NAME

perlhacktips - Tips for Perl core C code hacking

DESCRIPTION

This document will help you learn the best way to go about hacking on the Perl core C code. It covers common problems, debugging, profiling, and more.

If you haven't read *perlhack* and *perlhacktut* yet, you might want to do that first.

COMMON PROBLEMS

Perl source plays by ANSI C89 rules: no C99 (or C++) extensions. In some cases we have to take pre-ANSI requirements into consideration. You don't care about some particular platform having broken Perl? I hear there is still a strong demand for J2EE programmers.

Perl environment problems

- Not compiling with threading

  Compiling with threading (-Duseithreads) completely rewrites the function prototypes of Perl. You better try your changes with that. Related to this is the difference between "Perl_-less" and "Perl_-ly" APIs, for example:

  ```
  Perl_sv_setiv(aTHX_ ...);
  sv_setiv(...);
  ```

  The first one explicitly passes in the context, which is needed for e.g. threaded builds. The second one does that implicitly; do not get them mixed. If you are not passing in a `aTHX_`, you will need to do a dTHX (or a dVAR) as the first thing in the function.

  See *"How multiple interpreters and concurrency are supported" in perlguts* for further discussion about context.

- Not compiling with -DDEBUGGING

  The DEBUGGING define exposes more code to the compiler, therefore more ways for things to go wrong. You should try it.

- Introducing (non-read-only) globals

  Do not introduce any modifiable globals, truly global or file static. They are bad form and complicate multithreading and other forms of concurrency. The right way is to introduce them as new interpreter variables, see *intrpvar.h* (at the very end for binary compatibility).

  Introducing read-only (const) globals is okay, as long as you verify with e.g. `nm libperl.a|egrep -v ' [TURtr] '` (if your `nm` has BSD-style output) that the data you added really is read-only. (If it is, it shouldn't show up in the output of that command.)

  If you want to have static strings, make them constant:

  ```
  static const char etc[] = "...";
  ```

  If you want to have arrays of constant strings, note carefully the right combination of `consts`:

  ```
      static const char * const yippee[] =
          {"hi", "ho", "silver"};
  ```

  There is a way to completely hide any modifiable globals (they are all moved to heap), the compilation setting `-DPERL_GLOBAL_STRUCT_PRIVATE`. It is not normally used, but can be used for testing, read more about it in *"Background and PERL_IMPLICIT_CONTEXT" in perlguts*.

- Not exporting your new function

  Some platforms (Win32, AIX, VMS, OS/2, to name a few) require any function that is part of the public API (the shared Perl library) to be explicitly marked as exported. See the discussion

about *embed.pl* in *perlguts*.

- Exporting your new function

  The new shiny result of either genuine new functionality or your arduous refactoring is now ready and correctly exported. So what could possibly go wrong?

  Maybe simply that your function did not need to be exported in the first place. Perl has a long and not so glorious history of exporting functions that it should not have.

  If the function is used only inside one source code file, make it static. See the discussion about *embed.pl* in *perlguts*.

  If the function is used across several files, but intended only for Perl's internal use (and this should be the common case), do not export it to the public API. See the discussion about *embed.pl* in *perlguts*.

Portability problems

The following are common causes of compilation and/or execution failures, not common to Perl as such. The C FAQ is good bedtime reading. Please test your changes with as many C compilers and platforms as possible; we will, anyway, and it's nice to save oneself from public embarrassment.

If using gcc, you can add the `-std=c89` option which will hopefully catch most of these unportabilities. (However it might also catch incompatibilities in your system's header files.)

Use the Configure `-Dgccansipedantic` flag to enable the gcc `-ansi -pedantic` flags which enforce stricter ANSI rules.

If using the `gcc -Wall` note that not all the possible warnings (like `-Wuninitialized`) are given unless you also compile with `-O`.

Note that if using gcc, starting from Perl 5.9.5 the Perl core source code files (the ones at the top level of the source code distribution, but not e.g. the extensions under ext/) are automatically compiled with as many as possible of the `-std=c89`, `-ansi`, `-pedantic`, and a selection of `-W` flags (see cflags.SH).

Also study *perlport* carefully to avoid any bad assumptions about the operating system, filesystems, character set, and so forth.

You may once in a while try a "make microperl" to see whether we can still compile Perl with just the bare minimum of interfaces. (See README.micro.)

Do not assume an operating system indicates a certain compiler.

- Casting pointers to integers or casting integers to pointers

  ```
  void castaway(U8* p)
  {
    IV i = p;
  ```

  or

  ```
  void castaway(U8* p)
  {
    IV i = (IV)p;
  ```

  Both are bad, and broken, and unportable. Use the PTR2IV() macro that does it right. (Likewise, there are PTR2UV(), PTR2NV(), INT2PTR(), and NUM2PTR().)

- Casting between function pointers and data pointers

  Technically speaking casting between function pointers and data pointers is unportable and undefined, but practically speaking it seems to work, but you should use the FPTR2DPTR() and DPTR2FPTR() macros. Sometimes you can also play games with unions.

- Assuming sizeof(int) == sizeof(long)

  There are platforms where longs are 64 bits, and platforms where ints are 64 bits, and while we are out to shock you, even platforms where shorts are 64 bits. This is all legal according to the C standard. (In other words, "long long" is not a portable way to specify 64 bits, and "long long" is not even guaranteed to be any wider than "long".)

