NAME

Devel::Peek - A data debugging tool for the XS programmer

SYNOPSIS

```
        use Devel::Peek;
        Dump( $a );
        Dump( $a, 5 );
        DumpArray( 5, $a, $b, ... );
mstat "Point 5";

        use Devel::Peek ':opd=st';
```

DESCRIPTION

Devel::Peek contains functions which allows raw Perl datatypes to be manipulated from a Perl script. This is used by those who do XS programming to check that the data they are sending from C to Perl looks as they think it should look. The trick, then, is to know what the raw datatype is supposed to look like when it gets to Perl. This document offers some tips and hints to describe good and bad raw data.

It is very possible that this document will fall far short of being useful to the casual reader. The reader is expected to understand the material in the first few sections of *perlguts*.

Devel::Peek supplies a `Dump()` function which can dump a raw Perl datatype, and `mstat("marker")` function to report on memory usage (if perl is compiled with corresponding option). The function DeadCode() provides statistics on the data "frozen" into inactive `CV`. Devel::Peek also supplies `SvREFCNT()`, `SvREFCNT_inc()`, and `SvREFCNT_dec()` which can query, increment, and decrement reference counts on SVs. This document will take a passive, and safe, approach to data debugging and for that it will describe only the `Dump()` function.

Function `DumpArray()` allows dumping of multiple values (useful when you need to analyze returns of functions).

The global variable $Devel::Peek::pv_limit can be set to limit the number of character printed in various string values. Setting it to 0 means no limit.

If `use Devel::Peek` directive has a `:opd=FLAGS` argument, this switches on debugging of opcode dispatch. `FLAGS` should be a combination of `s`, `t`, and `P` (see **-D** flags in *perlrun*). `:opd` is a shortcut for `:opd=st`.

Runtime debugging

`CvGV($cv)` return one of the globs associated to a subroutine reference $cv.

debug_flags() returns a string representation of `$^D` (similar to what is allowed for **-D** flag). When called with a numeric argument, sets $^D to the corresponding value. When called with an argument of the form `"flags-flags"`, set on/off bits of `$^D` corresponding to letters before/after `-`. (The returned value is for `$^D` before the modification.)

runops_debug() returns true if the current *opcode dispatcher* is the debugging one. When called with an argument, switches to debugging or non-debugging dispatcher depending on the argument (active for newly-entered subs/etc only). (The returned value is for the dispatcher before the modification.)

Memory footprint debugging

When perl is compiled with support for memory footprint debugging (default with Perl's malloc()), Devel::Peek provides an access to this API.

Use mstat() function to emit a memory state statistic to the terminal. For more information on the format of output of mstat() see *"Using $ENV{PERL_DEBUG_MSTATS}" in perldebguts*.

Three additional functions allow access to this statistic from Perl. First, use `mstats_fillhash(%hash)` to get the information contained in the output of mstat() into %hash. The field of this hash are

```
minbucket nbuckets sbrk_good sbrk_slack sbrked_remains sbrks start_slack
topbucket topbucket_ev topbucket_odd total total_chain total_sbrk totfree
```

Two additional fields `free`, `used` contain array references which provide per-bucket count of free and used chunks. Two other fields `mem_size`, `available_size` contain array references which provide the information about the allocated size and usable size of chunks in each bucket. Again, see *"Using $ENV{PERL_DEBUG_MSTATS}" in perldebguts* for details.

Keep in mind that only the first several "odd-numbered" buckets are used, so the information on size of the "odd-numbered" buckets which are not used is probably meaningless.

The information in

```
mem_size available_size minbucket nbuckets
```

is the property of a particular build of perl, and does not depend on the current process. If you do not provide the optional argument to the functions mstats_fillhash(), fill_mstats(), mstats2hash(), then the information in fields `mem_size`, `available_size` is not updated.

`fill_mstats($buf)` is a much cheaper call (both speedwise and memory-wise) which collects the statistic into $buf in machine-readable form. At a later moment you may need to call `mstats2hash($buf, %hash)` to use this information to fill %hash.

All three APIs `fill_mstats($buf)`, `mstats_fillhash(%hash)`, and `mstats2hash($buf, %hash)` are designed to allocate no memory if used *the second time* on the same $buf and/or %hash.

