Bits+Bytes

Thursday, September 26, 2019 2:06 PM

Data representation: Bits, Bytes, and Integers

Everything is bits!!!

Each bit is either a 0 or a 1

By encoding/interpreting sets of bits, computers can exist and do everything we use them for today.

Why use a bit?
Electronic implementation - easy to store with bistable elements. There's probably going to be some noise in the circuits/current, so having only 2 possible states makes it easy to ignore that

Bits let us count in binary - 33 base 10 == 10001 base 2

Decimal fractions:

$1.5213 * 10^4$ in base 10 $==$ $1.1101101101101 * 2^{13}$ in base 2

Byte = 8 bits

Binary: 00000000 to 11111111

Often represented in Hexadecimal 00 to FF, in a base 16 number representation

0xF0 == 240
F in the '16s' place means there's 16 * 15, which is 240

FA1D37B is written in C as
0xfa1d37b

Hexadecimal is useful because it's a compact representation that keeps the association with binary - noted with the 0x notation, 0x3F

Bytes have 8 bits because it's convenient! It's totally arbitrary though. You can fit ASCII representations of all english letters in a singly byte. Possible to think of it as the smallest useful data type.

Datatypes in C have sizes that are multiples of 8

Possible to efficiently represent a set of numbers.

01101001 {0, 3, 5, 6}

76543210

01010101 {0, 2, 4, 6}

C Data Type	64-bit ISA Word Size
Char	1
Short	2
Int	4
Long	8
Float	4
Double	8
Long double	-- / 10 / 16
Pointer	8

(In bytes)

In low-level code, useful to use other stuff - uint8_t for an 8-bit data representation

Bitwise operations

Boolean Algebra - a branch of algebra where all values are either true or false
- 1 is true, 0 is false

And - & - Intersection
Or - | - Union
Xor - ^ - Symmetric Difference
Not - ~ - Complement
All of these are available in C, applicable to any integral data type (long, int, short, char, unsigned)

These apply to any integral data type - long, int, short, char, unsigned

Boolean Bitwise Operations on Bit Vectors

Symbol:	&	|	^	~
Number One:	01101011	01101011	01101011	01101011
Number Two:	10011010	10011010	10011010	
Result:	00001010	11111011	11110001	10010100

Logic Operations in C are NOT THE SAME
&&, ||, !
View 0 as "False"
Anything nonzero as "True"
Always returns 0 or 1

Early termination:
p && *p -> is p is null, this evaluates to false. This prevents segmentation faults

Shift operations:

Left shift: x << y
Shift bit-vector x left y positions
- Throw away the extra bits on left
- Fill with 0's on right

Right shift: x >> y
Shift bit-vector x right y positions
- Throw away extra bits on right
- In case of Logical Shift: fill with 0s on left
- Arithmetic shift: replicate the most signifigant bits on the left

Argument x	01100010
X << 3	00010000
x Log. >> 2	00011000
X Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Encoding Integers:

Unsigned - uses a formula called B2U(x)
- Represented normally, no negative numbers but high max range

Two's Complement - uses a formula called B2T(x)
- The most significant bit of Two's Complement has a negative weight

2 different ways of interpreting the same bit pattern

MSB - Most Signifigant Bit, could also be Most Signifigant Byte sometimes
LSB - Least Signifigant Bit, could also be Least Signifigant Byte sometimes

Mapping between Signed (aka two's complement) and Unsigned: T2U || U2T
When mapping, the bit pattern is maintained!!!
This means that when mapping Two's Complement to Unsigned, the large negative weight becomes a large positive weight instead

Numeric Ranges:
Minimum value / UMin: 0
Maximum value / UMax: $2^w - 1$

W is the number of bits

W	8	16	Etc.
UMax	255	65,535	$2^w - 1$
TMax	127	32,767	$2^{w-1} - 1$
TMin	-128	-32,768	$-(2^{w-1})$

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Signed range

Unsigned range

If you have 2 data types, and 1 is larger htan the other, always convert to the larger one so you don't use information

Unsigned takes precedence over signed

Constants in C are signed by default, if you add a U that specifies they're unsigned

0 -> 0U
42900023U

Explicit casting in C:

```
int tx, xy;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

Can also use implicit casting on assignment:

```
tx = ux;
uy = ty;
```

Floats always take precedence, when comparing this to other datatypes

One's Complement - the value obtained by inverting all the bits of a number

Converting between positive and negative number in Two's Complement is convenient!

All you have to do is invert the bits and add 1

Observation: if you add a number to its inverse, it will always be -1

X	10011101
~X	01100010
+	11111111
	It's -1

Converting a small precision Two's Complement number to a larger number can cause issues - this is called Sign Extension

If the MSB is 1, all new bits to the left are 1
If the MSB is 0, all new bits to the left are 0