Bitoperations

Variables

Lecturers: Mirko Jantschke, Pascal Scholz

4. Dezember 2018

Contents

Bit operations

A little bit of logic

As you know, all data is stored as *binary numbers* - sequences of 0 and 1.
In C, you can operate on this bit layer by using the following *logical bit* and *shift* operators:

Symbol	Operation	Example
|	logical or	0110 | 0101 == 0111
&	logical and	0110 & 0101 == 0100
^	logical xor	0110 ^ 0101 == 0011
~	logical negation	~0110 == 1001
<<	shift to the left	0110 << 2 == 011000
>>	shift to the right	0110 >> 2 == 0001

Computational arithmetics

With bit operations, some mathematical tasks can be solved more efficiently:

- Multiplying/dividing by 2^n is equivalent to a shift by n bits

```
5 * 8 == 5 << 3;
60 / 4 == 60 >> 2;
```

- Instead of % 2^n you can use & 2^{n-1}

```
22 % 2 == 22 & 1;
24 % 16 == 24 & 15;
```

Be aware of the fact that the readability of your code will suffer from that. Most of these optimisations are done by the compiler anyway.

Masking

x | *mask* sets all bits in *x* that are 1 in *mask*.

```
'A' | 32;   /* Let a capital letter be small */
```

x & *mask* deletes all bits in *x* that are 0 in *mask*.

```
'a' & ~32;  /* Let a small letter be capital */
```

x ^ *mask* inverts all bits in *x* that are 1 in *mask*.

```
'a' ^ 32;   /* "Toggle" a letter */
```

Bit fields

Although it may seem efficient to use each bit of a number to store information in it, it will become nasty to access all the values by: x & 1, x & 2, x & 4, ...all the way up to $2^{sizeof\ int-1}$

For this particular reason, C offers *bit fields* like the following:

```
1 struct traffic_light {
2     int red      : 1;
3     int yellow   : 1;
4     int green    : 1;
5     int          : 5;    /* not in use */
6 };
```

The members of bit fields can be accessed as if they were members of ordinary *structs*.