So, if you want to collect memory info in a cycle, you may call

```
$#buf = 999;
fill_mstats($_) for @buf;
mstats_fillhash(%report, 1);  # Static info too

foreach (@buf) {
  # Do something...
  fill_mstats $_;   # Collect statistic
}
foreach (@buf) {
  mstats2hash($_, %report);  # Preserve static info
  # Do something with %report
}
```

EXAMPLES

The following examples don't attempt to show everything as that would be a monumental task, and, frankly, we don't want this manpage to be an internals document for Perl. The examples do demonstrate some basics of the raw Perl datatypes, and should suffice to get most determined people on their way. There are no guidewires or safety nets, nor blazed trails, so be prepared to travel alone from this point and on and, if at all possible, don't fall into the quicksand (it's bad for business).

Oh, one final bit of advice: take *perlguts* with you. When you return we expect to see it well-thumbed.

A simple scalar string

Let's begin by looking a simple scalar which is holding a string.

```
use Devel::Peek;
$a = "hello";
Dump $a;
```

The output:

```
SV = PVIV(0xbc288)
  REFCNT = 1
  FLAGS = (POK,pPOK)
  IV = 0
  PV = 0xb2048 "hello"\0
  CUR = 5
  LEN = 6
```

This says $a is an SV, a scalar. The scalar is a PVIV, a string. Its reference count is 1. It has the POK flag set, meaning its current PV field is valid. Because POK is set we look at the PV item to see what is in the scalar. The \0 at the end indicate that this PV is properly NUL-terminated. If the FLAGS had been IOK we would look at the IV item. CUR indicates the number of characters in the PV. LEN indicates the number of bytes requested for the PV (one more than CUR, in this case, because LEN includes an extra byte for the end-of-string marker).

A simple scalar number

If the scalar contains a number the raw SV will be leaner.

```
use Devel::Peek;
$a = 42;
Dump $a;
```

The output:

```
SV = IV(0xbc818)
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 42
```

This says $a is an SV, a scalar. The scalar is an IV, a number. Its reference count is 1. It has the IOK flag set, meaning it is currently being evaluated as a number. Because IOK is set we look at the IV item to see what is in the scalar.

A simple scalar with an extra reference

If the scalar from the previous example had an extra reference:

```
use Devel::Peek;
$a = 42;
$b = \$a;
Dump $a;
```

The output:

```
SV = IV(0xbe860)
  REFCNT = 2
  FLAGS = (IOK,pIOK)
  IV = 42
```

Notice that this example differs from the previous example only in its reference count. Compare this to the next example, where we dump $b instead of $a.

A reference to a simple scalar

This shows what a reference looks like when it references a simple scalar.

```
use Devel::Peek;
$a = 42;
$b = \$a;
Dump $b;
```

The output:

```
SV = RV(0xf041c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xbab08
SV = IV(0xbe860)
  REFCNT = 2
  FLAGS = (IOK,pIOK)
  IV = 42
```

Starting from the top, this says $b is an SV. The scalar is an RV, a reference. It has the ROK flag set, meaning it is a reference. Because ROK is set we have an RV item rather than an IV or PV. Notice that Dump follows the reference and shows us what $b was referencing. We see the same $a that we found in the previous example.

Note that the value of RV coincides with the numbers we see when we stringify $b. The addresses inside RV() and IV() are addresses of X*** structure which holds the current state of an SV. This address may change during lifetime of an SV.

A reference to an array

This shows what a reference to an array looks like.

```
use Devel::Peek;
$a = [42];
Dump $a;
```

The output:

```
SV = RV(0xf041c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xb2850
SV = PVAV(0xbd448)
  REFCNT = 1
  FLAGS = ()
  IV = 0
  NV = 0
  ARRAY = 0xb2048
  ALLOC = 0xb2048
  FILL = 0
  MAX = 0
  ARYLEN = 0x0
  FLAGS = (REAL)
Elt No. 0 0xb5658
SV = IV(0xbe860)
  REFCNT = 1
  FLAGS = (IOK,pIOK)
```

```
  IV = 42
```

This says $a is an SV and that it is an RV. That RV points to another SV which is a PVAV, an array. The array has one element, element zero, which is another SV. The field FILL above indicates the last element in the array, similar to $#$a.

If $a pointed to an array of two elements then we would see the following.

```
use Devel::Peek 'Dump';
$a = [42,24];
Dump $a;
```

The output:

```
SV = RV(0xf041c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xb2850
SV = PVAV(0xbd448)
  REFCNT = 1
  FLAGS = ()
  IV = 0
  NV = 0
  ARRAY = 0xb2048
  ALLOC = 0xb2048
  FILL = 0
  MAX = 0
  ARYLEN = 0x0
  FLAGS = (REAL)
Elt No. 0  0xb5658
SV = IV(0xbe860)
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 42
Elt No. 1  0xb5680
SV = IV(0xbe818)
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 24
```

Note that Dump will not report *all* the elements in the array, only several first (depending on how deep it already went into the report tree).

