NAME

overload - Package for overloading Perl operations

SYNOPSIS

```
    package SomeThing;

    use overload
 '+' => \&myadd,
 '-' => \&mysub;
 # etc
    ...

    package main;
    $a = SomeThing->new( 57 );
    $b = 5 + $a;
    ...
    if (overload::Overloaded $b) {...}
    ...
    $strval = overload::StrVal $b;
```

DESCRIPTION

This pragma allows overloading of Perl's operators for a class. To overload built-in functions, see *"Overriding Built-in Functions" in perlsub* instead.

Fundamentals

Declaration

Arguments of the `use overload` directive are (key, value) pairs. For the full set of legal keys, see *Overloadable Operations* below.

Operator implementations (the values) can be subroutines, references to subroutines, or anonymous subroutines - in other words, anything legal inside a `&{ ... }` call. Values specified as strings are interpreted as method names. Thus

```
    package Number;
    use overload
        "-" => "minus",
        "*=" => \&muas,
        '""' => sub { ...; };
```

declares that subtraction is to be implemented by method `minus()` in the class `Number` (or one of its base classes), and that the function `Number::muas()` is to be used for the assignment form of multiplication, `*=`. It also defines an anonymous subroutine to implement stringification: this is called whenever an object blessed into the package `Number` is used in a string context (this subroutine might, for example, return the number as a Roman numeral).

Calling Conventions and Magic Autogeneration

The following sample implementation of `minus()` (which assumes that `Number` objects are simply blessed references to scalars) illustrates the calling conventions:

```
    package Number;
    sub minus {
        my ($self, $other, $swap) = @_;
        my $result = $$self - $other;         # *
        $result = -$result if $swap;
        ref $result ? $result : bless \$result;
```

```
}
# * may recurse once - see table below
```

Three arguments are passed to all subroutines specified in the `use overload` directive (with one exception - see *nomethod*). The first of these is the operand providing the overloaded operator implementation - in this case, the object whose `minus()` method is being called.

The second argument is the other operand, or `undef` in the case of a unary operator.

The third argument is set to TRUE if (and only if) the two operands have been swapped. Perl may do this to ensure that the first argument (`$self`) is an object implementing the overloaded operation, in line with general object calling conventions. For example, if `$x` and `$y` are `Numbers`:

```
operation   |   generates a call to
============|======================
$x - $y     |   minus($x, $y, '')
$x - 7      |   minus($x, 7, '')
7 - $x      |   minus($x, 7, 1)
```

Perl may also use `minus()` to implement other operators which have not been specified in the `use overload` directive, according to the rules for *Magic Autogeneration* described later. For example, the `use overload` above declared no subroutine for any of the operators `--`, `neg` (the overload key for unary minus), or `-=`. Thus

```
operation   |   generates a call to
============|======================
-$x         |   minus($x, 0, 1)
$x--        |   minus($x, 1, undef)
$x -= 3     |   minus($x, 3, undef)
```

Note the `undefs`: where autogeneration results in the method for a standard operator which does not change either of its operands, such as `-`, being used to implement an operator which changes the operand ("mutators": here, `--` and `-=`), Perl passes undef as the third argument. This still evaluates as FALSE, consistent with the fact that the operands have not been swapped, but gives the subroutine a chance to alter its behaviour in these cases.

In all the above examples, `minus()` is required only to return the result of the subtraction: Perl takes care of the assignment to $x. In fact, such methods should *not* modify their operands, even if `undef` is passed as the third argument (see *Overloadable Operations*).

The same is not true of implementations of ++ and `--`: these are expected to modify their operand. An appropriate implementation of `--` might look like

```
use overload '--' => "decr",
    # ...
sub decr { --${$_[0]}; }
```

Mathemagic, Mutators, and Copy Constructors

The term 'mathemagic' describes the overloaded implementation of mathematical operators. Mathemagical operations raise an issue. Consider the code:

```
$a = $b;
--$a;
```

If `$a` and `$b` are scalars then after these statements

```
$a == $b - 1
```

An object, however, is a reference to blessed data, so if $a and $b are objects then the assignment `$a = $b` copies only the reference, leaving $a and $b referring to the same object data. One might therefore expect the operation `--$a` to decrement $b as well as $a. However, this would not be consistent with how we expect the mathematical operators to work.

Perl resolves this dilemma by transparently calling a copy constructor before calling a method defined to implement a mutator (`--`, `+=`, and so on.). In the above example, when Perl reaches the decrement statement, it makes a copy of the object data in $a and assigns to $a a reference to the copied data. Only then does it call `decr()`, which alters the copied data, leaving $b unchanged. Thus the object metaphor is preserved as far as possible, while mathemagical operations still work according to the arithmetic metaphor.

Note: the preceding paragraph describes what happens when Perl autogenerates the copy constructor for an object based on a scalar. For other cases, see *Copy Constructor*.

