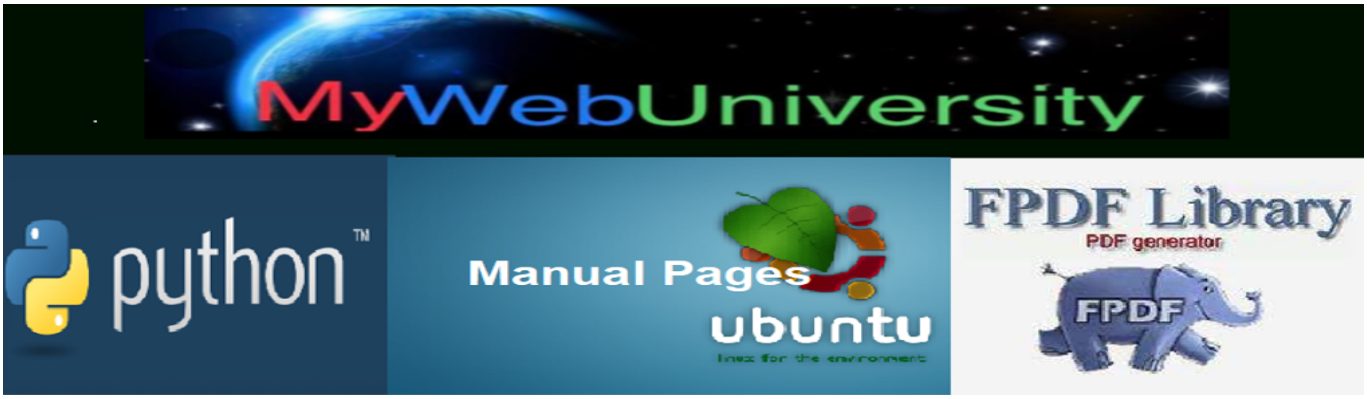

Rocky Enterprise Linux 9.2 Manual Pages on command 'Carp::Assert.3pm'

$ man Carp::Assert.3pm

Carp::Assert(3pm) User Contributed Perl Documentation Carp::Assert(3pm)

NAME

Carp::Assert - executable comments

SYNOPSIS

```
# Assertions are on.
use Carp::Assert;
$next_sunrise_time = sunrise();
# Assert that the sun must rise in the next 24 hours.
assert(($next_sunrise_time - time) < 24*60*60) if DEBUG;
# Assert that your customer's primary credit card is active
affirm {
    my @cards = @{$customer->credit_cards};
    $cards[0]->is_active;
};
# Assertions are off.
no Carp::Assert;
$next_pres = divine_next_president();
# Assert that if you predict Dan Quayle will be the next president
# your crystal ball might need some polishing.  However, since
# assertions are off, IT COULD HAPPEN!
shouldnt($next_pres, 'Dan Quayle') if DEBUG;
```

DESCRIPTION

"We are ready for any unforseen event that may or may not

occur."

- Dan Quayle

Carp::Assert is intended for a purpose like the ANSI C library assert.h <http://en.wikipedia.org/wiki/Assert.h>. If you're already familiar with assert.h, then you can probably skip this and go straight to the FUNCTIONS section.

Assertions are the explicit expressions of your assumptions about the reality your program is expected to deal with, and a declaration of those which it is not. They are used to prevent your program from blissfully processing garbage inputs (garbage in, garbage out becomes garbage in, error out) and to tell you when you've produced garbage output. (If I was going to be a cynic about Perl and the user nature, I'd say there are no user inputs but garbage, and Perl produces nothing but...)

An assertion is used to prevent the impossible from being asked of your code, or at least tell you when it does. For example:

```
# Take the square root of a number.
sub my_sqrt {
    my($num) = shift;
    # the square root of a negative number is imaginary.
    assert($num >= 0);
    return sqrt $num;
}
```

The assertion will warn you if a negative number was handed to your subroutine, a reality the routine has no intention of dealing with.

