NAME

Readonly – Facility for creating read–only scalars, arrays, hashes

Synopsis

```
    use Readonly;

    # Deep Read-only scalar
    Readonly::Scalar    $sca => $initial_value;
    Readonly::Scalar my $sca => $initial_value;

    # Deep Read-only array
    Readonly::Array    @arr => @values;
    Readonly::Array my @arr => @values;

    # Deep Read-only hash
    Readonly::Hash    %has => (key => value, key => value, ...);
    Readonly::Hash my %has => (key => value, key => value, ...);
    # or:
    Readonly::Hash    %has => {key => value, key => value, ...};

    # You can use the read-only variables like any regular variables:
    print $sca;
    $something = $sca + $arr[2];
    next if $has{$some_key};

    # But if you try to modify a value, your program will die:
    $sca = 7;
    push @arr, 'seven';
    delete $has{key};
    # The error message is "Modification of a read-only value attempted"

    # Alternate form (Perl 5.8 and later)
    Readonly    $sca => $initial_value;
    Readonly my $sca => $initial_value;
    Readonly    @arr => @values;
    Readonly my @arr => @values;
    Readonly    %has => (key => value, key => value, ...);
    Readonly my %has => (key => value, key => value, ...);
    Readonly my $sca; # Implicit undef, readonly value

    # Alternate form (for Perls earlier than v5.8)
    Readonly    \$sca => $initial_value;
    Readonly \my $sca => $initial_value;
    Readonly    \@arr => @values;
    Readonly \my @arr => @values;
    Readonly    \%has => (key => value, key => value, ...);
    Readonly \my %has => (key => value, key => value, ...);
```

Description

This is a facility for creating non-modifiable variables. This is useful for configuration files, headers, etc. It can also be useful as a development and debugging tool for catching updates to variables that should not be changed.

Variable Depth

Readonly has the ability to create both deep and shallow readonly variables.

If you pass a `$ref`, an `@array` or a `%hash` to corresponding functions `::Scalar()`, `::Array()` and

`::Hash()`, then those functions recurse over the data structure, marking everything as readonly. The entire structure is then non-modifiable. This is normally what you want.

If you want only the top level to be readonly, use the alternate (and poorly named) `::Scalar1()`, `::Array1()`, and `::Hash1()` functions.

Plain `Readonly()` creates what the original author calls a "shallow" readonly variable, which is great if you don't plan to use it on anything but only one dimensional scalar values.

`Readonly::Scalar()` makes the variable 'deeply' readonly, so the following snippet kills over as you expect:

```
use Readonly;

Readonly::Scalar my $ref => { 1 => 'a' };
$ref->{1} = 'b';
$ref->{2} = 'b';
```

While the following snippet does **not** make your structure 'deeply' readonly:

```
use Readonly;

Readonly my $ref => { 1 => 'a' };
$ref->{1} = 'b';
$ref->{2} = 'b';
```

The Past

The following sections are updated versions of the previous authors documentation.

Comparison with "use constant"

Perl provides a facility for creating constant values, via the constant pragma. There are several problems with this pragma.

- The constants created have no leading sigils.
- These constants cannot be interpolated into strings.
- Syntax can get dicey sometimes. For example:

  ```
  use constant CARRAY => (2, 3, 5, 7, 11, 13);
  $a_prime = CARRAY[2];        # wrong!
  $a_prime = (CARRAY)[2];      # right -- MUST use parentheses
  ```

- You have to be very careful in places where barewords are allowed.

  For example:

  ```
  use constant SOME_KEY => 'key';
  %hash = (key => 'value', other_key => 'other_value');
  $some_value = $hash{SOME_KEY};        # wrong!
  $some_value = $hash{+SOME_KEY};       # right
  ```

  (who thinks to use a unary plus when using a hash to scalarize the key?)

- `use constant` works for scalars and arrays, not hashes.
- These constants are global to the package in which they're declared; cannot be lexically scoped.
- Works only at compile time.
- Can be overridden:

  ```
  use constant PI => 3.14159;
  ...
  use constant PI => 2.71828;
  ```

(this does generate a warning, however, if you have warnings enabled).

- It is very difficult to make and use deep structures (complex data structures) with `use constant`.

Comparison with typeglob constants

Another popular way to create read-only scalars is to modify the symbol table entry for the variable by using a typeglob:

```
*a = \'value';
```

This works fine, but it only works for global variables ("my" variables have no symbol table entry). Also, the following similar constructs do **not** work:

```
*a = [1, 2, 3];       # Does NOT create a read-only array
*a = { a => 'A'};     # Does NOT create a read-only hash
```

Pros

Readonly.pm, on the other hand, will work with global variables and with lexical ("my") variables. It will create scalars, arrays, or hashes, all of which look and work like normal, read-write Perl variables. You can use them in scalar context, in list context; you can take references to them, pass them to functions, anything.