Overloadable Operations

The complete list of keys that can be specified in the `use overload` directive are given, separated by spaces, in the values of the hash `%overload::ops`:

```
with_assign   => '+ - * / % ** << >> x .',
assign    => '+= -= *= /= %= **= <<= >>= x= .=',
num_comparison   => '< <= > >= == !=',
'3way_comparison'=> '<=> cmp',
str_comparison   => 'lt le gt ge eq ne',
binary    => '& &= | |= ^ ^=',
unary    => 'neg ! ~',
mutators   => '++ --',
func    => 'atan2 cos sin exp abs log sqrt int',
conversion   => 'bool "" 0+ qr',
iterators   => '<>',
filetest         => '-X',
dereferencing   => '${} @{} %{} &{} *{}',
matching   => '~~',
special   => 'nomethod fallback ='
```

Most of the overloadable operators map one-to-one to these keys. Exceptions, including additional overloadable operations not apparent from this hash, are included in the notes which follow.

* `not`

  The operator `not` is not a valid key for `use overload`. However, if the operator `!` is overloaded then the same implementation will be used for `not` (since the two operators differ only in precedence).

* `neg`

  The key `neg` is used for unary minus to disambiguate it from binary `-`.

* `++`, `--`

  Assuming they are to behave analogously to Perl's ++ and `--`, overloaded implementations of these operators are required to mutate their operands.

  No distinction is made between prefix and postfix forms of the increment and decrement operators: these differ only in the point at which Perl calls the associated subroutine when evaluating an expression.

* *Assignments*

  ```
  +=  -=  *=  /=  %=  **=  <<=  >>=  x=  .=
  &=  |=  ^=
  ```

Simple assignment is not overloadable (the `'='` key is used for the *Copy Constructor*). Perl does have a way to make assignments to an object do whatever you want, but this involves using tie(), not overload - see *"tie" in perlfunc* and the *COOKBOOK* examples below.

The subroutine for the assignment variant of an operator is required only to return the result of the operation. It is permitted to change the value of its operand (this is safe because Perl calls the copy constructor first), but this is optional since Perl assigns the returned value to the left-hand operand anyway.

An object that overloads an assignment operator does so only in respect of assignments to that object. In other words, Perl never calls the corresponding methods with the third argument (the "swap" argument) set to TRUE. For example, the operation

```
$a *= $b
```

cannot lead to `$b`'s implementation of `*=` being called, even if `$a` is a scalar. (It can, however, generate a call to `$b`'s method for `*`).

* *Non-mutators with a mutator variant*

```
+  -  *  /  %  **  <<  >>  x  .
&  |  ^
```

As described *above*, Perl may call methods for operators like `+` and `&` in the course of implementing missing operations like `++`, `+=`, and `&=`. While these methods may detect this usage by testing the definedness of the third argument, they should in all cases avoid changing their operands. This is because Perl does not call the copy constructor before invoking these methods.

* `int`

Traditionally, the Perl function `int` rounds to 0 (see *"int" in perlfunc*), and so for floating-point-like types one should follow the same semantic.

* *String, numeric, boolean, and regexp conversions*

```
""  0+  bool
```

These conversions are invoked according to context as necessary. For example, the subroutine for `'""'` (stringify) may be used where the overloaded object is passed as an argument to `print`, and that for `'bool'` where it is tested in the condition of a flow control statement (like `while`) or the ternary `?:` operation.

Of course, in contexts like, for example, `$obj + 1`, Perl will invoke `$obj`'s implementation of `+` rather than (in this example) converting `$obj` to a number using the numify method `'0+'` (an exception to this is when no method has been provided for `'+'` and *fallback* is set to TRUE).

The subroutines for `'""'`, `'0+'`, and `'bool'` can return any arbitrary Perl value. If the corresponding operation for this value is overloaded too, the operation will be called again with this value.

As a special case if the overload returns the object itself then it will be used directly. An overloaded conversion returning the object is probably a bug, because you're likely to get something that looks like `YourPackage=HASH(0x8172b34)`.

```
qr
```

The subroutine for `'qr'` is used wherever the object is interpolated into or used as a regexp, including when it appears on the RHS of a `=~` or `!~` operator.

`qr` must return a compiled regexp, or a ref to a compiled regexp (such as `qr//` returns), and any further overloading on the return value will be ignored.

* *Iteration*

If <> is overloaded then the same implementation is used for both the *read-filehandle* syntax <$var> and *globbing* syntax <${var}>.

BUGS Even in list context, the iterator is currently called only once and with scalar context.

* *File tests*

The key '-X' is used to specify a subroutine to handle all the filetest operators (-f, -x, and so on: see *"-X" in perlfunc* for the full list); it is not possible to overload any filetest operator individually. To distinguish them, the letter following the '-' is passed as the second argument (that is, in the slot that for binary operators is used to pass the second operand).

Calling an overloaded filetest operator does not affect the stat value associated with the special filehandle _. It still refers to the result of the last stat, lstat or unoverloaded filetest.

This overload was introduced in Perl 5.12.

* *Matching*

The key "~~" allows you to override the smart matching logic used by the ~~ operator and the switch construct (given/when). See *"switch" in perlsyn* and *feature*.

Unusually, the overloaded implementation of the smart match operator does not get full control of the smart match behaviour. In particular, in the following code:

```
package Foo;
use overload '~~' => 'match';

my $obj =  Foo->new();
$obj ~~ [ 1,2,3 ];
```

the smart match does *not* invoke the method call like this:

```
$obj->match([1,2,3],0);
```

rather, the smart match distributive rule takes precedence, so $obj is smart matched against each array element in turn until a match is found, so you may see between one and three of these calls instead:

```
$obj->match(1,0);
$obj->match(2,0);
$obj->match(3,0);
```

Consult the match table in *"Smart matching in detail" in perlsyn* for details of when overloading is invoked.

