COMP1521 24T3 — Floating-Point Numbers

https://www.cse.unsw.edu.au/~cs1521/24T3/

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.

Floating Point Number - Output

```c
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

Fractions in different Bases

The decimal fraction 0.75 means

- $7*10^{-1} + 5*10^{-2} = 0.7 + 0.05 = 0.75$
- or equivalently $75/10^2 = 75/100 = 0.75$

Similary 0b0.11 means

- $1*2^{-1} + 1*2^{-2} = 0.5 + 0.25 = 0.75$
- or equivalently $3/2^2 = 3/4 = 0.75$

Similarly 0x0.C means

- $12*16^{-1} = 0.75$
- or equivalently $12/16^1 = 3/4 = 0.75$

Note: We call the **.** a radix point rather than a decimal point when we are dealing with other bases.

Fractions in different Bases

The algorithm to convert a decimal fraction to another base is:

- take the fractional component and multiply by the base
- the whole number becomes the next digit to the right of the radix point in our fraction.
- repeat this process until the fractional part becomes exhausted or we have sufficient digits
- this process is not guaranteed to terminate.

Converting Decimal Fractions to Binary

For example if we want to convert 0.3125 to base 2

- 0.3125 * 2 = **0**.625
- 0.625 * 2 = **1**.25
- 0.25 * 2 = **0**.5
- 0.5 * 2 = **1**.0

Therefore 0.3125 = 0b0.0101

Exercise 2: Fractions: Decimal → Binary

Convert the following decimal values into binary

- 12.625
- 0.1

Floating Point Numbers

- can have fractional numbers in other bases, e.g.: $110.101_2 == 6.625_{10}$
- 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 disasterous

Fixed-Point Representation

- fixed-point is a simple trick to represent fractional numbers as integers
  - every value is multiplied by a particular constant, e.g. 1000 and stored as integer
  - so if constant is 1000, could represent 56.125 as an integer (56125)
  - but not 3.141592
- usable for some problems, but not ideal
- used on small embedded processors without silicon floating point
- major limitation is only small range of values can be represented
  - for example with 32 bits, and using 65536 (2^{16}) as constant
    - 16 bits used for integer part
    - 16 bits used for the fraction
  - minimum $2^{-16} \approx 0.000015$
  - maximum $2^{15} \approx 32768$

exponentional representation - a better approach

- you met scientific notation, e.g 6.0221515 * 10^23 in physics or other science classes
- we can represent numbers on a computer in a similar way to scientific notation
- but using binary instead of base ten, e.g 10.6875

  =1010.1011 = $1.0101011 * 2^{11_2}$ = $(1 + 43/128) * 2^3$ = $1.3359375 * 8$ = 10.6875
- allows a much bigger range of values to be represented than fixed point
- using only 8 bits for the exponent, we can represent numbers from 10^{-38} .. 10^{+38}
- using only 11 bits for the exponent, we can represent numbers from 10^{-308} .. 10^{+308}
- leads to numbers close to zero having higher precision (more accurate) which is good

choosing which exponentional representation

- exponent notation allows multiple representations for a single value
  - e.g $1.0101011 * 2^{11_2}$ == 10.6875 and $10.101011 * 2^{10_2}$ == 10.6875
- having multiple representations would make implementing arithmetic slower on CPU
- better to have only one representation (one bit pattern) representing a value
- decision - use representation with exactly one digit in front of decimal point
  - use $1.0101011 * 2^{11_2}$ not $10.101011 * 2^{10_2}$ or $1010.1011 * 2^{0_2}$
  - this is called normalization
- weird hack: as we are using binary the first digit must be a one we don't need to store it
  - as we long we have a separate representation for zero

floating_types.c - print characteristics of floating point types

