Modulo operation

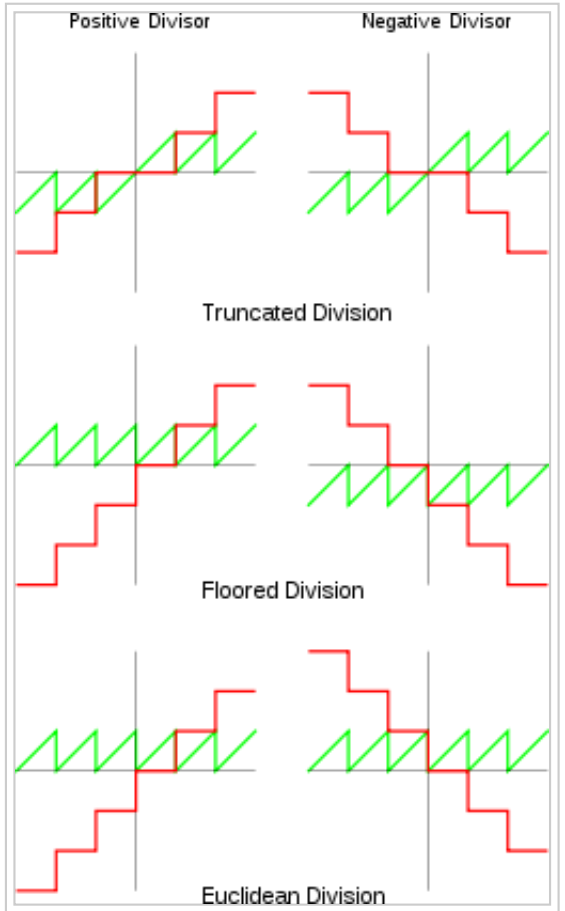
From Wikipedia, the free encyclopedia

In computing, the **modulo** operation finds the remainder after division of one number by another (sometimes called *modulus*).

Given two positive numbers, a (the dividend) and n (the divisor), a **modulo** n (abbreviated as a mod n) is the remainder of the Euclidean division of a by n. For instance, the expression "5 mod 2" would evaluate to 1 because 5 divided by 2 leaves a quotient of 2 and a remainder of 1, while "9 mod 3" would evaluate to 0 because the division of 9 by 3 has a quotient of 3 and leaves a remainder of 0; there is nothing to subtract from 9 after multiplying 3 times 3. (Note that doing the division with a calculator will not show the result referred to here by this operation; the quotient will be expressed as a decimal fraction.)

Although typically performed with a and n both being integers, many computing systems allow other types of numeric operands. The range of numbers for an integer modulo of n is 0 to $n − 1$. (n mod 1 is always 0; n mod 0 is undefined, possibly resulting in a "Division by zero" error in computer programming languages) See modular arithmetic for an older and related convention applied in number theory.

When either a or n is negative, the naive definition breaks down and programming languages differ in how these values are defined.



Quotient (red) and remainder (green) functions using different algorithms

Contents

Remainder calculation for the modulo operation

In mathematics the result of the modulo operation is the remainder of the Euclidean division. However, other conventions are possible. Computers and calculators have various ways of storing and representing numbers; thus their definition of the modulo operation depends on the programming language and/or the underlying hardware.

In nearly all computing systems, the quotient q and the remainder r of a divided by n satisfy

$$\begin{aligned} q &\in \mathbb{Z} \\ a &= nq + r \\ |r| &< |n|. \end{aligned} \qquad (1)$$

However, this still leaves a sign ambiguity if the remainder is nonzero: there are two possible choices for the remainder, one negative and the other positive, and there are also two possible choices for the quotient. Usually, in number theory, the positive remainder is always chosen, but programming languages choose depending on the language and the signs of a and/or n.[5] Standard Pascal and Algol68 give a positive remainder (or 0) even for negative divisors, and some programming languages, such as C90, leave it up to the implementation when either of n or a is negative. See the table for details. a modulo 0 is undefined in the majority of systems, although some do define it to be a.

