NAME

Data::Dumper - stringified perl data structures, suitable for both printing and `eval`

SYNOPSIS

```
use Data::Dumper;

# simple procedural interface
print Dumper($foo, $bar);

# extended usage with names
print Data::Dumper->Dump([$foo, $bar], [qw(foo *ary)]);

# configuration variables
{
  local $Data::Dumper::Purity = 1;
  eval Data::Dumper->Dump([$foo, $bar], [qw(foo *ary)]);
}

# OO usage
$d = Data::Dumper->new([$foo, $bar], [qw(foo *ary)]);
   ...
print $d->Dump;
   ...
$d->Purity(1)->Terse(1)->Deepcopy(1);
eval $d->Dump;
```

DESCRIPTION

Given a list of scalars or reference variables, writes out their contents in perl syntax. The references can also be objects. The content of each variable is output in a single Perl statement. Handles self-referential structures correctly.

The return value can be `eval`ed to get back an identical copy of the original reference structure. (Please do consider the security implications of eval'ing code from untrusted sources!)

Any references that are the same as one of those passed in will be named $VAR*n* (where *n* is a numeric suffix), and other duplicate references to substructures within $VAR*n* will be appropriately labeled using arrow notation. You can specify names for individual values to be dumped if you use the `Dump()` method, or you can change the default $VAR prefix to something else. See `$Data::Dumper::Varname` and `$Data::Dumper::Terse` below.

The default output of self-referential structures can be `eval`ed, but the nested references to $VAR*n* will be undefined, since a recursive structure cannot be constructed using one Perl statement. You should set the `Purity` flag to 1 to get additional statements that will correctly fill in these references. Moreover, if `eval`ed when strictures are in effect, you need to ensure that any variables it accesses are previously declared.

In the extended usage form, the references to be dumped can be given user-specified names. If a name begins with a `*`, the output will describe the dereferenced type of the supplied reference for hashes and arrays, and coderefs. Output of names will be avoided where possible if the `Terse` flag is set.

In many cases, methods that are used to set the internal state of the object will return the object itself, so method calls can be conveniently chained together.

Several styles of output are possible, all controlled by setting the `Indent` flag. See *Configuration Variables or Methods* below for details.

Methods

PACKAGE->new(*ARRAYREF [*, *ARRAYREF]*)

Returns a newly created `Data::Dumper` object. The first argument is an anonymous array of values to be dumped. The optional second argument is an anonymous array of names for the values. The names need not have a leading $ sign, and must be comprised of alphanumeric characters. You can begin a name with a * to specify that the dereferenced type must be dumped instead of the reference itself, for ARRAY and HASH references.

The prefix specified by `$Data::Dumper::Varname` will be used with a numeric suffix if the name for a value is undefined.

Data::Dumper will catalog all references encountered while dumping the values. Cross-references (in the form of names of substructures in perl syntax) will be inserted at all possible points, preserving any structural interdependencies in the original set of values. Structure traversal is depth-first, and proceeds in order from the first supplied value to the last.

$OBJ->Dump *or PACKAGE*->Dump(*ARRAYREF [*, *ARRAYREF]*)

Returns the stringified form of the values stored in the object (preserving the order in which they were supplied to new), subject to the configuration options below. In a list context, it returns a list of strings corresponding to the supplied values.

The second form, for convenience, simply calls the new method on its arguments before dumping the object immediately.

$OBJ->Seen(*[HASHREF]*)

Queries or adds to the internal table of already encountered references. You must use `Reset` to explicitly clear the table if needed. Such references are not dumped; instead, their names are inserted wherever they are encountered subsequently. This is useful especially for properly dumping subroutine references.

Expects an anonymous hash of name => value pairs. Same rules apply for names as in new. If no argument is supplied, will return the "seen" list of name => value pairs, in a list context. Otherwise, returns the object itself.

$OBJ->Values(*[ARRAYREF]*)

Queries or replaces the internal array of values that will be dumped. When called without arguments, returns the values as a list. When called with a reference to an array of replacement values, returns the object itself. When called with any other type of argument, dies.

$OBJ->Names(*[ARRAYREF]*)

Queries or replaces the internal array of user supplied names for the values that will be dumped. When called without arguments, returns the names. When called with an array of replacement names, returns the object itself. If the number of replacement names exceeds the number of values to be named, the excess names will not be used. If the number of replacement names falls short of the number of values to be named, the list of replacement names will be exhausted and remaining values will not be renamed. When called with any other type of argument, dies.

$OBJ->Reset

Clears the internal table of "seen" references and returns the object itself.

Functions

Dumper(*LIST*)

Returns the stringified form of the values in the list, subject to the configuration options below. The values will be named $VAR*n* in the output, where *n* is a numeric suffix. Will return a list of strings in a list context.