* *Dereferencing*

```
${}  @{}  %{}  &{}  *{}
```

If these operators are not explicitly overloaded then they work in the normal way, yielding the underlying scalar, array, or whatever stores the object data (or the appropriate error message if the dereference operator doesn't match it). Defining a catch-all 'nomethod' (see *below*) makes no difference to this as the catch-all function will not be called to implement a missing dereference operator.

If a dereference operator is overloaded then it must return a *reference* of the appropriate type (for example, the subroutine for key '${}' should return a reference to a scalar, not a scalar), or another object which overloads the operator: that is, the subroutine only determines what is dereferenced and the actual dereferencing is left to Perl. As a special case, if the subroutine returns the object itself then it will not be called again - avoiding infinite recursion.

* *Special*

```
nomethod  fallback  =
```

See *Special Keys for `use overload`*.

Magic Autogeneration

If a method for an operation is not found then Perl tries to autogenerate a substitute implementation from the operations that have been defined.

Note: the behaviour described in this section can be disabled by setting `fallback` to FALSE (see *fallback*).

In the following tables, numbers indicate priority. For example, the table below states that, if no implementation for `'!'` has been defined then Perl will implement it using `'bool'` (that is, by inverting the value returned by the method for `'bool'`); if boolean conversion is also unimplemented then Perl will use `'0+'` or, failing that, `'""'`.

```
operator | can be autogenerated from
         |
         | 0+   ""   bool   .   x
=========|=========================
   0+    |       1     2
   ""    |  1          2
   bool  |  1    2
   int   |  1    2     3
   !     |  2    3     1
   qr    |  2    1     3
   .     |  2    1     3
   x     |  2    1     3
   .=    |  3    2     4    1
   x=    |  3    2     4        1
   <>    |  2    1     3
   -X    |  2    1     3
```

Note: The iterator (`'<>'`) and file test (`'-X'`) operators work as normal: if the operand is not a blessed glob or IO reference then it is converted to a string (using the method for `'""'`, `'0+'`, or `'bool'`) to be interpreted as a glob or filename.

```
operator | can be autogenerated from
         |
         |  <   <=>   neg   -=    -
=========|=========================
   neg   |                        1
   -=    |                        1
   --    |                   1    2
   abs   | a1    a2    b1        b2    [*]
   <     |        1
   <=    |        1
   >     |        1
   >=    |        1
   ==    |        1
   !=    |        1

* one from [a1, a2] and one from [b1, b2]
```

Just as numeric comparisons can be autogenerated from the method for `'<=>'`, string comparisons can be autogenerated from that for `'cmp'`:

```
 operators          |  can be autogenerated from
====================|============================
 lt gt le ge eq ne  |  cmp
```

Similarly, autogeneration for keys `'+='` and `'++'` is analogous to `'-='` and `'--'` above:

```
operator | can be autogenerated from
         |
         |  +=    +
=========|==========================
    +=   |        1
    ++   |   1    2
```

And other assignment variations are analogous to `'+='` and `'-='` (and similar to `'.='` and `'x='` above):

```
           operator ||  *= /= %= **= <<= >>= &= ^= |=
--------------------||-------------------------------
autogenerated from ||  *  /  %  **  <<  >>  &  ^  |
```

Note also that the copy constructor (key `'='`) may be autogenerated, but only for objects based on scalars. See *Copy Constructor*.

Minimal Set of Overloaded Operations

Since some operations can be automatically generated from others, there is a minimal set of operations that need to be overloaded in order to have the complete set of overloaded operations at one's disposal. Of course, the autogenerated operations may not do exactly what the user expects. The minimal set is:

```
+ - * / % ** << >> x
<=> cmp
& | ^ ~
atan2 cos sin exp log sqrt int
"" 0+ bool
~~
```

Of the conversions, only one of string, boolean or numeric is needed because each can be generated from either of the other two.

Special Keys for use overload

nomethod

The `'nomethod'` key is used to specify a catch-all function to be called for any operator that is not individually overloaded. The specified function will be passed four parameters. The first three arguments coincide with those that would have been passed to the corresponding method if it had been defined. The fourth argument is the `use overload` key for that missing method.

For example, if `$a` is an object blessed into a package declaring

```
use overload 'nomethod' => 'catch_all', # ...
```

then the operation

```
3 + $a
```

could (unless a method is specifically declared for the key `'+'`) result in a call

```
catch_all($a, 3, 1, '+')
```

See *How Perl Chooses an Operator Implementation*.

fallback

The value assigned to the key `'fallback'` tells Perl how hard it should try to find an alternative way to implement a missing operator.

* defined, but FALSE

```
use overload "fallback" => 0, # ... ;
```

This disables *Magic Autogeneration*.

* `undef`

In the default case where no value is explicitly assigned to `fallback`, magic autogeneration is enabled.

* TRUE

The same as for `undef`, but if a missing operator cannot be autogenerated then, instead of issuing an error message, Perl is allowed to revert to what it would have done for that operator if there had been no `use overload` directive.

Note: in most cases, particularly the *Copy Constructor*, this is unlikely to be appropriate behaviour.

