Floating Point Numbers

- C has three floating point types
  - `float` ... typically 32-bit (lower precision, narrower range)
  - `double` ... typically 64-bit (higher precision, wider range)
  - `long double` ... typically 128-bits (but maybe only 80 bits used)

- Floating point constants, e.g.: `3.14159` `1.0e-9` are `double`

- Reminder: division of 2 ints in C yields an int.
  - but division of double and int in C yields a double.

---

text出自：https://www.cse.unsw.edu.au/~cs1521/22T1/
Floating Point Numbers

- if we represent floating point numbers with a fixed small number of bits
  - there are only a finite number of bit patterns
  - can only represent a finite subset of reals
- almost all real values will have no exact representation
- value of arithmetic operations may be real with no exact representation
- we must use closest value which can be exactly represented
- this approximation introduces an error into our calculations
- often, does not matter
- sometimes ... can be disastrous

Fixed Point Representation

- can have fractional numbers in other bases, e.g.: $110.101_2 = 6.625_{10}$
- could represent fractional numbers similarly to integers by assuming decimal point is in fixed position
- for example with 32 bits:
  - 16 bits could be used for integer part
  - 16 bits could be used for the fraction
  - equivalent to storing values as integers after multiplying (scaling) by $2^{16}$
  - major limitation is only small range of values can be represented
    - minimum $2^{-16} \approx 0.000015$
    - maximum $2^{15} \approx 32768$
- usable for some problems, but not ideal
- used on small embedded processors without silicon floating point

floating_types.c - print characteristics of floating point types

```c
float f;
double d;
long double l;
printf("float %2lu bytes min=%-12g max=%g\n", sizeof f, FLT_MIN, FLT_MAX);
printf("double %2lu bytes min=%-12g max=%g\n", sizeof d, DBL_MIN, DBL_MAX);
printf("long double %2lu bytes min=%-12Lg max=%Lg\n", sizeof l, LDBL_MIN, LDBL_MAX);
```

```
$ ./floating_types
float  4 bytes min=1.17549e-38  max=3.40282e+38
double 8 bytes min=2.22507e-308  max=1.79769e+308
long double 16 bytes min=3.3621e-4932  max=1.18973e+4932
```
IEEE 754 standard

- C floats almost always IEEE 754 single precision (binary32)
- C double almost always IEEE 754 double precision (binary64)
- C long double might be IEEE 754 (binary128)
- IEEE 754 representation has 3 parts: sign, fraction and exponent
- numbers have form \( \text{sign} \times \text{fraction} \times 2^{\text{exponent}} \), where sign is +/-.
- fraction always has 1 digit before decimal point (normalized)
  - as a consequence only 1 representation for any value
- exponent is stored as positive number by adding constant value (bias)
- numbers close to zero have higher precision (more accurate)

### Floating Point Numbers

Example of normalising the fraction part in binary:
- \( 1010.1011 \) is normalized as \( 1.0101011 \times 2^{011} \)
- \( 1010.1011 = 10 + 11/16 = 10.6875 \)
- \( 1.0101011 \times 2^{011} = (1 + 43/128) \times 2^3 = 1.3359375 \times 8 = 10.6875 \)

The normalised fraction part always has 1 before the decimal point.

Example of determining the exponent in binary:
- if exponent is 8-bits, then the bias = \( 2^{8-1} - 1 = 127 \)
- valid bit patterns for exponent 00000000 .. 11111110
- correspond to \( B \) exponent values -126 .. 127

### Internal structure of floating point values

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent (exp)</th>
<th>Fraction (fraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent (exp)</th>
<th>Fraction (fraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11 bits</td>
<td>52 bits</td>
</tr>
</tbody>
</table>
IEEE-754 Single Precision example: **0.15625**

0.15625 is represented in IEEE-754 single-precision by these bits:
```
00111110100000000000000000000000
```

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0111100</td>
<td>01000000000000000000000000000000</td>
</tr>
</tbody>
</table>

- **sign bit = 0**
- **sign = +**
- **raw exponent = 0111100** binary
  - = 124 decimal
- **actual exponent = 124 - exponent_bias**
  - = 124 - 127
  - = -3
- **number = +1.01000000000000000000000 binary * 2**⁻³
  - = 1.25 decimal * 2⁻³
  - = 1.25 * 0.125
  - = 0.15625

source code for explain_float_representation.c

$ ./explain_float_representation -0.125

-0.125 is represented as a float (IEEE-754 single-precision) by these bits:
```
10111110000000000000000000000000
```

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0111100</td>
<td>00000000000000000000000000000000</td>
</tr>
</tbody>
</table>

- **sign bit = 1**
- **sign = -**
- **raw exponent = 0111100** binary
  - = 124 decimal
- **actual exponent = 124 - exponent_bias**
  - = 124 - 127
  - = -3
- **number = -1.00000000000000000000000 binary * 2**⁻³
  - = -1 decimal * 2⁻³
  - = -1 * 0.125
  - = -0.125

IEEE-754 Single Precision example: **150.75**

$ ./explain_float_representation 150.75

150.75 is represented in IEEE-754 single-precision by these bits:
```
01000011000101101100000000000000
```

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000110</td>
<td>00101101100000000000000000000000</td>
</tr>
</tbody>
</table>