Configuration Variables or Methods

Several configuration variables can be used to control the kind of output generated when using the procedural interface. These variables are usually `localized` in a block so that other parts of the code are not affected by the change.

These variables determine the default state of the object created by calling the `new` method, but cannot be used to alter the state of the object thereafter. The equivalent method names should be used instead to query or set the internal state of the object.

The method forms return the object itself when called with arguments, so that they can be chained together nicely.

- $Data::Dumper::Indent *or $OBJ*->Indent(*[NEWVAL]*)

  Controls the style of indentation. It can be set to 0, 1, 2 or 3. Style 0 spews output without any newlines, indentation, or spaces between list items. It is the most compact format possible that can still be called valid perl. Style 1 outputs a readable form with newlines but no fancy indentation (each level in the structure is simply indented by a fixed amount of whitespace). Style 2 (the default) outputs a very readable form which takes into account the length of hash keys (so the hash value lines up). Style 3 is like style 2, but also annotates the elements of arrays with their index (but the comment is on its own line, so array output consumes twice the number of lines). Style 2 is the default.

- $Data::Dumper::Purity *or $OBJ*->Purity(*[NEWVAL]*)

  Controls the degree to which the output can be `eval`ed to recreate the supplied reference structures. Setting it to 1 will output additional perl statements that will correctly recreate nested references. The default is 0.

- $Data::Dumper::Pad *or $OBJ*->Pad(*[NEWVAL]*)

  Specifies the string that will be prefixed to every line of the output. Empty string by default.

- $Data::Dumper::Varname *or $OBJ*->Varname(*[NEWVAL]*)

  Contains the prefix to use for tagging variable names in the output. The default is "VAR".

- $Data::Dumper::Useqq *or $OBJ*->Useqq(*[NEWVAL]*)

  When set, enables the use of double quotes for representing string values. Whitespace other than space will be represented as `[\n\t\r]`, "unsafe" characters will be backslashed, and unprintable characters will be output as quoted octal integers. The default is 0.

- $Data::Dumper::Terse *or $OBJ*->Terse(*[NEWVAL]*)

  When set, Data::Dumper will emit single, non-self-referential values as atoms/terms rather than statements. This means that the $VAR*n* names will be avoided where possible, but be advised that such output may not always be parseable by `eval`.

- $Data::Dumper::Freezer *or* $*OBJ*->Freezer(*[NEWVAL]*)

  Can be set to a method name, or to an empty string to disable the feature. Data::Dumper will invoke that method via the object before attempting to stringify it. This method can alter the contents of the object (if, for instance, it contains data allocated from C), and even rebless it in a different package. The client is responsible for making sure the specified method can be called via the object, and that the object ends up containing only perl data types after the method has been called. Defaults to an empty string.

  If an object does not support the method specified (determined using UNIVERSAL::can()) then the call will be skipped. If the method dies a warning will be generated.

- $Data::Dumper::Toaster *or* $*OBJ*->Toaster(*[NEWVAL]*)

  Can be set to a method name, or to an empty string to disable the feature. Data::Dumper will emit a method call for any objects that are to be dumped using the syntax `bless(DATA,`

`CLASS)->METHOD()`. Note that this means that the method specified will have to perform any modifications required on the object (like creating new state within it, and/or reblessing it in a different package) and then return it. The client is responsible for making sure the method can be called via the object, and that it returns a valid object. Defaults to an empty string.

- $Data::Dumper::Deepcopy *or* $*OBJ*->Deepcopy(*[NEWVAL]*)

  Can be set to a boolean value to enable deep copies of structures. Cross-referencing will then only be done when absolutely essential (i.e., to break reference cycles). Default is 0.

- $Data::Dumper::Quotekeys *or* $*OBJ*->Quotekeys(*[NEWVAL]*)

  Can be set to a boolean value to control whether hash keys are quoted. A defined false value will avoid quoting hash keys when it looks like a simple string. Default is 1, which will always enclose hash keys in quotes.

- $Data::Dumper::Bless *or* $*OBJ*->Bless(*[NEWVAL]*)

  Can be set to a string that specifies an alternative to the `bless` builtin operator used to create objects. A function with the specified name should exist, and should accept the same arguments as the builtin. Default is `bless`.

- $Data::Dumper::Pair *or* $*OBJ*->Pair(*[NEWVAL]*)

  Can be set to a string that specifies the separator between hash keys and values. To dump nested hash, array and scalar values to JavaScript, use: `$Data::Dumper::Pair = ' : ';`. Implementing `bless` in JavaScript is left as an exercise for the reader. A function with the specified name exists, and accepts the same arguments as the builtin.

  Default is: => .

- $Data::Dumper::Maxdepth *or* $*OBJ*->Maxdepth(*[NEWVAL]*)

  Can be set to a positive integer that specifies the depth beyond which we don't venture into a structure. Has no effect when `Data::Dumper::Purity` is set. (Useful in debugger when we often don't want to see more than enough). Default is 0, which means there is no maximum depth.