See *How Perl Chooses an Operator Implementation*.

Copy Constructor

As mentioned *above*, this operation is called when a mutator is applied to a reference that shares its object with some other reference. For example, if `$b` is mathemagical, and `'++'` is overloaded with `'incr'`, and `'='` is overloaded with `'clone'`, then the code

```
$a = $b;
# ... (other code which does not modify $a or $b) ...
++$b;
```

would be executed in a manner equivalent to

```
$a = $b;
# ...
$b = $b->clone(undef, "");
$b->incr(undef, "");
```

Note:

- The subroutine for `'='` does not overload the Perl assignment operator: it is used only to allow mutators to work as described here. (See *Assignments* above.)
- As for other operations, the subroutine implementing '=' is passed three arguments, though the last two are always `undef` and `''`.
- The copy constructor is called only before a call to a function declared to implement a mutator, for example, if `++$b;` in the code above is effected via a method declared for key `'++'` (or 'nomethod', passed `'++'` as the fourth argument) or, by autogeneration, `'+='`. It is not called if the increment operation is effected by a call to the method for `'+'` since, in the equivalent code,

```
$a = $b;
```

```
$b = $b + 1;
```

the data referred to by $a is unchanged by the assignment to $b of a reference to new object data.

- The copy constructor is not called if Perl determines that it is unnecessary because there is no other reference to the data being modified.
- If `'fallback'` is undefined or TRUE then a copy constructor can be autogenerated, but only for objects based on scalars. In other cases it needs to be defined explicitly. Where an object's data is stored as, for example, an array of scalars, the following might be appropriate:

```
use overload '=' => sub { bless [ @{$_[0]} ] },  # ...
```

- If `'fallback'` is TRUE and no copy constructor is defined then, for objects not based on scalars, Perl may silently fall back on simple assignment - that is, assignment of the object reference. In effect, this disables the copy constructor mechanism since no new copy of the object data is created. This is almost certainly not what you want. (It is, however, consistent: for example, Perl's fallback for the ++ operator is to increment the reference itself.)

How Perl Chooses an Operator Implementation

Which is checked first, `nomethod` or `fallback`? If the two operands of an operator are of different types and both overload the operator, which implementation is used? The following are the precedence rules:

1. If the first operand has declared a subroutine to overload the operator then use that implementation.
2. Otherwise, if fallback is TRUE or undefined for the first operand then see if the *rules for autogeneration* allows another of its operators to be used instead.
3. Unless the operator is an assignment (+=, -=, etc.), repeat step (1) in respect of the second operand.
4. Repeat Step (2) in respect of the second operand.
5. If the first operand has a "nomethod" method then use that.
6. If the second operand has a "nomethod" method then use that.
7. If `fallback` is TRUE for both operands then perform the usual operation for the operator, treating the operands as numbers, strings, or booleans as appropriate for the operator (see note).
8. Nothing worked - die.

Where there is only one operand (or only one operand with overloading) the checks in respect of the other operand above are skipped.

There are exceptions to the above rules for dereference operations (which, if Step 1 fails, always fall back to the normal, built-in implementations - see Dereferencing), and for ~~ (which has its own set of rules - see *Matching*).

Note on Step 7: some operators have a different semantic depending on the type of their operands. As there is no way to instruct Perl to treat the operands as, e.g., numbers instead of strings, the result here may not be what you expect. See *BUGS AND PITFALLS*.

Losing Overloading

The restriction for the comparison operation is that even if, for example, `cmp' should return a blessed reference, the autogenerated `lt' function will produce only a standard logical value based on the numerical value of the result of `cmp'. In particular, a working numeric conversion is needed in this

case (possibly expressed in terms of other conversions).

Similarly, `.=` and x= operators lose their mathemagical properties if the string conversion substitution is applied.

When you chop() a mathemagical object it is promoted to a string and its mathemagical properties are lost. The same can happen with other operations as well.

Inheritance and Overloading

Overloading respects inheritance via the @ISA hierarchy. Inheritance interacts with overloading in two ways.

Method names in the `use overload` directive

If `value` in

```
use overload key => value;
```

is a string, it is interpreted as a method name - which may (in the usual way) be inherited from another class.

Overloading of an operation is inherited by derived classes

Any class derived from an overloaded class is also overloaded and inherits its operator implementations. If the same operator is overloaded in more than one ancestor then the implementation is determined by the usual inheritance rules.

For example, if `A` inherits from `B` and `C` (in that order), `B` overloads + with `\&D::plus_sub`, and `C` overloads + by `"plus_meth"`, then the subroutine `D::plus_sub` will be called to implement operation + for an object in package `A`.

Note that since the value of the `fallback` key is not a subroutine, its inheritance is not governed by the above rules. In the current implementation, the value of `fallback` in the first overloaded ancestor is used, but this is accidental and subject to change.

Run-time Overloading

Since all `use` directives are executed at compile-time, the only way to change overloading during run-time is to

```
eval 'use overload "+" => \&addmethod';
```

You can also use

```
eval 'no overload "+", "--", "<="';
```

though the use of these constructs during run-time is questionable.

