



**Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!**

***Rocky Enterprise Linux 9.2 Manual Pages on command 'Filter::Util::Call.3perl'***

***\$ man Filter::Util::Call.3perl***

Filter::Util::Call(3perl) Perl Programmers Reference Guide Filter::Util::Call(3perl)

NAME

Filter::Util::Call - Perl Source Filter Utility Module

SYNOPSIS

```
use Filter::Util::Call ;
```

DESCRIPTION

This module provides you with the framework to write Source Filters in Perl.

An alternate interface to Filter::Util::Call is now available. See Filter::Simple for more details.

A Perl Source Filter is implemented as a Perl module. The structure of the module can take one of two broadly similar formats. To distinguish between them, the first will be referred to as method filter and the second as closure filter.

Here is a skeleton for the method filter:

```
package MyFilter ;
use Filter::Util::Call ;

sub import
{
    my($type, @arguments) = @_ ;
    filter_add([]) ;
}

sub filter
{
    my($self) = @_ ;
```

```

my($status) ;
$status = filter_read() ;
$status ;
}
1 ;

```

and this is the equivalent skeleton for the closure filter:

```

package MyFilter ;
use Filter::Util::Call ;
sub import
{
my($type, @arguments) = @_ ;
filter_add(
sub
{
my($status) ;
$status = filter_read() ;
$status ;
})
}
1 ;

```

To make use of either of the two filter modules above, place the line below in a Perl source file.

```
use MyFilter;
```

In fact, the skeleton modules shown above are fully functional Source Filters, albeit fairly useless ones. All they does is filter the source stream without modifying it at all.

As you can see both modules have a broadly similar structure. They both make use of the "Filter::Util::Call" module and both have an "import" method. The difference between them is that the method filter requires a filter method, whereas the closure filter gets the equivalent of a filter method with the anonymous sub passed to filter\_add.

To make proper use of the closure filter shown above you need to have a good understanding of the concept of a closure. See [perlref](#) for more details on the mechanics of closures.

The following functions are exported by "Filter::Util::Call":

```
filter_add()  
filter_read()  
filter_read_exact()  
filter_del()
```

import()

The "import" method is used to create an instance of the filter. It is called indirectly by Perl when it encounters the "use MyFilter" line in a source file (See "import" in perlfunc for more details on "import").

It will always have at least one parameter automatically passed by Perl - this corresponds to the name of the package. In the example above it will be "MyFilter".

Apart from the first parameter, import can accept an optional list of parameters. These can be used to pass parameters to the filter. For example:

```
use MyFilter qw(a b c) ;
```

will result in the @\_ array having the following values:

```
@_ [0] => "MyFilter"  
@_ [1] => "a"  
@_ [2] => "b"  
@_ [3] => "c"
```

Before terminating, the "import" function must explicitly install the filter by calling "filter\_add".

filter\_add()

The function, "filter\_add", actually installs the filter. It takes one parameter which should be a reference. The kind of reference used will dictate which of the two filter types will be used.

If a CODE reference is used then a closure filter will be assumed.

If a CODE reference is not used, a method filter will be assumed. In a method filter, the reference can be used to store context information. The reference will be blessed into the package by "filter\_add", unless the reference was already blessed.

See the filters at the end of this documents for examples of using context information using both method filters and closure filters.

filter() and anonymous sub

Both the "filter" method used with a method filter and the anonymous sub used with a

closure filter is where the main processing for the filter is done.

The big difference between the two types of filter is that the method filter uses the object passed to the method to store any context data, whereas the closure filter uses the lexical variables that are maintained by the closure.

Note that the single parameter passed to the method filter, `$self`, is the same reference that was passed to "filter\_add" blessed into the filter's package. See the example filters later on for details of using `$self`.

Here is a list of the common features of the anonymous sub and the "filter()" method.

`$_` Although `$_` doesn't actually appear explicitly in the sample filters above, it is implicitly used in a number of places.

Firstly, when either "filter" or the anonymous sub are called, a local copy of `$_` will automatically be created. It will always contain the empty string at this point.

Next, both "filter\_read" and "filter\_read\_exact" will append any source data that is read to the end of `$_`.

Finally, when "filter" or the anonymous sub are finished processing, they are expected to return the filtered source using `$_`.

This implicit use of `$_` greatly simplifies the filter.

#### `$status`

The status value that is returned by the user's "filter" method or anonymous sub and the "filter\_read" and "read\_exact" functions take the same set of values, namely:

- < 0 Error
- = 0 EOF
- > 0 OK

#### `filter_read` and `filter_read_exact`

These functions are used by the filter to obtain either a line or block from the next filter in the chain or the actual source file if there aren't any other filters.

The function "filter\_read" takes two forms:

- `$status = filter_read() ;`
- `$status = filter_read($size) ;`

The first form is used to request a line, the second requests a block.

In line mode, "filter\_read" will append the next source line to the end of the `$_` scalar.

In block mode, "filter\_read" will append a block of data which is `<= $size` to the end

of the `$_` scalar. It is important to emphasise that "filter\_read" will not necessarily read a block which is precisely `$size` bytes.

If you need to be able to read a block which has an exact size, you can use the function "filter\_read\_exact". It works identically to "filter\_read" in block mode, except it will try to read a block which is exactly `$size` bytes in length. The only circumstances when it will not return a block which is `$size` bytes long is on EOF or error.