Readonly.pm also works well with complex data structures, allowing you to tag the whole structure as nonmodifiable, or just the top level.

Also, Readonly variables may not be reassigned. The following code will die:

```
Readonly::Scalar $pi => 3.14159;
...
Readonly::Scalar $pi => 2.71828;
```

Cons

Readonly.pm used to impose a performance penalty. It was pretty slow. How slow? Run the `eg/benchmark.pl` script that comes with Readonly. On my test system, "use constant" (const), typeglob constants (tglob), regular read/write Perl variables (normal/literal), and the new Readonly (ro/ro_simple) are all about the same speed, the old, tie based Readonly.pm constants were about 1/22 the speed.

However, there is relief. There is a companion module available, Readonly::XS. You won't need this if you're using Perl 5.8.x or higher.

I repeat, you do not need Readonly::XS if your environment has perl 5.8.x or higher. Please see section entitled Internals for more.

Functions

Readonly::Scalar `$var` => `$value`;

Creates a nonmodifiable scalar, `$var`, and assigns a value of `$value` to it. Thereafter, its value may not be changed. Any attempt to modify the value will cause your program to die.

A value *must* be supplied. If you want the variable to have `undef` as its value, you must specify `undef`.

If `$value` is a reference to a scalar, array, or hash, then this function will mark the scalar, array, or hash it points to as being Readonly as well, and it will recursively traverse the structure, marking the whole thing as Readonly. Usually, this is what you want. However, if you want only the `$value` marked as Readonly, use `Scalar1`.

If `$var` is already a Readonly variable, the program will die with an error about reassigning Readonly variables.

Readonly::Array `@arr` => (value, value, ...);

Creates a nonmodifiable array, `@arr`, and assigns the specified list of values to it. Thereafter, none of its values may be changed; the array may not be lengthened or shortened or spliced. Any attempt to do so will cause your program to die.

If any of the values passed is a reference to a scalar, array, or hash, then this function will mark the scalar, array, or hash it points to as being Readonly as well, and it will recursively traverse the structure, marking the whole thing as Readonly. Usually, this is what you want. However, if you want only the hash `%@arr` itself marked as Readonly, use `Array1`.

If `@arr` is already a Readonly variable, the program will die with an error about reassigning Readonly variables.

Readonly::Hash `%h` => (key => value, key => value, ...);
Readonly::Hash `%h` => {key => value, key => value, ...};

Creates a nonmodifiable hash, `%h`, and assigns the specified keys and values to it. Thereafter, its keys or values may not be changed. Any attempt to do so will cause your program to die.

A list of keys and values may be specified (with parentheses in the synopsis above), or a hash reference may be specified (curly braces in the synopsis above). If a list is specified, it must have an even number of elements, or the function will die.

If any of the values is a reference to a scalar, array, or hash, then this function will mark the scalar, array, or hash it points to as being Readonly as well, and it will recursively traverse the structure, marking the whole thing as Readonly. Usually, this is what you want. However, if you want only the hash `%h` itself marked as Readonly, use `Hash1`.

If `%h` is already a Readonly variable, the program will die with an error about reassigning Readonly variables.

Readonly `$var` => `$value`;
Readonly `@arr` => (value, value, ...);
Readonly `%h` => (key => value, ...);
Readonly `%h` => {key => value, ...};
Readonly `$var`;

The `Readonly` function is an alternate to the `Scalar`, `Array`, and `Hash` functions. It has the advantage (if you consider it an advantage) of being one function. That may make your program look neater, if you're initializing a whole bunch of constants at once. You may or may not prefer this uniform style.

It has the disadvantage of having a slightly different syntax for versions of Perl prior to 5.8. For earlier versions, you must supply a backslash, because it requires a reference as the first parameter.

```
Readonly \$var => $value;
Readonly \@arr => (value, value, ...);
Readonly \%h   => (key => value, ...);
Readonly \%h   => {key => value, ...};
```

You may or may not consider this ugly.

Note that you can create implicit undefined variables with this function like so `Readonly my $var;` while a verbose undefined value must be passed to the standard `Scalar`, `Array`, and `Hash` functions.