Public Functions

Package `overload.pm` provides the following public functions:

overload::StrVal(arg)

Gives string value of `arg` as in absence of stringify overloading. If you are using this to get the address of a reference (useful for checking if two references point to the same thing) then you may be better off using `Scalar::Util::refaddr()`, which is faster.

overload::Overloaded(arg)

Returns true if `arg` is subject to overloading of some operations.

overload::Method(obj,op)

Returns `undef` or a reference to the method that implements `op`.

Overloading Constants

For some applications, the Perl parser mangles constants too much. It is possible to hook into this process via `overload::constant()` and `overload::remove_constant()` functions.

These functions take a hash as an argument. The recognized keys of this hash are:

integer

to overload integer constants,

float

to overload floating point constants,

binary

to overload octal and hexadecimal constants,

q

to overload q-quoted strings, constant pieces of `qq`- and `qx`-quoted strings and here-documents,

qr

to overload constant pieces of regular expressions.

The corresponding values are references to functions which take three arguments: the first one is the *initial* string form of the constant, the second one is how Perl interprets this constant, the third one is how the constant is used. Note that the initial string form does not contain string delimiters, and has backslashes in backslash-delimiter combinations stripped (thus the value of delimiter is not relevant for processing of this string). The return value of this function is how this constant is going to be interpreted by Perl. The third argument is undefined unless for overloaded `q`- and `qr`- constants, it is `q` in single-quote context (comes from strings, regular expressions, and single-quote HERE documents), it is `tr` for arguments of `tr`/y operators, it is `s` for right-hand side of `s`-operator, and it is `qq` otherwise.

Since an expression `"ab$cd,,"` is just a shortcut for `'ab' . $cd . ',,'`, it is expected that overloaded constant strings are equipped with reasonable overloaded catenation operator, otherwise absurd results will result. Similarly, negative numbers are considered as negations of positive constants.

Note that it is probably meaningless to call the functions overload::constant() and overload::remove_constant() from anywhere but import() and unimport() methods. From these methods they may be called as

```
sub import {
  shift;
  return unless @_;
  die "unknown import: @_" unless @_ == 1 and $_[0] eq ':constant';
  overload::constant integer => sub {Math::BigInt->new(shift)};
}
```

IMPLEMENTATION

What follows is subject to change RSN.

The table of methods for all operations is cached in magic for the symbol table hash for the package. The cache is invalidated during processing of `use overload`, `no overload`, new function definitions, and changes in @ISA. However, this invalidation remains unprocessed until the next `bless`ing into the package. Hence if you want to change overloading structure dynamically, you'll need an additional (fake) `bless`ing to update the table.

(Every SVish thing has a magic queue, and magic is an entry in that queue. This is how a single variable may participate in multiple forms of magic simultaneously. For instance, environment variables regularly have two forms at once: their %ENV magic and their taint magic. However, the magic which implements overloading is applied to the stashes, which are rarely used directly, thus should not slow down Perl.)

If an object belongs to a package using overload, it carries a special flag. Thus the only speed penalty during arithmetic operations without overloading is the checking of this flag.

In fact, if `use overload` is not present, there is almost no overhead for overloadable operations, so most programs should not suffer measurable performance penalties. A considerable effort was made to minimize the overhead when overload is used in some package, but the arguments in question do not belong to packages using overload. When in doubt, test your speed with `use overload` and without it. So far there have been no reports of substantial speed degradation if Perl is compiled with optimization turned on.

There is no size penalty for data if overload is not used. The only size penalty if overload is used in some package is that *all* the packages acquire a magic during the next `bless`ing into the package. This magic is three-words-long for packages without overloading, and carries the cache table if the package is overloaded.

It is expected that arguments to methods that are not explicitly supposed to be changed are constant (but this is not enforced).

COOKBOOK

Please add examples to what follows!

Two-face Scalars

Put this in *two_face.pm* in your Perl library directory:

```
package two_face;  # Scalars with separate string and
                               # numeric values.
sub new { my $p = shift; bless [@_], $p }
use overload '""' => \&str, '0+' => \&num, fallback => 1;
sub num {shift->[1]}
sub str {shift->[0]}
```

Use it as follows:

```
require two_face;
my $seven = two_face->new("vii", 7);
printf "seven=$seven, seven=%d, eight=%d\n", $seven, $seven+1;
print "seven contains `i'\n" if $seven =~ /i/;
```

(The second line creates a scalar which has both a string value, and a numeric value.) This prints:

```
seven=vii, seven=7, eight=8
seven contains `i'
```

Two-face References

Suppose you want to create an object which is accessible as both an array reference and a hash reference.

```
package two_refs;
use overload '%{}' => \&gethash, '@{}' => sub { $ {shift()} };
sub new {
  my $p = shift;
```

```
  bless \ [@_], $p;
}
sub gethash {
  my %h;
  my $self = shift;
  tie %h, ref $self, $self;
  \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
  my $self = ${shift()};
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $$self->[$key] = shift;
}
sub FETCH {
  my $self = ${shift()};
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $$self->[$key];
}
```

Now one can access an object using both the array and hash syntax:

```
my $bar = two_refs->new(3,4,5,6);
$bar->[2] = 11;
$bar->{two} == 11 or die 'bad hash fetch';
```

Note several important features of this example. First of all, the *actual* type of $bar is a scalar reference, and we do not overload the scalar dereference. Thus we can get the *actual* non-overloaded contents of $bar by just using `$$bar` (what we do in functions which overload dereference). Similarly, the object returned by the TIEHASH() method is a scalar reference.