- Many implementations use **truncated division**, where the quotient is defined by truncation $q = \operatorname{trunc}(a/n)$ and thus according to equation (**1**) the remainder would have *same sign as the dividend*. The quotient is rounded towards zero: equal to the first integer in the direction of zero from the exact rational quotient.

$$r = a - n \operatorname{trunc}\left(\frac{a}{n}\right)$$

- Donald Knuth[7] described **floored division** where the quotient is defined by the floor function $q = \lfloor a/n \rfloor$ and thus according to equation (**1**) the

remainder would have the *same sign as the divisor*. Due to the floor function, the quotient is always rounded downwards, even if it is already negative.

$$r = a - n \left\lfloor \frac{a}{n} \right\rfloor$$

- Raymond T. Boute[8] describes the Euclidean definition in which the remainder is always nonnegative, $0 \leq r$, and is therefore consistent with the **Euclidean division** algorithm. This convention is denoted *Always positive* in the table. In this case,

$$n > 0 \Rightarrow q = \left\lfloor \frac{a}{n} \right\rfloor$$
$$n < 0 \Rightarrow q = \left\lceil \frac{a}{n} \right\rceil$$

or equivalently

$$q = \operatorname{sgn}(n) \left\lfloor \frac{a}{|n|} \right\rfloor$$

where sgn is the sign function, and thus

$$r = a - |n| \left\lfloor \frac{a}{|n|} \right\rfloor.$$

- Common Lisp also defines round-division and ceiling-division where the quotient is given by $q = \text{round}(a/n)$ and q = ceil(a/n) respectively.
- IEEE 754 defines a remainder function where the quotient is a/n rounded according to the round to nearest convention. Therefore, the sign of the remainder is chosen so as to be *closest to zero*.

As described by Leijen,

Integer modulo operators in various programming languages

Language	Operator	Result has the same sign as
ActionScript	`%`	Dividend
Ada	`mod`	Divisor
	`rem`	Dividend
ASP	`Mod`	Not defined
ALGOL-68	`÷×, mod`	Always positive
AMPL	`mod`	Dividend
APL	`|`	Divisor
AppleScript	`mod`	Dividend
AWK	`%`	Dividend
BASIC	`Mod`	Not defined
bash	`%`	Dividend
bc	`%`	Dividend
C (ISO 1990)	`%`	Implementation-defined
	`div`	Dividend
C++ (ISO 1998)	`%`	Implementation-defined[1]
	`div`	Dividend
C (ISO 1999)	`%`, `div`	Dividend[2]
C++ (ISO 2011)	`%`, `div`	Dividend
C#	`%`	Dividend

> Boute argues that Euclidean division is superior to the other ones in terms of regularity and useful mathematical properties, although floored division, promoted by Knuth, is also a good definition. Despite its widespread use, truncated division is shown to be inferior to the other definitions.
>
> — Daan Leijen, *Division and Modulus for Computer Scientists*[9]

Common pitfalls

When the result of a modulo operation has the sign of the dividend, it can sometimes lead to surprising mistakes:

For example, to test whether an integer is odd, one might be inclined to test whether the remainder by 2 is equal to 1:

```
bool is_odd(int n) {
    return n % 2 == 1;
}
```

But in a language where modulo has the sign of the dividend, that is incorrect, because when *n* (the dividend) is negative and odd, *n % 2* returns −1, and the function returns false.

One correct alternative is to test that it is not 0 (because remainder 0 is the same regardless of the signs):

```
bool is_odd(int n) {
    return n % 2 != 0;
}
```

Or, by understanding in the first place that for any odd number, the modulo remainder may be either 1 or −1:

```
bool is_odd(int n) {
    return n % 2 == 1 || n % 2 == -1;
```

CLARION	`%`	Dividend
Clojure	`mod`	Divisor
COBOL[1]	`FUNCTION MOD`	Divisor
CoffeeScript	`%`	Dividend
	`%%`	Divisor[3]
ColdFusion	`%, MOD`	Dividend
Common Lisp	`mod`	Divisor
	`rem`	Dividend
D	`%`	Dividend[4]
Dart	`%`	Always Positive
	remainder()	Dividend
Eiffel	`\\`	Dividend
Erlang	`rem`	Dividend
Euphoria	`mod`	Divisor
	`remainder`	Dividend
F#	`%`	Dividend
FileMaker	`Mod`	Divisor
Forth	`mod`	implementation defined
Fortran	`mod`	Dividend
	`modulo`	Divisor
Frink	`mod`	Divisor
GML (Game Maker)	`mod`	Dividend
GDScript	`%`	Dividend
Go	`%`	Dividend
	`mod`	Divisor

```
}
```

Modulo operation expression

Some calculators have a mod() function button, and many programming languages have a mod() function or similar, expressed as mod(*a*, *n*), for example. Some also support expressions that use "%", "mod", or "Mod" as a modulo or remainder operator, such as

```
a % n
```

or

```
a mod n
```

or equivalent, for environments lacking a mod() function (note that 'int' inherently produces the floor value of a/n)

```
a - (n * int(a/n)).
```