Readonly::Scalar1 `$var` => `$value`;
Readonly::Array1 `@arr` => (value, value, ...);
Readonly::Hash1 `%h` => (key => value, key => value, ...);
Readonly::Hash1 `%h` => {key => value, key => value, ...};

These alternate functions create shallow Readonly variables, instead of deep ones. For example:

```
Readonly::Array1 @shal => (1, 2, {perl=>'Rules', java=>'Bites'}, 4, 5);
Readonly::Array  @deep => (1, 2, {perl=>'Rules', java=>'Bites'}, 4, 5);

$shal[1] = 7;            # error
$shal[2]{APL}='Weird';   # Allowed! since the hash isn't Readonly
$deep[1] = 7;            # error
```

```
        $deep[2]{APL}='Weird';  # error, since the hash is Readonly
```

Cloning

When cloning using Storable or Clone you will notice that the value stays readonly, which is correct. If you want to clone the value without copying the readonly flag, use the `Clone` function:

```
    Readonly::Scalar my $scalar => {qw[this that]};
    # $scalar->{'eh'} = 'foo'; # Modification of a read-only value attempted
    my $scalar_clone = Readonly::Clone $scalar;
    $scalar_clone->{'eh'} = 'foo';
    # $scalar_clone is now {this => 'that', eh => 'foo'};
```

The new variable (`$scalar_clone`) is a mutable clone of the original `$scalar`.

Examples

These are a few very simple examples:

Scalars

A plain old read-only value

```
    Readonly::Scalar $a => "A string value";
```

The value need not be a compile-time constant:

```
    Readonly::Scalar $a => $computed_value;
```

Arrays/Lists

A read-only array:

```
    Readonly::Array @a => (1, 2, 3, 4);
```

The parentheses are optional:

```
    Readonly::Array @a => 1, 2, 3, 4;
```

You can use Perl's built-in array quoting syntax:

```
    Readonly::Array @a => qw/1 2 3 4/;
```

You can initialize a read-only array from a variable one:

```
    Readonly::Array @a => @computed_values;
```

A read-only array can be empty, too:

```
    Readonly::Array @a => ();
    Readonly::Array @a;        # equivalent
```

Hashes

Typical usage:

```
    Readonly::Hash %a => (key1 => 'value1', key2 => 'value2');
```

A read-only hash can be initialized from a variable one:

```
    Readonly::Hash %a => %computed_values;
```

A read-only hash can be empty:

```
    Readonly::Hash %a => ();
    Readonly::Hash %a;        # equivalent
```

If you pass an odd number of values, the program will die:

```
    Readonly::Hash %a => (key1 => 'value1', "value2");
    # This dies with "May not store an odd number of values in a hash"
```

Exports

Historically, this module exports the `Readonly` symbol into the calling program's namespace by default. The following symbols are also available for import into your program, if you like: `Scalar`, `Scalar1`, `Array`, `Array1`, `Hash`, and `Hash1`.

Internals

Some people simply do not understand the relationship between this module and Readonly::XS so I'm adding this section. Odds are, they still won't understand but I like to write so...

In the past, Readonly's "magic" was performed by `tie()`–ing variables to the `Readonly::Scalar`, `Readonly::Array`, and `Readonly::Hash` packages (not to be confused with the functions of the same names) and acting on `WRITE`, `READ`, et. al. While this worked well, it was slow. Very slow. Like 20–30 times slower than accessing variables directly or using one of the other const-related modules that have cropped up since Readonly was released in 2003.

To 'fix' this, Readonly::XS was written. If installed, Readonly::XS used the internal methods `SvREADONLY` and `SvREADONLY_on` to lock simple scalars. On the surface, everything was peachy but things weren't the same behind the scenes. In edge cases, code performed very differently if Readonly::XS was installed and because it wasn't a required dependency in most code, it made downstream bugs very hard to track.

In the years since Readonly::XS was released, the then private internal methods have been exposed and can be used in pure perl. Similar modules were written to take advantage of this and a patch to Readonly was created. We no longer need to build and install another module to make Readonly useful on modern builds of perl.

- You do not need to install Readonly::XS.
- You should stop listing Readonly::XS as a dependency or expect it to be installed.
- Stop testing the `$Readonly::XSokay` variable!

Requirements

Please note that most users of Readonly no longer need to install the companion module Readonly::XS which is recommended but not required for perl 5.6.x and under. Please do not force it as a requirement in new code and do not use the package variable `$Readonly::XSokay` in code/tests. For more, see "Internals" in the section on Readonly's new internals.

There are no non-core requirements.

Bug Reports

If email is better for you, my address is mentioned below but I would rather have bugs sent through the issue tracker found at http://github.com/sanko/readonly/issues.

Acknowledgements

Thanks to Slaven Rezic for the idea of one common function (Readonly) for all three types of variables (13 April 2002).

Thanks to Ernest Lergon for the idea (and initial code) for deeply-Readonly data structures (21 May 2002).

Thanks to Damian Conway for the idea (and code) for making the Readonly function work a lot smoother under perl 5.8+.

Author

Sanko Robinson <sanko@cpan.org> – http://sankorobinson.com/

CPAN ID: SANKO

Original author: Eric J. Roode, roode@cpan.org

License and Legal