Second, we create a new tied hash each time the hash syntax is used. This allows us not to worry about a possibility of a reference loop, which would lead to a memory leak.

Both these problems can be cured. Say, if we want to overload hash dereference on a reference to an object which is *implemented* as a hash itself, the only problem one has to circumvent is how to access this *actual* hash (as opposed to the *virtual* hash exhibited by the overloaded dereference operator). Here is one possible fetching routine:

```
sub access_hash {
  my ($self, $key) = (shift, shift);
  my $class = ref $self;
  bless $self, 'overload::dummy'; # Disable overloading of %{}
  my $out = $self->{$key};
  bless $self, $class; # Restore overloading
  $out;
}
```

To remove creation of the tied hash on each access, one may an extra level of indirection which allows a non-circular structure of references:

```
package two_refs1;
use overload '%{}' => sub { ${shift()}->[1] },
             '@{}' => sub { ${shift()}->[0] };
sub new {
  my $p = shift;
  my $a = [@_];
  my %h;
  tie %h, $p, $a;
  bless \ [$a, \%h], $p;
}
sub gethash {
  my %h;
  my $self = shift;
  tie %h, ref $self, $self;
  \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
  my $a = ${shift()};
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $a->[$key] = shift;
}
sub FETCH {
  my $a = ${shift()};
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $a->[$key];
}
```

Now if $baz is overloaded like this, then `$baz` is a reference to a reference to the intermediate array, which keeps a reference to an actual array, and the access hash. The tie()ing object for the access hash is a reference to a reference to the actual array, so

- There are no loops of references.
- Both "objects" which are blessed into the class `two_refs1` are references to a reference to an array, thus references to a *scalar*. Thus the accessor expression `$$foo->[$ind]` involves no overloaded operations.

Symbolic Calculator

Put this in *symbolic.pm* in your Perl library directory:

```
package symbolic;  # Primitive symbolic calculator
use overload nomethod => \&wrap;

sub new { shift; bless ['n', @_] }
sub wrap {
  my ($obj, $other, $inv, $meth) = @_;
  ($obj, $other) = ($other, $obj) if $inv;
  bless [$meth, $obj, $other];
}
```

This module is very unusual as overloaded modules go: it does not provide any usual overloaded operators, instead it provides an implementation for *nomethod*. In this example the `nomethod` subroutine returns an object which encapsulates operations done over the objects: `symbolic->new(3)` contains `['n', 3]`, `2 + symbolic->new(3)` contains `['+', 2, ['n', 3]]`.

Here is an example of the script which "calculates" the side of circumscribed octagon using the above package:

```
require symbolic;
my $iter = 1;   # 2**($iter+2) = 8
my $side = symbolic->new(1);
my $cnt = $iter;

while ($cnt--) {
  $side = (sqrt(1 + $side**2) - 1)/$side;
}
print "OK\n";
```

The value of $side is

```
['/', ['-', ['sqrt', ['+', 1, ['**', ['n', 1], 2]],
               undef], 1], ['n', 1]]
```

Note that while we obtained this value using a nice little script, there is no simple way to *use* this value. In fact this value may be inspected in debugger (see *perldebug*), but only if `bareStringify` **O** ption is set, and not via p command.

If one attempts to print this value, then the overloaded operator `""` will be called, which will call `nomethod` operator. The result of this operator will be stringified again, but this result is again of type `symbolic`, which will lead to an infinite loop.

Add a pretty-printer method to the module *symbolic.pm*:

```
sub pretty {
  my ($meth, $a, $b) = @{+shift};
  $a = 'u' unless defined $a;
  $b = 'u' unless defined $b;
  $a = $a->pretty if ref $a;
  $b = $b->pretty if ref $b;
  "[$meth $a $b]";
}
```

Now one can finish the script by

```
print "side = ", $side->pretty, "\n";
```

The method `pretty` is doing object-to-string conversion, so it is natural to overload the operator `""` using this method. However, inside such a method it is not necessary to pretty-print the *components* $a and $b of an object. In the above subroutine `"[$meth $a $b]"` is a catenation of some strings and components $a and $b. If these components use overloading, the catenation operator will look for an overloaded operator `.`; if not present, it will look for an overloaded operator `""`. Thus it is enough to use

```
use overload nomethod => \&wrap, '""' => \&str;
sub str {
  my ($meth, $a, $b) = @{+shift};
```

```
    $a = 'u' unless defined $a;
    $b = 'u' unless defined $b;
    "[$meth $a $b]";
  }
```

Now one can change the last line of the script to

```
  print "side = $side\n";
```

which outputs

```
  side = [/ [- [sqrt [+ 1 [** [n 1 u] 2]] u] 1] [n 1 u]]
```

and one can inspect the value in debugger using all the possible methods.

Something is still amiss: consider the loop variable $cnt of the script. It was a number, not an object. We cannot make this value of type `symbolic`, since then the loop will not terminate.

Indeed, to terminate the cycle, the $cnt should become false. However, the operator `bool` for checking falsity is overloaded (this time via overloaded `""`), and returns a long string, thus any object of type `symbolic` is true. To overcome this, we need a way to compare an object to 0. In fact, it is easier to write a numeric conversion routine.