An assertion should also be used as something of a reality check, to make sure what your code just did really did happen:

```
open(FILE, $filename) || die $!;
@stuff = <FILE>;
@stuff = do_something(@stuff);
# I should have some stuff.
assert(@stuff > 0);
```

The assertion makes sure you have some @stuff at the end. Maybe the file was empty, maybe do_something() returned an empty list... either way, the assert() will give you a clue as to where the problem lies, rather than 50 lines down at when you wonder why your program isn't printing anything.

Since assertions are designed for debugging and will remove themelves from production code, your assertions should be carefully crafted so as to not have any side-effects, change any variables, or otherwise have any effect on your program. Here is an example of a bad assertation:

```
assert($error = 1 if $king ne 'Henry');  # Bad!
```

It sets an error flag which may then be used somewhere else in your program. When you shut off your assertions with the $DEBUG flag, $error will no longer be set.

Here's another example of bad use:

```
assert($next_pres ne 'Dan Quayle' or goto Canada);  # Bad!
```

This assertion has the side effect of moving to Canada should it fail. This is a very bad assertion since error handling should not be placed in an assertion, nor should it have side-effects.

In short, an assertion is an executable comment. For instance, instead of writing this

```
# $life ends with a '!'
$life = begin_life();
```

you'd replace the comment with an assertion which enforces the comment.

```
$life = begin_life();
assert( $life =~ /!$/ );
```

FUNCTIONS

assert

```
assert(EXPR) if DEBUG;
assert(EXPR, $name) if DEBUG;
```

assert's functionality is effected by compile time value of the DEBUG constant, controlled by saying "use Carp::Assert" or "no Carp::Assert". In the former case, assert will function as below. Otherwise, the assert function will compile itself out of the program. See "Debugging vs Production" for details.

Give assert an expression, assert will Carp::confess() if that expression is false, otherwise it does nothing. (DO NOT use the return value of assert for anything, I mean it... really!).

The error from assert will look something like this:

```
Assertion failed!
        Carp::Assert::assert(0) called at prog line 23
        main::foo called at prog line 50
```

Indicating that in the file "prog" an assert failed inside the function main::foo() on line 23 and that foo() was in turn called from line 50 in the same file.

If given a $name, assert() will incorporate this into your error message, giving users something of a better idea what's going on.

```
assert( Dogs->isa('People'), 'Dogs are people, too!' ) if DEBUG;
# Result - "Assertion (Dogs are people, too!) failed!"
```

affirm

```
affirm BLOCK if DEBUG;
affirm BLOCK $name if DEBUG;
```

Very similar to assert(), but instead of taking just a simple expression it takes an entire block of code and evaluates it to make sure its true. This can allow more complicated assertions than assert() can without letting the debugging code leak out into production and without having to smash together several statements into one.

```
affirm {
    my $customer = Customer->new($customerid);
    my @cards = $customer->credit_cards;
    grep { $_->is_active } @cards;
} "Our customer has an active credit card";
```

affirm() also has the nice side effect that if you forgot the "if DEBUG" suffix its arguments will not be evaluated at all. This can be nice if you stick affirm()s with expensive checks into hot loops and other time-sensitive parts of your program.

If the $name is left off and your Perl version is 5.6 or higher the affirm() diagnostics will include the code begin affirmed.

should

shouldnt

```
should  ($this, $shouldbe)   if DEBUG;
shouldnt($this, $shouldntbe) if DEBUG;
```

Similar to assert(), it is specially for simple "this should be that" or "this should be anything but that" style of assertions.

Due to Perl's lack of a good macro system, assert() can only report where something failed, but it can't report what failed or how. should() and shouldnt() can produce more informative error messages:

```
Assertion ('this' should be 'that'!) failed!
```

Carp::Assert::should('this', 'that') called at moof line 29

main::foo() called at moof line 58

So this:

should($this, $that) if DEBUG;

is similar to this:

assert($this eq $that) if DEBUG;

except for the better error message.

Currently, should() and shouldnt() can only do simple eq and ne tests (respectively).

Future versions may allow regexes.

Debugging vs Production

Because assertions are extra code and because it is sometimes necessary to place them in 'hot' portions of your code where speed is paramount, Carp::Assert provides the option to remove its assert() calls from your program.

