Introduction to Programming Advanced 2020 S2

Dr. John Stavrakakis

School of Computer Science, University of Sydney

Copyright Warning

Lecture 2: Data types

Understanding base representation and operators

Bits recap

Representation of data types internally

Every data type starts with bits in memory. Deciding how many bits will determine the number of values that can be represented.

n bit = 2^n values

bits	values
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
...	...
2^{32}	4,294,967,296
...	...
2^{64}	1.8446744e+19
...	...
2^{128}	3.4028237e+38
...	...
2^{4096}	1.0443888814131525067e+1233

Operations with bits

- `x << y`
  Returns x with the bits shifted to the left by y places (and new bits on the right-hand-side are zeros). This is the same as multiplying x by 2**y.
- `x >> y`
  Returns x with the bits shifted to the right by y places. This is the same as //'ing x by 2**y.
- `x & y`
  Does a "bitwise and". Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0.
- `x | y`
  Does a "bitwise or". Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.

Operations with bits (cont.)

- `~x`
  Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as -x - 1.
- `x ^ y`
  Does a "bitwise exclusive or". Each bit of the output is the same as the corresponding bit in x if that bit in y is 0, and it's the complement of the bit in x if that bit in y is 1.

Why bitwise operators?

IsatM2M is a global, very low data rate service providing a two-way short burst data service for machine-to-machine communications.
Speeds of 10.5 or 25.5 bytes in the send direction and 100 bytes in the receive direction, with latency typically between 30 to 60 seconds.
isatm2m

Compression of information, variable bit length sequences for encoding/decoding.
Vorbis
Golomb-Rice coding

Why bitwise operators? (cont.)

Modern commodity CPU supports addition with similar speed to bitwise operators. However, addition circuitry requires more gates, more power, more time (typically).
Adder circuits

We can optimise operations using base 2 numbers and bitwise operators.

Logic	Optimisation
$x/4$	$x >> 2$
$x * 128$	$x << 7$
$x * 2$	$x << 1$
$x * 15$	$(x << 4) - x$

Using bitwise to support other operators

Create operator for adding 1 to an unsigned number using only bitwise operators: or, and, xor, not, shift, complement and flow control if/while

```
x + 1
```

Start with small cases

Using bitwise to support other operators (cont.)

```
Case 1:
00
01 +
------
01

1st rule = OR

Case 2:
01
01 +
------
10

2nd rule = 10
```

Using bitwise to support other operators (cont.)

```
Case 3:
10
01 +
------
11

3rd rule = OR

Case 4:
11
01 +
------
00

4th rule = 100 (overflow 1)
```

Plus one abstraction

If we can solve the 2 bit addition and reserve the overflow bit, we can propagate this across all the bits of the integer.

Perform an addition operation by scanning from least significant bit to most significant bit. Be sure to keep the overflow bit!

Example input to add one and it's expected result:

```
110101111
000000001
---------------
110110000
```

Plus one abstraction

Write a function rule(a,b) that returns the three bits resulting from the operation $ab + 01$ where a is higher bit and b is lower bit.

Then, write a function `plus_one(x)`, that returns the bits resulting in adding 1. x is a string of 1's and 0's.

There are much better methods than using hard rules as above.

Plus one: Look at the data

Add one to a number (XOR with shift):

```
int m = 1;

// Flip all the set bits
// until we find a 0
while( x & m )
{
    x = x ^ m;
    m <<= 1;
}

// flip the rightmost 0 bit
x = x ^ m;
return x;
```

Consider the case $v_n ... v_3 v_2 v_1 01111 + 0001 \rightarrow$ all bits flip and it does not matter what values v_i are.

Plus one: Look at the data (cont.)

We know that we are adding one bit.

Suppose you have a long string of 1's. When starting at the lower end, we will flip each bit until we reach a 0. So find that location!

Hence we AND (x & m) the 1 bit from the LSB with x, if that is 1, keep searching, but while this is happening, whatever bit x has, it must be one, so it needs to be flipped. So we perform an XOR on the current bit with m, resulting in zero.

Now change m to be the next LSB

Repeat until we find a zero.

