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:

$$
Value = (-1)^{sign} \times Mantissa_2 \times 2^{Exp} = (-1)^s \times 1.frac_2 \times 2^{exp-bias}
$$

The binary representation for floating point values uses three fields:

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

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>1 bit</td>
<td>8 bits</td>
</tr>
<tr>
<td>double</td>
<td>1 bit</td>
<td>11 bits</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>exp</th>
<th>frac</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>zero</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?

2. What is the smallest, positive, normalized value that can be stored using float?

3. Convert the decimal number 1.25 into single precision floating point representation:

4. What are the decimal values of the following floats?
   - 0x80000000 0xFF94BEF 0x41180000
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.

6. If \(x\) and \(y\) are variable type \textit{float}, give two \textit{different} reasons why \((x + 2\times y) - y \neq x + y\) might evaluate to false.

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 \texttt{calculator.c} from the class webpage:
   
   \[
   \texttt{wget https://courses.cs.washington.edu/courses/cse351/17sp/sections/03/calculator.c}
   \]

2) Compile the file \textit{with debugging symbols} (-g flag):
   
   \[
   \texttt{gcc -Wall -std=gnu99 -g calculator.c -o calculator}
   \]

3) Load the binary (executable) into GDB. This will spit out a bunch of information (\textit{e.g.} version, license).
   
   \[
   \texttt{gdb calculator}
   \]

4) Inside of GDB, use the run command (\texttt{run} or just \texttt{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 \texttt{run}:
      
      \[
      \texttt{(gdb) run 3 4 +}
      \]
   b. To step through the program starting at \texttt{main()} instead, use the start command (\texttt{start} or just \texttt{sta}):
      
      \[
      \texttt{(gdb) start}
      \]

5) To view source code (C) while debugging, use the \texttt{list} command (\texttt{list} or just \texttt{l}).
   a. You can give list a function name ("list <function>") to look at the beginning of a function.
      
      \[
      \texttt{(gd) list main}
      \]
   b. You can give list a line number ("list <line>") to look at the lines \textit{around} that line number, or provide a specific range ("list <start>, <end>").
      
      \[
      \texttt{(gdb) list 45}
      \texttt{(gdb) list 10, 15}
      \]
   c. ”\texttt{list}” will display the next 10 lines of code \textit{after} whatever was last displayed and ”\texttt{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 argc*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``)