COMP1521 21T3 — Floating-Point Numbers

https://www.cse.unsw.edu.au/~cs1521/21T3/
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.
double d = 4/7.0;
// prints in decimal with (default) 6 decimal places
printf("\%lf\n", d);   // prints 0.571429
// prints in scientific notation
printf("\%le\n", d);    // prints 5.714286e-01
// picks best of decimal and scientific notation
printf("\%lg\n", d);    // prints 0.571429
// prints in decimal with 9 decimal places
printf("\%.9lf\n", d);  // prints 0.571428571
// prints in decimal with 1 decimal place and field width of 5
printf("\%10.1lf\n", d); // prints 0.6

source code for float_output.c
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
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 $sign \times fraction \times 2^{exponent}$, 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)
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 00000001 .. 11111110
- correspond to $B$ exponent values -126 .. 127
Floating Point Numbers

Internal structure of floating point values

**Single Precision**

- **Sign**: 1 bit
- **Exponent**: 8 bits
- **Fraction**: 23 bits

**Double Precision**

- **Sign**: 1 bit
- **Exponent**: 11 bits
- **Fraction**: 52 bits
0.15625 is represented in IEEE-754 single-precision by these bits:
00111110001000000000000000000000

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0111100</td>
<td>01000000000000000000000</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>01111100</td>
<td>00000000000000000000000</td>
</tr>
</tbody>
</table>

sign bit = 1
sign = -

raw exponent = 01111100 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
$ ./explain_float_representation 150.75
150.75 is represented in IEEE-754 single-precision by these bits:
01000011000101101100000000000000
sign | exponent | fraction
0 | 10000110 | 001011011000000000000000
sign bit = 0
sign = +
raw exponent = 10000110 binary
= 134 decimal
actual exponent = 134 - exponent_bias
= 134 - 127
= 7
number = +1.001011011000000000000000 binary * 2**7
= 1.17773 decimal * 2**7
= 1.17773 * 128
= 150.75
$ ./explain_float_representation -96.125

-96.125 is represented in IEEE-754 single-precision by these bits:
110000101100000001000000000000000

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000101</td>
<td>100000010000000000000000</td>
</tr>
</tbody>
</table>

sign bit = 1
sign = -

raw exponent = 10000101 binary
= 133 decimal

actual exponent = 133 - exponent_bias
= 133 - 127
= 6

number = -1.1000000010000000000000000000000 binary * 2**6
= -1.50195 decimal * 2**6
= -1.50195 * 64
= -96.125
$ ./explain_float_representation 00111101110011001100110011001101
sign bit = 0
sign = +
raw exponent = 01111011 binary
= 123 decimal
actual exponent = 123 - exponent_bias
= 123 - 127
= -4
number = +1.10011001100110011001101 binary * 2**-4
= 1.6 decimal * 2**-4
= 1.6 * 0.0625
= 0.1
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 propagates sensibly through calculations

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)
$ ./explain_float_representation inf
inf is represented in IEEE-754 single-precision by these bits:
01111111110000000000000000000000

sign | exponent | fraction
    0 | 11111111  | 00000000000000000000000

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 explain_float_representation.c
double a, b;
a = 0.1;
b = 1 - (a + a + a + a + a + a + a + a + a + a);
if (b != 0) {
    // better would be fabs(b) > 0.000001
    printf("1 != 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1\n");
}
printf("b = %g\n", b); // prints 1.11022e-16

- do not use == and != with floating point values
- instead check if values are close
Consequences of most reals not having exact representations

double x = 0.000000011;
double y = (1 - cos(x)) / (x * x);
// correct answer y = ~0.5
// prints y = 0.917540
printf("y = %lf\n", y);
// division of similar approximate value
// produces large error
// sometimes called catastrophic cancellation
printf("%g\n", 1 - cos(x)); // prints 1.11022e-16
printf("%g\n", x * x); // prints 1.21e-16

source code for double_catastrophe.c

https://www.cse.unsw.edu.au/~cs1521/21T3/
Another reason not to use == with floating point values

```c
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 420000000000000000
  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/~cs1521/21T3/
Consequences of most reals not having exact representations

```c
double d = 9007199254740992;
// loop never terminates
while (d < 9007199254740999) {
    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;
}
```

- 9007199254740993 is $2^{53} + 1$
  it is 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
Convert the following floating point numbers to decimal.
Assume that they are in IEEE 754 single-precision format.

0 10000000 11000000000000000000000

1 01111110 10000000000000000000000