Performance issues

Modulo operations might be implemented such that a division with a remainder is calculated each time. For special cases, there are faster alternatives on some hardware. For example, the modulo of powers of 2 can alternatively be expressed as a bitwise AND operation:

```
x % 2^n == x & (2^n - 1).
```

Examples (assuming x is a positive integer):

```
x % 2 == x & 1
x % 4 == x & 3
x % 8 == x & 7.
```

In devices and software that implement bitwise operations more efficiently than modulo, these alternative forms can result in faster calculations.[10]

Haskell		
	`rem`	Dividend
Haxe	`%`	Dividend
J	`|~`	Divisor
Java	`%`	Dividend
	`Math.floorMod`	Divisor
JavaScript	`%`	Dividend
Julia	`mod`	Divisor
	`rem`	Dividend
LibreOffice	`=MOD()`	Divisor
Lua 5	`%`	Divisor
Lua 4	`mod(x,y)`	Divisor
Liberty BASIC	`MOD`	Dividend
MathCad	`mod(x,y)`	Divisor
Maple	`e mod m`	Always positive
Mathematica	`Mod`	Divisor
MATLAB	`mod`	Divisor
	`rem`	Dividend
Maxima	`mod`	Divisor
	`remainder`	Dividend
Maya Embedded Language	`%`	Dividend
Microsoft Excel	`=MOD()`	Divisor
Minitab	`MOD`	Divisor
mksh	`%`	Dividend

Optimizing compilers may recognize expressions of the form `expression % constant` where `constant` is a power of two and automatically implement them as `expression & (constant-1)`. This can allow the programmer to write clearer code without compromising performance. (Note: This will not work for the languages whose modulo have the sign of the dividend (including C), because if the dividend is negative, the modulo will be negative; however, `expression & (constant-1)` will always produce a positive result. So special treatment has to be made when the dividend can be negative.)

Equivalencies

Some modulo operations can be factored or expanded similar to other mathematical operations. This may be useful in cryptography proofs, such as the Diffie–Hellman key exchange.

- Identity:
  - $(a \bmod n) \bmod n = a \bmod n$
  - $n^x \bmod n = 0$ for all positive integer values of x.
  - If n is a prime number which is not a divisor of b, then $ab^{n-1} \bmod n = a \bmod n$, due to Fermat's little theorem.
- Inverse:
  - $((-a \bmod n) + (a \bmod n)) \bmod n = 0$
  - $b^{-1} \bmod n$ denotes the modular multiplicative inverse, which is defined if and only if b and n are relatively prime, which is the case when the left hand side is defined: $((b^{-1} \bmod n)\,(b \bmod n)) \bmod n = 1$.
- Distributive:
  - $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
  - $ab \bmod n = ((a \bmod n)\,(b \bmod n)) \bmod n$
- Division (definition): $\frac{a}{b} \bmod n = ((a \bmod n)(b^{-1} \bmod n)) \bmod n$, when the right hand side is defined. Not defined otherwise.
- Inverse Multiplication: $((ab \bmod n)\,(b^{-1} \bmod n)) \bmod n = a \bmod n$

See also

- Modulo (disambiguation) and modulo (jargon) – many uses of the word "modulo", all of which grew out of Carl F. Gauss's introduction of *modular arithmetic* in 1801.

Modula-2	`MOD`	Divisor[2]
MUMPS	`#`	Divisor
NASM NASMX	`%`	Unsigned Modulo Operator
	`%%`	Signed Modulo Operator
Oberon	`MOD`	Divisor[3]
OCaml	`mod`	Dividend
Occam	`\`	Dividend
Pascal (Delphi)	`mod`	Dividend
Pascal (ISO-7185 and ISO-10206)	`mod`	Always positive
Perl	`%`	Divisor[4]
PHP	`%`	Dividend
PIC Basic Pro	`\\`	Dividend
PL/I	`mod`	Divisor (ANSI PL/I)
PowerShell	`%`	Dividend
Progress	`modulo`	Dividend
Prolog (ISO 1995)	`mod`	Divisor
	`rem`	Dividend
Python	`%`	Divisor
	`math.fmod`	Dividend
Racket	`remainder`	Dividend
RealBasic	`MOD`	Dividend

- Modular exponentiation

Notes

- ^ Perl usually uses arithmetic modulo operator that is machine-independent. See the Perl documentation (http://perldoc.perl.org/perlop.html#Multiplicative-Operators) for exceptions and examples.
- ^ Mathematically, these two choices are but two of the infinite number of choices available for the inequality satisfied by a remainder.
- ^ Divisor must be positive, otherwise not defined.
- ^ As implemented in ACUCOBOL, Micro Focus COBOL, and possibly others.