So, we provide a way to force Perl to inline the switched off assert() routine, thereby removing almost all performance impact on your production code.

```
no Carp::Assert;  # assertions are off.
assert(1==1) if DEBUG;
```

DEBUG is a constant set to 0. Adding the 'if DEBUG' condition on your assert() call gives perl the cue to go ahead and remove assert() call from your program entirely, since the if conditional will always be false.

```
# With C<no Carp::Assert> the assert() has no impact.
for (1..100) {
    assert( do_some_really_time_consuming_check ) if DEBUG;
}
```

If "if DEBUG" gets too annoying, you can always use affirm().

```
# Once again, affirm() has (almost) no impact with C<no Carp::Assert>
for (1..100) {
    affirm { do_some_really_time_consuming_check };
}
```

Another way to switch off all asserts, system wide, is to define the NDEBUG or the PERL_NDEBUG environment variable.

You can safely leave out the "if DEBUG" part, but then your assert() function will always execute (and its arguments evaluated and time spent). To get around this, use affirm().

You still have the overhead of calling a function but at least its arguments will not be evaluated.

Differences from ANSI C

assert() is intended to act like the function from ANSI C fame. Unfortunately, due to Perl's lack of macros or strong inlining, it's not nearly as unobtrusive.

Well, the obvious one is the "if DEBUG" part. This is cleanest way I could think of to cause each assert() call and its arguments to be removed from the program at compile-time, like the ANSI C macro does.

Also, this version of assert does not report the statement which failed, just the line number and call frame via Carp::confess. You can't do "assert('$a == $b')" because $a and $b will probably be lexical, and thus unavailable to assert(). But with Perl, unlike C, you always have the source to look through, so the need isn't as great.

EFFICIENCY

With "no Carp::Assert" (or NDEBUG) and using the "if DEBUG" suffixes on all your assertions, Carp::Assert has almost no impact on your production code. I say almost because it does still add some load-time to your code (I've tried to reduce this as much as possible).

If you forget the "if DEBUG" on an "assert()", "should()" or "shouldnt()", its arguments are still evaluated and thus will impact your code. You'll also have the extra overhead of calling a subroutine (even if that subroutine does nothing).

Forgetting the "if DEBUG" on an "affirm()" is not so bad. While you still have the overhead of calling a subroutine (one that does nothing) it will not evaluate its code block and that can save a lot.

Try to remember the if DEBUG.

ENVIRONMENT

NDEBUG

Defining NDEBUG switches off all assertions. It has the same effect as changing "use Carp::Assert" to "no Carp::Assert" but it effects all code.

PERL_NDEBUG

Same as NDEBUG and will override it. Its provided to give you something which won't conflict with any C programs you might be working on at the same time.

BUGS, CAVETS and other MUSINGS

Conflicts with "POSIX.pm"

The "POSIX" module exports an "assert" routine which will conflict with "Carp::Assert" if both are used in the same namespace. If you are using both together, prevent "POSIX" from exporting like so:

```
use POSIX ();
use Carp::Assert;
```

Since "POSIX" exports way too much, you should be using it like that anyway.

"affirm" and $^S

affirm() mucks with the expression's caller and it is run in an eval so anything that checks $^S will be wrong.

"shouldn't"

Yes, there is a "shouldn't" routine. It mostly works, but you must put the "if DEBUG" after it.

missing "if DEBUG"

It would be nice if we could warn about missing "if DEBUG".

SEE ALSO

assert.h <http://en.wikipedia.org/wiki/Assert.h> - the wikipedia page about "assert.h".

Carp::Assert::More provides a set of convenience functions that are wrappers around "Carp::Assert".

Sub::Assert provides support for subroutine pre- and post-conditions. The documentation says it's slow.

PerlX::Assert provides compile-time assertions, which are usually optimised away at compile time. Currently part of the Moops distribution, but may get its own distribution sometime in 2014.

Devel::Assert also provides an "assert" function, for Perl >= 5.8.1.

assertions provides an assertion mechanism for Perl >= 5.9.0.

REPOSITORY

<https://github.com/schwern/Carp-Assert>

COPYRIGHT

AUTHOR

Michael G Schwern <schwern@pobox.com>