- $Data::Dumper::Maxrecurse *or* $*OBJ*->Maxrecurse(*[NEWVAL]*)

  Can be set to a positive integer that specifies the depth beyond which recursion into a structure will throw an exception. This is intended as a security measure to prevent perl running out of stack space when dumping an excessively deep structure. Can be set to 0 to remove the limit. Default is 1000.

- $Data::Dumper::Useperl *or* $*OBJ*->Useperl(*[NEWVAL]*)

  Can be set to a boolean value which controls whether the pure Perl implementation of `Data::Dumper` is used. The `Data::Dumper` module is a dual implementation, with almost all functionality written in both pure Perl and also in XS ('C'). Since the XS version is much faster, it will always be used if possible. This option lets you override the default behavior, usually for testing purposes only. Default is 0, which means the XS implementation will be used if possible.

- $Data::Dumper::Sortkeys *or* $*OBJ*->Sortkeys(*[NEWVAL]*)

  Can be set to a boolean value to control whether hash keys are dumped in sorted order. A true value will cause the keys of all hashes to be dumped in Perl's default sort order. Can also be set to a subroutine reference which will be called for each hash that is dumped. In this case `Data::Dumper` will call the subroutine once for each hash, passing it the reference of the hash. The purpose of the subroutine is to return a reference to an array of the keys that will be dumped, in the order that they should be dumped. Using this feature, you can control both the order of the keys, and which keys are actually used. In other words, this subroutine acts as a filter by which you can exclude certain keys from being dumped. Default is 0, which means that hash keys are not sorted.

- $Data::Dumper::Deparse *or* $*OBJ*->Deparse(*[NEWVAL]*)

  Can be set to a boolean value to control whether code references are turned into perl source code. If set to a true value, `B::Deparse` will be used to get the source of the code reference. Using this option will force using the Perl implementation of the dumper, since the fast XSUB implementation doesn't support it.

  Caution : use this option only if you know that your coderefs will be properly reconstructed by `B::Deparse`.

- $Data::Dumper::Sparseseen *or* $*OBJ*->Sparseseen(*[NEWVAL]*)

  By default, Data::Dumper builds up the "seen" hash of scalars that it has encountered during serialization. This is very expensive. This seen hash is necessary to support and even just detect circular references. It is exposed to the user via the `Seen()` call both for writing and reading.

  If you, as a user, do not need explicit access to the "seen" hash, then you can set the `Sparseseen` option to allow Data::Dumper to eschew building the "seen" hash for scalars that are known not to possess more than one reference. This speeds up serialization considerably if you use the XS implementation.

  Note: If you turn on `Sparseseen`, then you must not rely on the content of the seen hash since its contents will be an implementation detail!

