** Only 23 bits of mantissa, so \(2 + 2^{50} = 2^{50}\) (2 gets rounded off). So LHS = 0, RHS = 2.
- **Distributive:** 0.1 and 0.2 have infinite representations in binary point \((0.2 = 0.00110011...)_2\), so the LHS and RHS suffer from different amounts of rounding (try it!).
- **Cumulative:** 1 is 25 powers of 2 away from \(2^{25}\), so \(2^{25} + 1 = 2^{25}\), but 4 is 23 powers of 2 away from \(2^{25}\), so it doesn’t get rounded off.

6. If \(x\) and \(y\) are variable type `float`, give two different reasons why \((x+2*y) - y == x+y\) might evaluate to false.

   - **(1)** Rounding error: like what is seen in the examples above.
   - **(2)** Overflow: if \(x\) and \(y\) are large enough, then \(x+2*y\) may result in infinity when \(x+y\) does not.

The GNU Debugger (GDB)

The GNU Debugger is a powerful debugging tool that will be critical to Lab 2 and Lab 3 and is a useful tool to know as a programmer moving forward. There are tutorials and reference sheets available on the course webpage, but the following tutorial should get you started with the basics:

GDB Tutorial:

1) Download calculator.c from the class webpage:
   ```bash
   > wget https://courses.cs.washington.edu/courses/cse410/17wi/sections/03/calculator.c
   ```

2) Compile the file with debugging symbols (\(-g\) flag):
   ```bash
   > gcc -Wall -std=gnu99 -g calculator.c -o calculator
   ```

3) Load the binary (executable) into GDB. This will spit out a bunch of information (e.g. version, license).
   ```bash
   > gdb calculator
   ```

4) Inside of GDB, use the run command (run or just r) to execute your program. By default, this will continue until an error is encountered or your program exits.
   a. Command-line arguments can be passed as additional arguments to run:
      ```bash
      (gdb) run 3 4 +
      ```
   b. To step through the program starting at `main()` instead, use the start command (start or just sta):
      ```bash
      (gdb) start
      ```

5) To view source code (C) while debugging, use the list command (list or just l).
   a. You can give list a function name ("list <function>") to look at the beginning of a function.
      ```bash
      (gdb) list main
      ```
   b. You can give list a line number ("list <line>") to look at the lines around that line number, or provide a specific range ("list <start>, <end>").
      ```bash
      (gdb) list 45
      ```
c. "list" will display the next 10 lines of code after whatever was last displayed and "list -" will display the previous 10 lines of code before whatever was last displayed.

6) Create breakpoints using the break command (break or b)
   a. A breakpoint will stop program execution before the shown instruction has been executed!
   b. You can create a breakpoint at a function name, source code line number, or assembly instruction address. The following all break at the same place:
      (gdb) break main
      (gdb) break 34
      (gdb) break *0x4005cc
   c. Each break point has an associated number. You can view your breakpoints using the info command (info or just i) and then enable (enable or just en) or disable (disable or just dis) specific ones.
      (gdb) info break
      (gdb) disable 3
      (gdb) enable 3

7) Navigating source code within GDB is done while program execution is started (run or start), but halted (e.g. at a breakpoint).
   a. Use the next command (next or just n) to execute the next # of lines of code and then break again. This will complete ("step over") any function calls found in the lines of code.
      (gdb) next
      (gdb) next 4
   b. Use the step command (step or just s) to execute the next # of lines of code and then break again. This will step into any function calls found in the lines of code.
      (gdb) step
      (gdb) step 4
   c. Use the finish command (finish or just fin) to step out of the current function call.
   d. Use the continue command (continue or just c) to resume continuous program execution (until next breakpoint is reached or your program terminates).

8) You can print the current value of variables or expressions using the print command (print or just p):
   a. The print command can take an optional format specifier: /x (hex), /d (decimal), /u (unsigned), /t (binary), /c (char), /f (float)
      (gdb) print /t argc
      (gdb) print /x argv
      (gdb) print /d argv*2+5
   b. The display command (display or just disp) is similar, but causes the expression to print in the specified format every time the program stops.

9) You can terminate the current program run using the kill command (kill or just k). This will allow you to restart execution (run or start) with your breakpoints intact.

10) You can exit GDB by either typing Ctrl-D or using the quit command (quit or just q)