NAME

perlnumber - semantics of numbers and numeric operations in Perl

SYNOPSIS

```
$n = 1234;      # decimal integer
$n = 0b1110011;     # binary integer
$n = 01234;      # octal integer
$n = 0x1234;     # hexadecimal integer
$n = 12.34e-56;     # exponential notation
$n = "-12.34e56";     # number specified as a string
$n = "1234";     # number specified as a string
```

DESCRIPTION

This document describes how Perl internally handles numeric values.

Perl's operator overloading facility is completely ignored here. Operator overloading allows user-defined behaviors for numbers, such as operations over arbitrarily large integers, floating points numbers with arbitrary precision, operations over "exotic" numbers such as modular arithmetic or p-adic arithmetic, and so on. See *overload* for details.

Storing numbers

Perl can internally represent numbers in 3 different ways: as native integers, as native floating point numbers, and as decimal strings. Decimal strings may have an exponential notation part, as in `"12.34e-56"`. *Native* here means "a format supported by the C compiler which was used to build perl".

The term "native" does not mean quite as much when we talk about native integers, as it does when native floating point numbers are involved. The only implication of the term "native" on integers is that the limits for the maximal and the minimal supported true integral quantities are close to powers of 2. However, "native" floats have a most fundamental restriction: they may represent only those numbers which have a relatively "short" representation when converted to a binary fraction. For example, 0.9 cannot be represented by a native float, since the binary fraction for 0.9 is infinite:

```
binary0.1110011001100...
```

with the sequence `1100` repeating again and again. In addition to this limitation, the exponent of the binary number is also restricted when it is represented as a floating point number. On typical hardware, floating point values can store numbers with up to 53 binary digits, and with binary exponents between -1024 and 1024. In decimal representation this is close to 16 decimal digits and decimal exponents in the range of -304..304. The upshot of all this is that Perl cannot store a number like 12345678901234567 as a floating point number on such architectures without loss of information.

Similarly, decimal strings can represent only those numbers which have a finite decimal expansion. Being strings, and thus of arbitrary length, there is no practical limit for the exponent or number of decimal digits for these numbers. (But realize that what we are discussing the rules for just the *storage* of these numbers. The fact that you can store such "large" numbers does not mean that the *operations* over these numbers will use all of the significant digits. See *Numeric operators and numeric conversions* for details.)

In fact numbers stored in the native integer format may be stored either in the signed native form, or in the unsigned native form. Thus the limits for Perl numbers stored as native integers would typically be -2**31..2**32-1, with appropriate modifications in the case of 64-bit integers. Again, this does not mean that Perl can do operations only over integers in this range: it is possible to store many more integers in floating point format.

Summing up, Perl numeric values can store only those numbers which have a finite decimal expansion or a "short" binary expansion.

Numeric operators and numeric conversions

As mentioned earlier, Perl can store a number in any one of three formats, but most operators typically understand only one of those formats. When a numeric value is passed as an argument to such an operator, it will be converted to the format understood by the operator.

Six such conversions are possible:

```
native integer        --> native floating point (*)
native integer        --> decimal string
native floating_point --> native integer  (*)
native floating_point --> decimal string  (*)
decimal string        --> native integer
decimal string        --> native floating point (*)
```

These conversions are governed by the following general rules:

- If the source number can be represented in the target form, that representation is used.
- If the source number is outside of the limits representable in the target form, a representation of the closest limit is used. (*Loss of information*)
- If the source number is between two numbers representable in the target form, a representation of one of these numbers is used. (*Loss of information*)
- In `native floating point --> native integer` conversions the magnitude of the result is less than or equal to the magnitude of the source. (*"Rounding to zero".*)
- If the `decimal string --> native integer` conversion cannot be done without loss of information, the result is compatible with the conversion sequence `decimal_string --> native_floating_point --> native_integer`. In particular, rounding is strongly biased to 0, though a number like `"0.99999999999999999"` has a chance of being rounded to 1.

RESTRICTION: The conversions marked with `(*)` above involve steps performed by the C compiler. In particular, bugs/features of the compiler used may lead to breakage of some of the above rules.

Flavors of Perl numeric operations

Perl operations which take a numeric argument treat that argument in one of four different ways: they may force it to one of the integer/floating/ string formats, or they may behave differently depending on the format of the operand. Forcing a numeric value to a particular format does not change the number stored in the value.

All the operators which need an argument in the integer format treat the argument as in modular arithmetic, e.g., `mod 2**32` on a 32-bit architecture. `sprintf "%u", -1` therefore provides the same result as `sprintf "%u", ~0`.

Arithmetic operators

The binary operators `+ - * / % == != > < >= <=` and the unary operators `- abs` and `--` will attempt to convert arguments to integers. If both conversions are possible without loss of precision, and the operation can be performed without loss of precision then the integer result is used. Otherwise arguments are converted to floating point format and the floating point result is used. The caching of conversions (as described above) means that the integer conversion does not throw away fractional parts on floating point numbers.

`++`

`++` behaves as the other operators above, except that if it is a string matching the format `/^[a-zA-Z]*[0-9]*\z/` the string increment described in *perlop* is used.

Arithmetic operators during `use integer`

In scopes where `use integer;` is in force, nearly all the operators listed above will force their argument(s) into integer format, and return an integer result. The exceptions, `abs`, `++` and `--`, do not change their behavior with `use integer;`

Other mathematical operators

Operators such as `**`, `sin` and `exp` force arguments to floating point format.

Bitwise operators

Arguments are forced into the integer format if not strings.

Bitwise operators during `use integer`

forces arguments to integer format. Also shift operations internally use signed integers rather than the default unsigned.

Operators which expect an integer

force the argument into the integer format. This is applicable to the third and fourth arguments of `sysread`, for example.

Operators which expect a string

force the argument into the string format. For example, this is applicable to `printf "%s", $value`.

Though forcing an argument into a particular form does not change the stored number, Perl remembers the result of such conversions. In particular, though the first such conversion may be time-consuming, repeated operations will not need to redo the conversion.

AUTHOR

Ilya Zakharevich `ilya@math.ohio-state.edu`

Editorial adjustments by Gurusamy Sarathy <gsar@ActiveState.com>

Updates for 5.8.0 by Nicholas Clark <nick@ccl4.org>

SEE ALSO

overload, *perlop*