```
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);
```

source code for floating_types.c

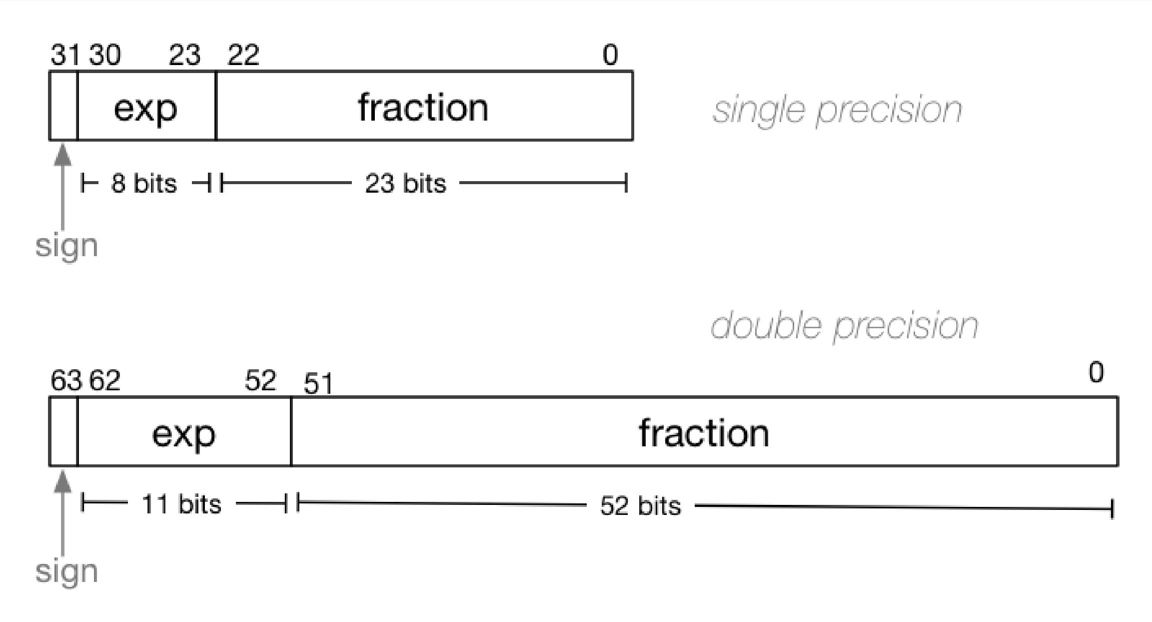
```
$ ./floating_types
float        4 bytes  min=1.17549e-38   max=3.40282e+38
double       8 bytes  min=2.22507e-308  max=1.79769e+308
long double 16 bytes  min=3.3621e-4932  max=1.18973e+4932
```

- 1970s Intel building microprocessors (single-chip CPUs)
- 1976 Intel developing coprocessor (separate chip) for floating-point arithmetic
- Intel asked William Kahan, University of California to design format
- other manufacturers didn't want to be left out
- IEEE 754 standard working group formed
- Kahan and others produced well-designed robust specification
- accepted by manufacturers who begin using it for new architectures
- IEEE 754 standard released in 1985 (update to standard in 2008)
- today, almost all computers use IEEE 754

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 $sign\ fraction * 2^{exponent}$, where $sign$ is +/-
- ***fraction*** always has 1 digit before decimal point (***normalized***)
- ***exponent*** is stored as positive number by adding constant value (***bias***)

Internal structure of floating point values



Floating Point Numbers

Example of normalising the fraction part in binary:

- 1010.1011 is normalized as $1.0101011 * 2^{011}$
- $1010.1011 = 10 + 11/16 = 10.6875$
- $1.0101011 * 2^{011} = (1 + 43/128) * 2^3 = 1.3359375 * 8 = 10.6875$

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 are `00000001` .. `11111110`
- these correspond to exponent values of -126 .. 127

Floating Point Numbers

Example (single-precision):

$150.75 = 10010110.11$

`// normalise fraction, compute exponent`

$= 1.001011011 \times 2^7$

`// sign bit = 0`

`// exponent = 10000110`

`// fraction = 00101101100000000000000`

`= 01000011000101101100000000000000`

Distribution of Floating Point Numbers

- floating point numbers not evenly distributed
- representations get further apart as values get bigger
  - this works well for most calculations
  - but can cause weird bugs
- double (IEEE 754 64 bit) has 52-bit fractions so:
  - between 2^n and 2^{n+1} there are 2^{52} doubles evenly spaced
    - e.g. in the interval 2^{-42} and 2^{-43} there are 2^{52} doubles
    - and in the interval between 1 and 2 there are 2^{52} doubles
    - and in the interval between 2^{42} and 2^{43} there are 2^{52}
  - near 0.001 - doubles are about 0.0000000000000000002 apart
  - near 1000 - doubles are about 0.0000000000002 apart
  - near 1000000000000000 - doubles are about 0.25 apart
  - above 2^{53} - doubles are more than 1 apart

IEEE-754 Single Precision example: 0.15625