It is very important to check the value of `$status` after every call to "filter\_read" or "filter\_read\_exact".

#### filter\_del

The function, "filter\_del", is used to disable the current filter. It does not affect the running of the filter. All it does is tell Perl not to call filter any more.

See "Example 4: Using filter\_del" for details.

#### real\_import

Internal function which adds the filter, based on the filter\_add argument type.

#### unimport()

May be used to disable a filter, but is rarely needed. See filter\_del.

## LIMITATIONS

See "LIMITATIONS" in `perlfiter` for an overview of the general problems filtering code in a textual line-level only.

#### \_\_DATA\_\_ is ignored

The content from the `__DATA__` block is not filtered. This is a serious limitation, e.g. for the Switch module. See <http://search.cpan.org/perldoc?Switch#LIMITATIONS> for more.

#### Max. codesize limited to 32-bit

Currently internal buffer lengths are limited to 32-bit only.

## EXAMPLES

Here are a few examples which illustrate the key concepts - as such most of them are of little practical use.

The "examples" sub-directory has copies of all these filters implemented both as method filters and as closure filters.

#### Example 1: A simple filter.

Below is a method filter which is hard-wired to replace all occurrences of the string

"Joe" to "Jim". Not particularly Useful, but it is the first example and I wanted to keep it simple.

```
package Joe2Jim ;
use Filter::Util::Call ;
sub import
{
    my($type) = @_ ;
    filter_add(bless []) ;
}
sub filter
{
    my($self) = @_ ;
    my($status) ;
    s/Joe/Jim/g
    if ($status = filter_read()) > 0 ;
    $status ;
}
1 ;
```

Here is an example of using the filter:

```
use Joe2Jim ;
print "Where is Joe?\n" ;
```

And this is what the script above will print:

```
Where is Jim?
```

Example 2: Using the context

The previous example was not particularly useful. To make it more general purpose we will make use of the context data and allow any arbitrary from and to strings to be used. This time we will use a closure filter. To reflect its enhanced role, the filter is called "Subst".

```
package Subst ;
use Filter::Util::Call ;
use Carp ;
sub import
{
```

```

croak("usage: use Subst qw(from to)")

unless @_ == 3 ;

my ($self, $from, $to) = @_ ;

filter_add(

    sub

    {

        my ($status) ;

        s/$from/$to/

        if ($status = filter_read()) > 0 ;

        $status ;

    })

}

1 ;

```

and is used like this:

```

use Subst qw(Joe Jim) ;

print "Where is Joe?\n" ;

```

### Example 3: Using the context within the filter

Here is a filter which a variation of the "Joe2Jim" filter. As well as substituting all occurrences of "Joe" to "Jim" it keeps a count of the number of substitutions made in the context object.

Once EOF is detected (\$status is zero) the filter will insert an extra line into the source stream. When this extra line is executed it will print a count of the number of substitutions actually made. Note that \$status is set to 1 in this case.

```

package Count ;

use Filter::Util::Call ;

sub filter

{

    my ($self) = @_ ;

    my ($status) ;

    if (($status = filter_read()) > 0 ) {

        s/Joe/Jim/g ;

        ++ $$self ;

    }

}

```

```

elseif ($$self >= 0) { # EOF
    $_ = "print q[Made ${$self} substitutions\n]" ;
    $status = 1 ;
    $$self = -1 ;
}
$status ;
}
sub import
{
    my ($self) = @_ ;
    my ($count) = 0 ;
    filter_add(\ $count) ;
}
1 ;

```

Here is a script which uses it:

```

use Count ;
print "Hello Joe\n" ;
print "Where is Joe\n" ;

```

Outputs:

```

Hello Jim
Where is Jim
Made 2 substitutions

```

#### Example 4: Using filter\_del

Another variation on a theme. This time we will modify the "Subst" filter to allow a starting and stopping pattern to be specified as well as the from and to patterns. If you know the vi editor, it is the equivalent of this command:

```
:/start/,/stop/s/from/to/
```

When used as a filter we want to invoke it like this:

```
use NewSubst qw(start stop from to) ;
```

Here is the module.

```

package NewSubst ;
use Filter::Util::Call ;
use Carp ;

```

```

sub import
{
    my ($self, $start, $stop, $from, $to) = @_ ;
    my ($found) = 0 ;
    croak("usage: use Subst qw(start stop from to)")
        unless @_ == 5 ;
    filter_add(
        sub
        {
            my ($status) ;
            if (($status = filter_read()) > 0) {
                $found = 1
                if $found == 0 and /$start/ ;
                if ($found) {
                    s/$from/$to/ ;
                    filter_del() if /$stop/ ;
                }
            }
            $status ;
        })
    }
    1 ;
}

```

#### Filter::Simple

If you intend using the Filter::Call functionality, I would strongly recommend that you check out Damian Conway's excellent Filter::Simple module. Damian's module provides a much cleaner interface than Filter::Util::Call. Although it doesn't allow the fine control that Filter::Util::Call does, it should be adequate for the majority of applications. It's available at

<http://search.cpan.org/dist/Filter-Simple/>

#### AUTHOR

Paul Marquess

#### DATE

26th January 1996

## LICENSE

Copyright (c) 1995-2011 Paul Marquess. All rights reserved. Copyright (c) 2011-2014 Reini Urban. All rights reserved. Copyright (c) 2014-2017 cPanel Inc. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

perl v5.34.0

2023-11-23

Filter::Util::Call(3perl)