Floating point number recap

$$\begin{aligned}110.1001 = \quad & 1\times 2^2 + 1\times 2^1 + 0\times 2^0 + \\ & 1\times 2^{-1} + 0\times 2^{-2} + 0\times 2^{-3} + 1\times 2^{-4}\end{aligned}$$

almost all platforms map Python floats to IEEE-754 "double precision"

IEEE754 double precision: floating-point, 8 bytes, $\approx \pm 2.23\times 10^{-308}$ to $\pm 1.80\times 10^{308}$ - accurate to about 15 digits

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, ...
```

Floating point operators

110.1001 is a conceptual representation

Adding two floats is easy with this form:

```
0110.1001
0010.0101 +
------
1000.1110
```

Floating point operators (cont.)

Revisit adding two numbers in base 10 scientific notation:

1.342×10^2
$4.600 \times 10^{-1}+$

???????

We can realise this calculation as: $134.2 + 0.46 = 134.66$

Notice it is easier to deal with the same exponent:

1.3420×10^2
$0.0046 \times 10^2 \quad +$

1.3466×10^2

Floating point operators (cont.)

When realised as a float, the format is typically:

sign	exponent	mantissa	total
1 bit	11 bits	53 bits	= 64 bits

For addition to be supported with floats, we need to change to the same exponent for the addition to take place:

sign	exponent	mantissa
sign	exponent	mantissa +

All in base 2, one of the exponent bits have to be *shifted* to match. We can then perform an integer style addition (when sign bit is the same).

Floating point operators (cont.)

Suppose we have floating point: operand1 + operand2

1. Let a be the smaller of the two operands i.e. min(operand1, operand2)
2. Let b be the other operand
3. Let e_a be exponent of operand a
4. Let e_b be exponent of operand b
5. Change exponent of operand a by $e_a - e_b$
6. Recall all floats are now written in normalised form $1.\{0,1\}^m$
7. Perform integer addition of $a + b$ (mantissa of each operand)
8. Return to the normalised form of $1.\{0,1\}^m$
9. Round the result because there may not be enough bits

Rounding numbers

Almost all binary operaters will promote to the datatype with the most precision. $\mathit{float} \times \mathit{int} = \mathit{float}$

We can see that $\mathit{float} + \mathit{float}$ requires rounding. A design decision about how to round is needed.

Rounding numbers (cont.)

Let's start with a simpler problem.

If we have floating point numbers to convert to an integer. How is it done? rather, how should it be done?

0.0	0
0.1	?
0.2	?
0.3	?
0.4	?
0.5	???
0.6	?
0.7	?
0.8	?
0.9	?
1.0	1

Rounding is approximating a value. There is no correct method of rounding because we are *always* losing information.

Rounding numbers (cont.)

- Rounding down toward negative infinity,
- Rounding up toward positive infinity,
- Rounding toward zero,
- Rounding away from zero (toward ±infinity),
- Rounding to nearest integer,
- Rounding to even/odd,
- Rounding randomly up/down with probability 50%,
- Rounding stochastically *The probability of rounding x to $\lfloor x \rfloor$ is proportional to the proximity of x to $\lfloor x \rfloor$* [1]

$$\text{Round}(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1 - (x - \lfloor x \rfloor) \\ \lfloor x \rfloor + 1 & \text{with probability } x - \lfloor x \rfloor \end{cases}$$

Rounding numbers (cont.)

Round to nearest integer and if it is exactly halfway, round half to even.

Bankers Rounding is an algorithm for rounding quantities to integers, in which numbers which are equidistant from the two nearest integers are rounded to the *nearest even integer*. Thus, 0.5 rounds down to 0; 1.5 rounds up to 2. A similar algorithm can be constructed for rounding to other sets besides the integers (in particular, sets which a constant interval between adjacent members).[2]

Rounding numbers (cont.)

input	rounded	reason
0.4	0	nearest 0
0.5	0	tie-breaker, round to even 0
0.6	1	nearest 1
-4.3	-4	nearest -4
-1.5	-2	tie-breaker, round to even -2
-0.3	0	nearest 0
-0.8	-1	nearest -1
17.3	17	nearest 17
17.5	18	tie-breaker, round to even 18
18.5	18	tie-breaker, round to even 18

IEEE 754 uses rounding half to even (remember, it is base 2)

[1] Deep Learning with Limited Numerical Precision
[2] http://wiki.c2.com/?BankersRounding