```
0.15625 is represented in IEEE-754 single-precision by these bits:
00111110001000000000000000000000
sign | exponent | fraction
   0 | 01111100 | 01000000000000000000000
sign bit = 0
sign = +
raw exponent    = 01111100 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

IEEE-754 Single Precision example: -0.125

```
$ ./explain_float_representation -0.125
-0.125 is represented as a float (IEEE-754 single-precision) by these bits:
10111110000000000000000000000000
sign | exponent | fraction
   1 | 01111100 | 00000000000000000000000
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
```


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
sign | exponent | fraction
   0 | 10000110 | 00101101100000000000000
sign bit = 0
sign = +
raw exponent    = 10000110 binary
                = 134 decimal
actual exponent = 134 - exponent_bias
                = 134 - 127
                = 7
number = +1.00101101100000000000000 binary * 2**7
       = 1.17773 decimal * 2**7
       = 1.17773 * 128
       = 150.75
```

IEEE-754 Single Precision example: -96.125

```
$ ./explain_float_representation -96.125
-96.125 is represented in IEEE-754 single-precision by these bits:
11000010110000000100000000000000
sign | exponent | fraction
   1 | 10000101 | 10000000100000000000000
sign bit = 1
sign = -
raw exponent    = 10000101 binary
                = 133 decimal
actual exponent = 133 - exponent_bias
                = 133 - 127
                = 6
number = -1.10000000100000000000000 binary * 2**6
       = -1.50195 decimal * 2**6
       = -1.50195 * 64
       = -96.125
```


IEEE-754 Single Precision exploring bit patterns #1

```
$ ./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
```


infinity.c: exploring infinity

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

```
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

nan.c: handling errors robustly

- 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)
```

source code for nan.c

IEEE-754 Single Precision example: inf

```
$ ./explain_float_representation inf
inf is represented in IEEE-754 single-precision by these bits:
01111111100000000000000000000000
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

Consequences of most reals not having exact representations

```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
```

source code for double_imprecision.c

- do not use **==** and **!=** with floating point values
- instead check if values are close

Consequences of most reals not having exact representations

- The approximate representation of reals can produce unexpected errors.
- If we subtract or divide two values which are very close together a large relative error can result.
- This is called cancellation or catastrophic cancellation.
- For example, if x is close to 0, cos(x) is close to 1
  - calculating 1 - cos(x) can produce a large error in a calculation
- we can avoid the error by replacing 1 - cos(x) with 2 * sin(x/2) * sin(x/2)

Consequences of most reals not having exact representations

```c
double x;
printf("Enter x: ");
scanf("%lf", &x);
printf("(1 - cos(x)) / (x * x) = %lf\n", (1 - cos(x)) / (x * x));
printf("(2 * sin(x/2) * sin(x/2)) / (x * x) = %lf\n", (2 * sin(x/2) * sin(x/2)) / (x * x
```

source code for cancelled.c

```
$ ./a.out
Enter x: 0.123
(1 - cos(x)) / (x * x) = 0.499370
(2 * sin(x/2) * sin(x/2)) / (x * x) = 0.499370
$ ./a.out
Enter x: 0.000000011
(1 - cos(x)) / (x * x) = 0.917540
(2 * sin(x/2) * sin(x/2)) / (x * x) = 0.500000
```

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

```
$ 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

Consequences of most reals not having exact representations

```
// 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 $2^{53} + 1$
  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

Exercise: Floating point $\longrightarrow$ 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
```