NAME

SelfLoader - load functions only on demand

SYNOPSIS

```
package FOOBAR;
use SelfLoader;

... (initializing code)

__DATA__
sub {....
```

DESCRIPTION

This module tells its users that functions in the FOOBAR package are to be autoloaded from after the __DATA__ token. See also *"Autoloading" in perlsub*.

The __DATA__ token

The __DATA__ token tells the perl compiler that the perl code for compilation is finished. Everything after the __DATA__ token is available for reading via the filehandle FOOBAR::DATA, where FOOBAR is the name of the current package when the __DATA__ token is reached. This works just the same as __END__ does in package 'main', but for other modules data after __END__ is not automatically retrievable, whereas data after __DATA__ is. The __DATA__ token is not recognized in versions of perl prior to 5.001m.

Note that it is possible to have __DATA__ tokens in the same package in multiple files, and that the last __DATA__ token in a given package that is encountered by the compiler is the one accessible by the filehandle. This also applies to __END__ and main, i.e. if the 'main' program has an __END__, but a module 'require'd (_not_ 'use'd) by that program has a 'package main;' declaration followed by an '__DATA__', then the DATA filehandle is set to access the data after the __DATA__ in the module, _not_ the data after the __END__ token in the 'main' program, since the compiler encounters the 'require'd file later.

SelfLoader autoloading

The **SelfLoader** works by the user placing the __DATA__ token *after* perl code which needs to be compiled and run at 'require' time, but *before* subroutine declarations that can be loaded in later - usually because they may never be called.

The **SelfLoader** will read from the FOOBAR::DATA filehandle to load in the data after __DATA__, and load in any subroutine when it is called. The costs are the one-time parsing of the data after __DATA__, and a load delay for the _first_ call of any autoloaded function. The benefits (hopefully) are a speeded up compilation phase, with no need to load functions which are never used.

The **SelfLoader** will stop reading from __DATA__ if it encounters the __END__ token - just as you would expect. If the __END__ token is present, and is followed by the token DATA, then the **SelfLoader** leaves the FOOBAR::DATA filehandle open on the line after that token.

The **SelfLoader** exports the AUTOLOAD subroutine to the package using the **SelfLoader**, and this loads the called subroutine when it is first called.

There is no advantage to putting subroutines which will _always_ be called after the __DATA__ token.

Autoloading and package lexicals

A 'my $pack_lexical' statement makes the variable $pack_lexical local _only_ to the file up to the __DATA__ token. Subroutines declared elsewhere _cannot_ see these types of variables, just as if you declared subroutines in the package but in another file, they cannot see these variables.

So specifically, autoloaded functions cannot see package lexicals (this applies to both the **SelfLoader** and the Autoloader). The `vars` pragma provides an alternative to defining package-level globals that will be visible to autoloaded routines. See the documentation on **vars** in the pragma section of *perlmod*.

SelfLoader and AutoLoader

The **SelfLoader** can replace the AutoLoader - just change 'use AutoLoader' to 'use SelfLoader' (though note that the **SelfLoader** exports the AUTOLOAD function - but if you have your own AUTOLOAD and are using the AutoLoader too, you probably know what you're doing), and the `__END__` token to `__DATA__`. You will need perl version 5.001m or later to use this (version 5.001 with all patches up to patch m).

There is no need to inherit from the **SelfLoader**.

The **SelfLoader** works similarly to the AutoLoader, but picks up the subs from after the `__DATA__` instead of in the 'lib/auto' directory. There is a maintenance gain in not needing to run AutoSplit on the module at installation, and a runtime gain in not needing to keep opening and closing files to load subs. There is a runtime loss in needing to parse the code after the `__DATA__`. Details of the **AutoLoader** and another view of these distinctions can be found in that module's documentation.

__DATA__, __END__, and the FOOBAR::DATA filehandle.

This section is only relevant if you want to use the `FOOBAR::DATA` together with the **SelfLoader**.

Data after the `__DATA__` token in a module is read using the FOOBAR::DATA filehandle. `__END__` can still be used to denote the end of the `__DATA__` section if followed by the token DATA - this is supported by the **SelfLoader**. The `FOOBAR::DATA` filehandle is left open if an `__END__` followed by a DATA is found, with the filehandle positioned at the start of the line after the `__END__` token. If no `__END__` token is present, or an `__END__` token with no DATA token on the same line, then the filehandle is closed.