References

1. "ISO/IEC 14882:2003 : Programming languages -- C++". 5.6.4: ISO, IEC. 2003.. "the binary % operator yields the remainder from the division of the first expression by the second. .... If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined".
2. open-std.org (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf), section 6.5.5
3. CoffeeScript operators (http://coffeescript.org/#operators)
4. "Expressions". *D Programming Language 2.0*. Digital Mars. Retrieved 29 July 2010.
5. r6rs.org (http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-14.html#node_sec_11.7.3.1)
6. "ISO/IEC 9899:1990 : Programming languages -- C". 7.5.6.4: ISO, IEC. 1990.. "The `fmod` function returns the value `x - i * y`, for some integer `i` such that, if `y` is nonzero, the result as the same sign as `x` and magnitude less than the magnitude of `y`.".
7. Knuth, Donald. E. (1972). *The Art of Computer Programming*. Addison-Wesley.
8. Boute, Raymond T. (April 1992). "The Euclidean definition of the functions div and mod". *ACM Transactions on Programming Languages and Systems (TOPLAS)* (ACM Press (New York, NY, USA)) **14** (2): 127–144. doi:10.1145/128861.128862.
9. Leijen, Daan (December 3, 2001). "Division and Modulus for Computer Scientists" (PDF). Retrieved 2014-12-25.
10. Horvath, Adam (July 5, 2012). "Faster division and modulo operation - the power of two".

Categories: Computer arithmetic | Operators (programming) | Modular arithmetic | Binary operations

R	`%%`	Divisor
REXX	`//`	Dividend
RPG	`%REM`	Dividend
Ruby	`%, modulo()`	Divisor
	`remainder()`	Dividend
Scala	`%`	Dividend
Scheme	`modulo`	Divisor
	`remainder`	Dividend
Scheme R^{6}RS	`mod`	Always positive[5]
	`mod0`	Closest to zero[5]
Seed7	`mod`	Divisor
	`rem`	Dividend
SenseTalk	`modulo`	Divisor
	`rem`	Dividend
Smalltalk	`\\`	Divisor
	`rem:`	Dividend
SQL (SQL:1999)	`mod(x,y)`	Dividend
Standard ML	`mod`	Divisor
	`Int.rem`	Dividend
Stata	`mod(x,y)`	Always positive
Swift	`%`	Dividend
Tcl	`%`	Divisor
Torque Game	`%`	Dividend

Engine		
Turing	`mod`	Divisor
Verilog (2001)	`%`	Dividend
VHDL	`mod`	Divisor
	`rem`	Dividend
Visual Basic	`Mod`	Dividend
x86 Assembly	`IDIV`	Dividend
Xbase++	`%`	Dividend
	`Mod()`	Divisor
Z3 theorem prover	`div`, `mod`	Always positive

Floating-point modulo operators in various programming languages

Language	Operator	Result has the same sign as
C (ISO 1990)	`fmod`	Dividend[6]
C (ISO 1999)	`fmod`	Dividend
	`remainder`	Closest to zero
C++ (ISO 1998)	`std::fmod`	Dividend
C++ (ISO 2011)	`std::fmod`	Dividend
	`std::remainder`	Closest to zero
C#	`%`	Dividend
Common Lisp	`mod`	Divisor
	`rem`	Dividend
D	`%`	Dividend
Dart	`%`	Always Positive
	remainder()	Dividend
F#	`%`	Dividend
Fortran	`mod`	Dividend
	`modulo`	Divisor
Go	`math.Mod`	Dividend
Haskell (GHC)	`Data.Fixed.mod'`	Divisor
Java	`%`	Dividend

JavaScript	`%`	Dividend
Microsoft Excel	`=MOD()`	Divisor
OCaml	`mod_float`	Dividend
Perl	`POSIX::fmod`	Dividend
Perl6	`%`	Divisor
PHP	`fmod`	Dividend
Python	`%`	Divisor
	`math.fmod`	Dividend
REXX	`//`	Dividend
Ruby	`%, modulo()`	Divisor
	`remainder()`	Dividend
Scheme R^{6}RS	`flmod`	Always positive
	`flmod0`	Closest to zero
Standard ML	`Real.rem`	Dividend
Swift	`%`	Dividend
Xbase++	`%`	Dividend
	`Mod()`	Divisor