A short guide to the modulo operator

Carsten Brachem <carsten.brachem@gmail.com>

November 3, 2011

1 Introduction

The purpose of this document is to clarify the meaning of the modulo operator in computer programming and to explain how it can be used – especially to create a circular buffer.

Modulo arithmetic can seem confusing at first, but once one gets the hang of it, it's quite simple actually.

2 Modulo what?

The modulo operator (%) calculates the remainder of a division.
If I write (15 % 4), the computer will divide 15 by 4 (= 3.75), round that down to the next integer (= 3) and then subtract $4 \times 3 = 12$ from 15. So we get 3 as a result.
If you want that in one formula, here it is (But if you don't want the formula, just leave it there and ignore it.):

$$(a \ \% \ b) = a - \left\lfloor \frac{a}{b} \right\rfloor \times b$$

If you're not familiar with them, those weird brackets ($\lfloor\rfloor$) mean "round that stuff down".

A few more examples:

$$(5 \ \% \ 6) = 5$$
$$(6 \ \% \ 6) = 0$$
$$(115 \ \% \ 100) = 15$$

3 Okay, I got it. Now how could that be useful?

This stuff is useful if you want numbers to stay inside a certain range.

Imagine a an analogue clock, like the one shown in Figure 1. If it's 9 o'clock now, and I ask you what time the clock will show in 4 hours, you won't say 13. You'll say 1.

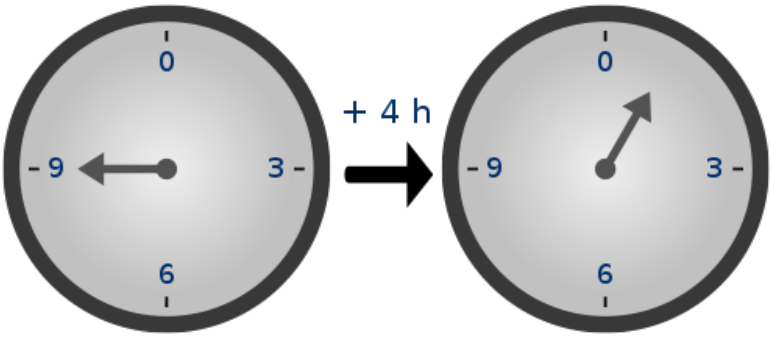

Figure 1: A clock is an example for a modulo group. (Image taken from `http://en.wikipedia.org/wiki/File:Clock_group.svg`.)

Because – and you know that without even thinking about it – times on a clock are modulo 12[1].
On a clock, we can add and subtract hours without worrying that we might get a time that isn't on it. Add 25 hours? No problem! Subtract a week? Okay.

And this property can be extremely useful when we try to implement a certain data structure – a circular buffer.

4 The circular buffer – a table without an end

So, how did we get here again? Right, we wanted a data table. And not just a regular one. We want a table that always holds the latest 20 entries (or 50, or 110000 – I'll go with 20 for this example). Okay, a table. No problem. Here we go:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Now how do we make sure that we never get outside of the table range, no matter how much we add or subtract? Modulo!
Try imagining this table not in a table form, but rather with the 0 and 19 glued together, so it forms a circle, like a clock with 20 hours on it.
After we've filled entries 0 to 19, we start at 0 again and fill 0, then 1, then 2 and so on.

But if we do it like this, we need to keep track of where in the table we are right now. We'll use an index that I will call i. i is just a number that we increment by one after each step. **But** but ensure that i will stay within our table range, afterwards, we use modulo: $i := i \ \% \ 20$.

[1] Well, technically, this is a bit different from the modulo operator used in computer programming. On the clock, we have a group from 1 to 12, whereas on a computer, we would have 0 to 11. But otherwise, it's the same.

Now we have a pointer to the newest entry in the table. That's `Table[i]`. But what about the oldest entry in the table? Well, we have 20 entries, so we go back 19 entries. But we're inside of a modulo group. That means, going back 19 steps is the same as going forward 1 step. Think again of the clock. Instead of going back 11 hours, you can go forward 1 hour and end up with the same time.
Therefore, the oldest entry in the table that we still have is at position $(i + 1)$. But we have to make sure that $(i + 1)$ is within our modulo group, so it becomes $(i + 1)$ % 20. As in `Table[(i+1)%20]`.

And that's about it.

I hope you now have a better understanding of these concepts.