- **sign bit = 0**
- **sign = +**
- **raw exponent = 1000110** binary
  - = 134 decimal
- **actual exponent = 134 - exponent_bias**
  - = 134 - 127
  - = 7
- **number = +1.0010110110000000000000000 binary * 2**⁷
  - = 1.17773 decimal * 2⁷
  - = 1.17773 * 128
  - = 150.75
IEEE-754 Single Precision example: \(-96.125\)

\[
\text{\$ ./explain_float_representation -96.125} \\
-96.125 \text{ is represented in IEEE-754 single-precision by these bits:} \\
11000001000000000000000000000000 \\
\begin{array}{c|c|c}
\text{sign} & \text{exponent} & \text{fraction} \\
1 & 1000101 & 100000001000000000000000 \\
\end{array} \\
\text{sign bit} = 1 \\
\text{sign} = - \\
\text{raw exponent} = 1000101 \text{ binary} = 133 \text{ decimal} \\
\text{actual exponent} = 133 - \text{exponent_bias} = 133 - 127 = 6 \\
\text{number} = -1.10000000100000000000000 \text{ binary } \times 2^{6} \\
= -1.50195 \text{ decimal } \times 2^{6} \\
= -1.50195 \times 64 \\
= -96.125
\]

IEEE-754 Single Precision exploring bit patterns #1

\[
\text{\$ ./explain_float_representation 00111101110011001100110011001101} \\
\text{sign bit} = 0 \\
\text{sign} = + \\
\text{raw exponent} = 0111011 \text{ binary} = 123 \text{ decimal} \\
\text{actual exponent} = 123 - \text{exponent_bias} = 123 - 127 = -4 \\
\text{number} = +1.10011001100110011001101 \text{ binary } \times 2^{-4} \\
= 1.6 \text{ decimal } \times 2^{-4} \\
= 1.6 \times 0.0625 \\
= 0.1
\]

infinity.c: exploring infinity

- IEEE 754 has a representation for +/- infinity
- propagates sensibly through calculations

```c
double x = 1.0/0.0;
printf("%lf\n", x); // prints inf
printf("%lf\n", -x); // prints -inf
printf("%lf\n", x - 1); // prints inf
printf("%lf\n", 2 * atan(x)); // prints 3.141593
printf("%d\n", 42 < x); // prints 1 (true)
printf("%d\n", x == INFINITY); // prints 1 (true)
```

source code for infinity.c
C (IEEE-754) has a representation for invalid results:

- NaN (not a number)
- Ensures errors propagate sensibly through calculations

```c
double x = 0.0/0.0;
printf("%lf\n", x); // prints nan
printf("%lf\n", x - 1); // prints nan
printf("%d\n", x == x); // prints 0 (false)
printf("%d\n", isnan(x)); // prints 1 (true)
```

IEEE-754 Single Precision example: inf

```
$ ./explain_float_representation inf
inf is represented in IEEE-754 single-precision by these bits:
01111111000000000000000000000000
sign | exponent | fraction
 0 | 11111111 | 00000000000000000000000000000000
sign bit = 0
sign = +
raw exponent = 11111111 binary
= 255 decimal
number = +inf
```

IEEE-754 Single Precision exploring bit patterns #2

```
$ ./explain_float_representation 01111111110000000000000000000000
sign bit = 0
sign = +
raw exponent = 11111111 binary
= 255 decimal
number = NaN
```

Source code for `nan.c`

[https://www.cse.unsw.edu.au/~cs1521/22T1/](https://www.cse.unsw.edu.au/~cs1521/22T1/)
double a, b;
  a = 0.1;
  b = 1 - (a + a + a + a + a + a + a + ... check if values are close
https://www.cse.unsw.edu.au/~cs1521/22T1/ COMP1521 22T1 — Floating-Point Numbers 19 / 24

Another reason not to use == with floating point values

if (d == d) {
  printf("d == d is true\n");
} else {
  // will be executed if d is a NaN
  printf("d == d is not true\n");
}
if (d == d + 1) {
  // may be executed if d is large
  // because closest possible representation for d + 1
  // is also closest possible representation for d
  printf("d == d + 1 is true\n");
} else {
  printf("d == d + 1 is false\n");
}

source code for double_not_always.c
Another reason not to use `==` with floating point values

```bash
$ dcc double_not_always.c -o double_not_always
$ ./double_not_always 42.3
d = 42.3
d == d is true
d == d + 1 is false
$ ./double_not_always 4200000000000000000
  d = 4.2e+18
  d == d is true
  d == d + 1 is true
$ ./double_not_always NaN
  d = nan
  d == d is not true
  d == d + 1 is false
```

because closest possible representation for `d + 1` is also closest possible representation for `d`

source code for `double_not_always.c`

https://www.cse.unsw.edu.au/~cs1521/22T1/

Consequences of most reals not having exact representations

```c
// loop looks to print 10 numbers but actually never terminates
double d = 9007199254740990;
while (d < 9007199254741000) {
    printf("%lf\n", d); // always prints 9007199254740992.000000
    // 9007199254740993 can not be represented as a double
    // closest double is 9007199254740992.0
    // so 9007199254740992.0 + 1 = 9007199254740992.0
    d = d + 1;
}
```

source code for `double_disaster.c`

9007199254740993 is the smallest integer which can not be represented exactly as a double

The closest double to 9007199254740993 is 9007199254740992.0

aside: 9007199254740993 can not be represented by a int32_t

it can be represented by int64_t

Exercise: Floating point → Decimal

Convert the following floating point numbers to decimal.

Assume that they are in IEEE 754 single-precision format.

0 10000000 11000000000000000000000

1 01111110 10000000000000000000000