Here is the text of *symbolic.pm* with such a routine added (and slightly modified str()):

```
  package symbolic;  # Primitive symbolic calculator
  use overload
    nomethod => \&wrap, '""' => \&str, '0+' => \&num;

  sub new { shift; bless ['n', @_] }
  sub wrap {
    my ($obj, $other, $inv, $meth) = @_;
    ($obj, $other) = ($other, $obj) if $inv;
    bless [$meth, $obj, $other];
  }
  sub str {
    my ($meth, $a, $b) = @{+shift};
    $a = 'u' unless defined $a;
    if (defined $b) {
      "[$meth $a $b]";
    } else {
      "[$meth $a]";
    }
  }
  my %subr = ( n => sub {$_[0]},
        sqrt => sub {sqrt $_[0]},
        '-' => sub {shift() - shift()},
        '+' => sub {shift() + shift()},
        '/' => sub {shift() / shift()},
        '*' => sub {shift() * shift()},
        '**' => sub {shift() ** shift()},
      );
  sub num {
    my ($meth, $a, $b) = @{+shift};
    my $subr = $subr{$meth}
      or die "Do not know how to ($meth) in symbolic";
    $a = $a->num if ref $a eq __PACKAGE__;
```

```
    $b = $b->num if ref $b eq __PACKAGE__;
    $subr->($a,$b);
  }
```

All the work of numeric conversion is done in %subr and num(). Of course, %subr is not complete, it contains only operators used in the example below. Here is the extra-credit question: why do we need an explicit recursion in num()? (Answer is at the end of this section.)

Use this module like this:

```
  require symbolic;
  my $iter = symbolic->new(2); # 16-gon
  my $side = symbolic->new(1);
  my $cnt = $iter;

  while ($cnt) {
    $cnt = $cnt - 1;  # Mutator `--' not implemented
    $side = (sqrt(1 + $side**2) - 1)/$side;
  }
  printf "%s=%f\n", $side, $side;
  printf "pi=%f\n", $side*(2**($iter+2));
```

It prints (without so many line breaks)

```
  [/ [- [sqrt [+ 1 [** [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1]
     [n 1]] 2]]] 1]
     [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1] [n 1]]]=0.198912
  pi=3.182598
```

The above module is very primitive. It does not implement mutator methods (++, -= and so on), does not do deep copying (not required without mutators!), and implements only those arithmetic operations which are used in the example.

To implement most arithmetic operations is easy; one should just use the tables of operations, and change the code which fills %subr to

```
  my %subr = ( 'n' => sub {$_[0]} );
  foreach my $op (split " ", $overload::ops{with_assign}) {
    $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
  }
  my @bins = qw(binary 3way_comparison num_comparison str_comparison);
  foreach my $op (split " ", "@overload::ops{ @bins }") {
    $subr{$op} = eval "sub {shift() $op shift()}";
  }
  foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
    print "defining `$op'\n";
    $subr{$op} = eval "sub {$op shift()}";
  }
```

Since subroutines implementing assignment operators are not required to modify their operands (see *Overloadable Operations* above), we do not need anything special to make += and friends work, besides adding these operators to %subr and defining a copy constructor (needed since Perl has no way to know that the implementation of `'+='` does not mutate the argument - see *Copy Constructor*).

To implement a copy constructor, add `'=' => \&cpy` to `use overload` line, and code (this code assumes that mutators change things one level deep only, so recursive copying is not needed):

```
sub cpy {
  my $self = shift;
  bless [@$self], ref $self;
}
```

To make ++ and -- work, we need to implement actual mutators, either directly, or in `nomethod`. We continue to do things inside `nomethod`, thus add

```
if ($meth eq '++' or $meth eq '--') {
  @$obj = ($meth, (bless [@$obj]), 1); # Avoid circular reference
  return $obj;
}
```

after the first line of wrap(). This is not a most effective implementation, one may consider

```
sub inc { $_[0] = bless ['++', shift, 1]; }
```

instead.

As a final remark, note that one can fill %subr by

```
my %subr = ( 'n' => sub {$_[0]} );
foreach my $op (split " ", $overload::ops{with_assign}) {
  $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
  $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
  $subr{$op} = eval "sub {$op shift()}";
}
$subr{'++'} = $subr{'+'};
$subr{'--'} = $subr{'-'};
```

This finishes implementation of a primitive symbolic calculator in 50 lines of Perl code. Since the numeric values of subexpressions are not cached, the calculator is very slow.

Here is the answer for the exercise: In the case of str(), we need no explicit recursion since the overloaded `.`-operator will fall back to an existing overloaded operator `""`. Overloaded arithmetic operators *do not* fall back to numeric conversion if `fallback` is not explicitly requested. Thus without an explicit recursion num() would convert `['+', $a, $b]` to `$a + $b`, which would just rebuild the argument of num().

If you wonder why defaults for conversion are different for str() and num(), note how easy it was to write the symbolic calculator. This simplicity is due to an appropriate choice of defaults. One extra note: due to the explicit recursion num() is more fragile than sym(): we need to explicitly check for the type of $a and $b. If components $a and $b happen to be of some related type, this may lead to problems.

Really Symbolic Calculator

One may wonder why we call the above calculator symbolic. The reason is that the actual calculation of the value of expression is postponed until the value is *used*.