The **SelfLoader** reads from wherever the current position of the `FOOBAR::DATA` filehandle is, until the EOF or `__END__`. This means that if you want to use that filehandle (and ONLY if you want to), you should either

1. Put all your subroutine declarations immediately after the `__DATA__` token and put your own data after those declarations, using the `__END__` token to mark the end of subroutine declarations. You must also ensure that the **SelfLoader** reads first by calling 'SelfLoader->load_stubs();', or by using a function which is selfloaded;

or

2. You should read the `FOOBAR::DATA` filehandle first, leaving the handle open and positioned at the first line of subroutine declarations.

You could conceivably do both.

Classes and inherited methods.

For modules which are not classes, this section is not relevant. This section is only relevant if you have methods which could be inherited.

A subroutine stub (or forward declaration) looks like

```
sub stub;
```

i.e. it is a subroutine declaration without the body of the subroutine. For modules which are not classes, there is no real need for stubs as far as autoloading is concerned.

For modules which ARE classes, and need to handle inherited methods, stubs are needed to ensure that the method inheritance mechanism works properly. You can load the stubs into the module at

'require' time, by adding the statement 'SelfLoader->load_stubs();' to the module to do this.

The alternative is to put the stubs in before the __DATA__ token BEFORE releasing the module, and for this purpose the `Devel::SelfStubber` module is available. However this does require the extra step of ensuring that the stubs are in the module. If this is done I strongly recommend that this is done BEFORE releasing the module - it should NOT be done at install time in general.

Multiple packages and fully qualified subroutine names

Subroutines in multiple packages within the same file are supported - but you should note that this requires exporting the `SelfLoader::AUTOLOAD` to every package which requires it. This is done automatically by the **SelfLoader** when it first loads the subs into the cache, but you should really specify it in the initialization before the __DATA__ by putting a 'use SelfLoader' statement in each package.

Fully qualified subroutine names are also supported. For example,

```
__DATA__
sub foo::bar {23}
package baz;
sub dob {32}
```

will all be loaded correctly by the **SelfLoader**, and the **SelfLoader** will ensure that the packages 'foo' and 'baz' correctly have the **SelfLoader** `AUTOLOAD` method when the data after __DATA__ is first parsed.

AUTHOR

`SelfLoader` is maintained by the perl5-porters. Please direct any questions to the canonical mailing list. Anything that is applicable to the CPAN release can be sent to its maintainer, though.

Author and Maintainer: The Perl5-Porters <perl5-porters@perl.org>

Maintainer of the CPAN release: Steffen Mueller <smueller@cpan.org>

COPYRIGHT AND LICENSE

This package has been part of the perl core since the first release of perl5. It has been released separately to CPAN so older installations can benefit from bug fixes.

This package has the same copyright and license as the perl core:

```
    the GNU General Public License or the Artistic License for more
details.

    You should have received a copy of the Artistic License with this
    Kit, in the file named "Artistic".  If not, I'll be glad to provide
one.

    You should also have received a copy of the GNU General Public License
    along with this program in the file named "Copying". If not, write to
the
    Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston,
    MA 02110-1301, USA or visit their web page on the internet at
    http://www.gnu.org/copyleft/gpl.html.

    For those of you that choose to use the GNU General Public License,
    my interpretation of the GNU General Public License is that no Perl
    script falls under the terms of the GPL unless you explicitly put
    said script under the terms of the GPL yourself.  Furthermore, any
    object code linked with perl does not automatically fall under the
    terms of the GPL, provided such object code only adds definitions
    of subroutines and variables, and does not otherwise impair the
    resulting interpreter from executing any standard Perl script.  I
    consider linking in C subroutines in this manner to be the moral
    equivalent of defining subroutines in the Perl language itself.  You
    may sell such an object file as proprietary provided that you provide
    or offer to provide the Perl source, as specified by the GNU General
    Public License.  (This is merely an alternate way of specifying input
    to the program.)  You may also sell a binary produced by the dumping of
    a running Perl script that belongs to you, provided that you provide or
    offer to provide the Perl source as specified by the GPL.  (The
    fact that a Perl interpreter and your code are in the same binary file
    is, in this case, a form of mere aggregation.)  This is my
interpretation
    of the GPL.  If you still have concerns or difficulties understanding
    my intent, feel free to contact me.  Of course, the Artistic License
    spells all this out for your protection, so you may prefer to use that.
```