A reference to a hash

The following shows the raw form of a reference to a hash.

```
use Devel::Peek;
$a = {hello=>42};
Dump $a;
```

The output:

```
SV = RV(0x8177858) at 0x816a618
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0x814fc10
```

```
SV = PVHV(0x8167768) at 0x814fc10
  REFCNT = 1
  FLAGS = (SHAREKEYS)
  IV = 1
  NV = 0
  ARRAY = 0x816c5b8  (0:7, 1:1)
  hash quality = 100.0%
  KEYS = 1
  FILL = 1
  MAX = 7
  RITER = -1
  EITER = 0x0
  Elt "hello" HASH = 0xc8fd181b
  SV = IV(0x816c030) at 0x814fcf4
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 42
```

This shows $a is a reference pointing to an SV. That SV is a PVHV, a hash. Fields RITER and EITER are used by *each*.

The "quality" of a hash is defined as the total number of comparisons needed to access every element once, relative to the expected number needed for a random hash. The value can go over 100%.

The total number of comparisons is equal to the sum of the squares of the number of entries in each bucket. For a random hash of <n> keys into <k> buckets, the expected value is:

```
n + n(n-1)/2k
```

Dumping a large array or hash

The `Dump()` function, by default, dumps up to 4 elements from a toplevel array or hash. This number can be increased by supplying a second argument to the function.

```
use Devel::Peek;
$a = [10,11,12,13,14];
Dump $a;
```

Notice that `Dump()` prints only elements 10 through 13 in the above code. The following code will print all of the elements.

```
use Devel::Peek 'Dump';
$a = [10,11,12,13,14];
Dump $a, 5;
```

A reference to an SV which holds a C pointer

This is what you really need to know as an XS programmer, of course. When an XSUB returns a pointer to a C structure that pointer is stored in an SV and a reference to that SV is placed on the XSUB stack. So the output from an XSUB which uses something like the T_PTROBJ map might look something like this:

```
SV = RV(0xf381c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xb8ad8
SV = PVMG(0xbb3c8)
```

```
  REFCNT = 1
  FLAGS = (OBJECT,IOK,pIOK)
  IV = 729160
  NV = 0
  PV = 0
  STASH = 0xc1d10       "CookBookB::Opaque"
```

This shows that we have an SV which is an RV. That RV points at another SV. In this case that second SV is a PVMG, a blessed scalar. Because it is blessed it has the `OBJECT` flag set. Note that an SV which holds a C pointer also has the `IOK` flag set. The `STASH` is set to the package name which this SV was blessed into.

The output from an XSUB which uses something like the T_PTRREF map, which doesn't bless the object, might look something like this:

```
SV = RV(0xf381c)
  REFCNT = 1
  FLAGS = (ROK)
  RV = 0xb8ad8
SV = PVMG(0xbb3c8)
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 729160
  NV = 0
  PV = 0
```

A reference to a subroutine

Looks like this:

```
SV = RV(0x798ec)
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x1d453c
SV = PVCV(0x1c768c)
  REFCNT = 2
  FLAGS = ()
  IV = 0
  NV = 0
  COMP_STASH = 0x31068  "main"
  START = 0xb20e0
  ROOT = 0xbece0
  XSUB = 0x0
  XSUBANY = 0
  GVGV::GV = 0x1d44e8   "MY" :: "top_targets"
  FILE = "(eval 5)"
  DEPTH = 0
  PADLIST = 0x1c9338
```

This shows that

- the subroutine is not an XSUB (since `START` and `ROOT` are non-zero, and `XSUB` is zero);
- that it was compiled in the package `main`;
- under the name `MY::top_targets`;
- inside a 5th eval in the program;

- it is not currently executed (see DEPTH);
- it has no prototype (PROTOTYPE field is missing).

EXPORTS

Dump, mstat, DeadCode, DumpArray, DumpWithOP and DumpProg, fill_mstats, mstats_fillhash, mstats2hash by default. Additionally available SvREFCNT, SvREFCNT_inc and SvREFCNT_dec.

BUGS

Readers have been known to skip important parts of *perlguts*, causing much frustration for all.

AUTHOR

Ilya Zakharevich ilya@math.ohio-state.edu

SEE ALSO

perlguts, and *perlguts*, again.