Exports

Dumper

EXAMPLES

Run these code snippets to get a quick feel for the behavior of this module. When you are through with these examples, you may want to add or change the various configuration variables described above, to see their behavior. (See the testsuite in the Data::Dumper distribution for more examples.)

```
use Data::Dumper;

package Foo;
sub new {bless {'a' => 1, 'b' => sub { return "foo" }}, $_[0]};

package Fuz;                       # a weird REF-REF-SCALAR object
sub new {bless \($_ = \ 'fu\'z'), $_[0]};

package main;
$foo = Foo->new;
$fuz = Fuz->new;
$boo = [ 1, [], "abcd", \*foo,
         {1 => 'a', 023 => 'b', 0x45 => 'c'},
         \\"p\q\'r", $foo, $fuz];

########
# simple usage
########

$bar = eval(Dumper($boo));
print($@) if $@;
print Dumper($boo), Dumper($bar);  # pretty print (no array indices)

$Data::Dumper::Terse = 1;        # don't output names where feasible
$Data::Dumper::Indent = 0;       # turn off all pretty print
```

```
print Dumper($boo), "\n";

$Data::Dumper::Indent = 1;       # mild pretty print
print Dumper($boo);

$Data::Dumper::Indent = 3;       # pretty print with array indices
print Dumper($boo);

$Data::Dumper::Useqq = 1;        # print strings in double quotes
print Dumper($boo);

$Data::Dumper::Pair = " : ";     # specify hash key/value separator
print Dumper($boo);

########
# recursive structures
########

@c = ('c');
$c = \@c;
$b = {};
$a = [1, $b, $c];
$b->{a} = $a;
$b->{b} = $a->[1];
$b->{c} = $a->[2];
print Data::Dumper->Dump([$a,$b,$c], [qw(a b c)]);

$Data::Dumper::Purity = 1;         # fill in the holes for eval
print Data::Dumper->Dump([$a, $b], [qw(*a b)]); # print as @a
print Data::Dumper->Dump([$b, $a], [qw(*b a)]); # print as %b

$Data::Dumper::Deepcopy = 1;       # avoid cross-refs
print Data::Dumper->Dump([$b, $a], [qw(*b a)]);

$Data::Dumper::Purity = 0;         # avoid cross-refs
print Data::Dumper->Dump([$b, $a], [qw(*b a)]);

########
# deep structures
########

$a = "pearl";
$b = [ $a ];
$c = { 'b' => $b };
$d = [ $c ];
$e = { 'd' => $d };
$f = { 'e' => $e };
print Data::Dumper->Dump([$f], [qw(f)]);

$Data::Dumper::Maxdepth = 3;       # no deeper than 3 refs down
print Data::Dumper->Dump([$f], [qw(f)]);
```

```
########
# object-oriented usage
########

$d = Data::Dumper->new([$a,$b], [qw(a b)]);
$d->Seen({'*c' => $c});            # stash a ref without printing it
$d->Indent(3);
print $d->Dump;
$d->Reset->Purity(0);              # empty the seen cache
print join "----\n", $d->Dump;

########
# persistence
########

package Foo;
sub new { bless { state => 'awake' }, shift }
sub Freeze {
    my $s = shift;
    print STDERR "preparing to sleep\n";
    $s->{state} = 'asleep';
    return bless $s, 'Foo::ZZZ';
}

package Foo::ZZZ;
sub Thaw {
    my $s = shift;
    print STDERR "waking up\n";
    $s->{state} = 'awake';
    return bless $s, 'Foo';
}

package main;
use Data::Dumper;
$a = Foo->new;
$b = Data::Dumper->new([$a], ['c']);
$b->Freezer('Freeze');
$b->Toaster('Thaw');
$c = $b->Dump;
print $c;
$d = eval $c;
print Data::Dumper->Dump([$d], ['d']);

########
# symbol substitution (useful for recreating CODE refs)
########

sub foo { print "foo speaking\n" }
*other = \&foo;
$bar = [ \&other ];
$d = Data::Dumper->new([\&other,$bar],['*other','bar']);
$d->Seen({ '*foo' => \&foo });
print $d->Dump;
```

```
########
# sorting and filtering hash keys
########

$Data::Dumper::Sortkeys = \&my_filter;
my $foo = { map { (ord, "$_$_$_") } 'I'..'Q' };
my $bar = { %$foo };
my $baz = { reverse %$foo };
print Dumper [ $foo, $bar, $baz ];

sub my_filter {
    my ($hash) = @_;
    # return an array ref containing the hash keys to dump
    # in the order that you want them to be dumped
    return [
      # Sort the keys of %$foo in reverse numeric order
        $hash eq $foo ? (sort {$b <=> $a} keys %$hash) :
      # Only dump the odd number keys of %$bar
        $hash eq $bar ? (grep {$_ % 2} keys %$hash) :
      # Sort keys in default order for all other hashes
        (sort keys %$hash)
    ];
}
```

BUGS

Due to limitations of Perl subroutine call semantics, you cannot pass an array or hash. Prepend it with a \ to pass its reference instead. This will be remedied in time, now that Perl has subroutine prototypes. For now, you need to use the extended usage form, and prepend the name with a * to output it as a hash or array.

`Data::Dumper` cheats with CODE references. If a code reference is encountered in the structure being processed (and if you haven't set the `Deparse` flag), an anonymous subroutine that contains the string '"DUMMY"' will be inserted in its place, and a warning will be printed if `Purity` is set. You can `eval` the result, but bear in mind that the anonymous sub that gets created is just a placeholder. Someday, perl will have a switch to cache-on-demand the string representation of a compiled piece of code, I hope. If you have prior knowledge of all the code refs that your data structures are likely to have, you can use the `Seen` method to pre-seed the internal reference table and make the dumped output point to them, instead. See *EXAMPLES* above.

The `Deparse` flag makes Dump() run slower, since the XSUB implementation does not support it.

SCALAR objects have the weirdest looking `bless` workaround.

Pure Perl version of `Data::Dumper` escapes UTF-8 strings correctly only in Perl 5.8.0 and later.

NOTE

Starting from Perl 5.8.1 different runs of Perl will have different ordering of hash keys. The change was done for greater security, see *"Algorithmic Complexity Attacks" in perlsec*. This means that different runs of Perl will have different Data::Dumper outputs if the data contains hashes. If you need to have identical Data::Dumper outputs from different runs of Perl, use the environment variable PERL_HASH_SEED, see *"PERL_HASH_SEED" in perlrun*. Using this restores the old (platform-specific) ordering: an even prettier solution might be to use the `Sortkeys` filter of Data::Dumper.

AUTHOR

Gurusamy Sarathy gsar@activestate.com

VERSION

Version 2.158 (March 13 2015)

SEE ALSO

perl(1)