To see it in action, add a method

```
sub STORE {
```

```
    my $obj = shift;
    $#$obj = 1;
    @$obj->[0,1] = ('=', shift);
  }
```

to the package `symbolic`. After this change one can do

```
  my $a = symbolic->new(3);
  my $b = symbolic->new(4);
  my $c = sqrt($a**2 + $b**2);
```

and the numeric value of $c becomes 5. However, after calling

```
  $a->STORE(12);  $b->STORE(5);
```

the numeric value of $c becomes 13. There is no doubt now that the module symbolic provides a *symbolic* calculator indeed.

To hide the rough edges under the hood, provide a tie()d interface to the package `symbolic`. Add methods

```
  sub TIESCALAR { my $pack = shift; $pack->new(@_) }
  sub FETCH { shift }
  sub nop {  }  # Around a bug
```

(the bug, fixed in Perl 5.14, is described in *BUGS*). One can use this new interface as

```
  tie $a, 'symbolic', 3;
  tie $b, 'symbolic', 4;
  $a->nop;  $b->nop; # Around a bug

  my $c = sqrt($a**2 + $b**2);
```

Now numeric value of $c is 5. After `$a = 12; $b = 5` the numeric value of $c becomes 13. To insulate the user of the module add a method

```
  sub vars { my $p = shift; tie($_, $p), $_->nop foreach @_; }
```

Now

```
  my ($a, $b);
  symbolic->vars($a, $b);
  my $c = sqrt($a**2 + $b**2);

  $a = 3; $b = 4;
  printf "c5  %s=%f\n", $c, $c;

  $a = 12; $b = 5;
  printf "c13  %s=%f\n", $c, $c;
```

shows that the numeric value of $c follows changes to the values of $a and $b.

AUTHOR

Ilya Zakharevich <*ilya@math.mps.ohio-state.edu*>.

SEE ALSO

The `overloading` pragma can be used to enable or disable overloaded operations within a lexical scope - see *overloading*.

DIAGNOSTICS

When Perl is run with the **-Do** switch or its equivalent, overloading induces diagnostic messages.

Using the `m` command of Perl debugger (see *perldebug*) one can deduce which operations are overloaded (and which ancestor triggers this overloading). Say, if `eq` is overloaded, then the method `(eq` is shown by debugger. The method `()` corresponds to the `fallback` key (in fact a presence of this method shows that this package has overloading enabled, and it is what is used by the `Overloaded` function of module `overload`).

The module might issue the following warnings:

Odd number of arguments for overload::constant

(W) The call to overload::constant contained an odd number of arguments. The arguments should come in pairs.

`%s' is not an overloadable type

(W) You tried to overload a constant type the overload package is unaware of.

`%s' is not a code reference

(W) The second (fourth, sixth, ...) argument of overload::constant needs to be a code reference. Either an anonymous subroutine, or a reference to a subroutine.

BUGS AND PITFALLS

- No warning is issued for invalid `use overload` keys. Such errors are not always obvious:

```
use overload "+0" => sub { ...; },   # should be "0+"
    "not" => sub { ...; };           # should be "!"
```

  (Bug #74098)

- A pitfall when fallback is TRUE and Perl resorts to a built-in implementation of an operator is that some operators have more than one semantic, for example `|`:

```
use overload '0+' => sub { $_[0]->{n}; },
    fallback => 1;
my $x = bless { n => 4 }, "main";
my $y = bless { n => 8 }, "main";
print $x | $y, "\n";
```

  You might expect this to output "12". In fact, it prints "<": the ASCII result of treating "|" as a bitwise string operator - that is, the result of treating the operands as the strings "4" and "8" rather than numbers. The fact that numify (`0+`) is implemented but stringify (`""`) isn't makes no difference since the latter is simply autogenerated from the former.

  The only way to change this is to provide your own subroutine for `'|'`.

- Magic autogeneration increases the potential for inadvertently creating self-referential structures. Currently Perl will not free self-referential structures until cycles are explicitly broken. For example,

```
use overload '+' => 'add';
sub add { bless [ \$_[0], \$_[1] ] };
```

  is asking for trouble, since

```
$obj += $y;
```

will effectively become

```
$obj = add($obj, $y, undef);
```

with the same result as

```
$obj = [\$obj, \$foo];
```

Even if no *explicit* assignment-variants of operators are present in the script, they may be generated by the optimizer. For example,

```
"obj = $obj\n"
```

may be optimized to

```
my $tmp = 'obj = ' . $obj;  $tmp .= "\n";
```

- Because it is used for overloading, the per-package hash `%OVERLOAD` now has a special meaning in Perl. The symbol table is filled with names looking like line-noise.
- For the purpose of inheritance every overloaded package behaves as if `fallback` is present (possibly undefined). This may create interesting effects if some package is not overloaded, but inherits from two overloaded packages.
- Before Perl 5.14, the relation between overloading and tie()ing was broken. Overloading is triggered or not basing on the *previous* class of the tie()d variable.

  This happened because the presence of overloading was checked too early, before any tie()d access was attempted. If the class of the value FETCH()ed from the tied variable does not change, a simple workaround for code that is to run on older Perl versions is to access the value (via `( ) = $foo` or some such) immediately after tie()ing, so that after this call the *previous* class coincides with the current one.
- Barewords are not covered by overloaded string constants.