  Instead, use the definitions IV, UV, IVSIZE, I32SIZE, and so forth. Avoid things like I32 because they are **not** guaranteed to be *exactly* 32 bits, they are *at least* 32 bits, nor are they guaranteed to be **int** or **long**. If you really explicitly need 64-bit variables, use I64 and U64, but only if guarded by HAS_QUAD.

- Assuming one can dereference any type of pointer for any type of data

  ```
  char *p = ...;
  long pony = *(long *)p;    /* BAD */
  ```

  Many platforms, quite rightly so, will give you a core dump instead of a pony if the p happens not to be correctly aligned.

- Lvalue casts

  ```
  (int)*p = ...;    /* BAD */
  ```

  Simply not portable. Get your lvalue to be of the right type, or maybe use temporary variables, or dirty tricks with unions.

- Assume **anything** about structs (especially the ones you don't control, like the ones coming from the system headers)
  - That a certain field exists in a struct
  - That no other fields exist besides the ones you know of
  - That a field is of certain signedness, sizeof, or type
  - That the fields are in a certain order
    - While C guarantees the ordering specified in the struct definition, between different platforms the definitions might differ
  - That the sizeof(struct) or the alignments are the same everywhere
    - There might be padding bytes between the fields to align the fields - the bytes can be anything
    - Structs are required to be aligned to the maximum alignment required by the fields - which for native types is for usually equivalent to sizeof() of the field

- Assuming the character set is ASCIIish

  Perl can compile and run under EBCDIC platforms. See *perlebcdic*. This is transparent for the most part, but because the character sets differ, you shouldn't use numeric (decimal, octal, nor hex) constants to refer to characters. You can safely say `'A'`, but not `0x41`. You can safely say `'\n'`, but not `\012`. However, you can use macros defined in *utf8.h* to specify any code point portably. `LATIN1_TO_NATIVE(0xDF)` is going to be the code point that means LATIN SMALL LETTER SHARP S on whatever platform you are running on (on ASCII platforms it compiles without adding any extra code, so there is zero performance hit on those). The acceptable inputs to `LATIN1_TO_NATIVE` are from `0x00` through `0xFF`. If your input isn't guaranteed to be in that range, use `UNICODE_TO_NATIVE` instead. `NATIVE_TO_LATIN1` and `NATIVE_TO_UNICODE` translate the opposite direction.

  If you need the string representation of a character that doesn't have a mnemonic name in C,

you should add it to the list in *regen/unicode_constants.pl*, and have Perl create `#define`'s for you, based on the current platform.

Note that the `is`*F00* and `to`*F00* macros in *handy.h* work properly on native code points and strings.

Also, the range 'A' - 'Z' in ASCII is an unbroken sequence of 26 upper case alphabetic characters. That is not true in EBCDIC. Nor for 'a' to 'z'. But '0' - '9' is an unbroken range in both systems. Don't assume anything about other ranges. (Note that special handling of ranges in regular expression patterns and transliterations makes it appear to Perl code that the aforementioned ranges are all unbroken.)

Many of the comments in the existing code ignore the possibility of EBCDIC, and may be wrong therefore, even if the code works. This is actually a tribute to the successful transparent insertion of being able to handle EBCDIC without having to change pre-existing code.

UTF-8 and UTF-EBCDIC are two different encodings used to represent Unicode code points as sequences of bytes. Macros with the same names (but different definitions) in *utf8.h* and *utfebcdic.h* are used to allow the calling code to think that there is only one such encoding. This is almost always referred to as `utf8`, but it means the EBCDIC version as well. Again, comments in the code may well be wrong even if the code itself is right. For example, the concept of UTF-8 `invariant characters` differs between ASCII and EBCDIC. On ASCII platforms, only characters that do not have the high-order bit set (i.e. whose ordinals are strict ASCII, 0 - 127) are invariant, and the documentation and comments in the code may assume that, often referring to something like, say, `hibit`. The situation differs and is not so simple on EBCDIC machines, but as long as the code itself uses the `NATIVE_IS_INVARIANT()` macro appropriately, it works, even if the comments are wrong.

As noted in *"TESTING" in perlhack*, when writing test scripts, the file *t/charset_tools.pl* contains some helpful functions for writing tests valid on both ASCII and EBCDIC platforms. Sometimes, though, a test can't use a function and it's inconvenient to have different test versions depending on the platform. There are 20 code points that are the same in all 4 character sets currently recognized by Perl (the 3 EBCDIC code pages plus ISO 8859-1 (ASCII/Latin1)). These can be used in such tests, though there is a small possibility that Perl will become available in yet another character set, breaking your test. All but one of these code points are C0 control characters. The most significant controls that are the same are `\0`, `\r`, and `\N{VT}` (also specifiable as `\cK`, `\x0B`, `\N{U+0B}`, or `\013`). The single non-control is U+00B6 PILCROW SIGN. The controls that are the same have the same bit pattern in all 4 character sets, regardless of the UTF8ness of the string containing them. The bit pattern for U+B6 is the same in all 4 for non-UTF8 strings, but differs in each when its containing string is UTF-8 encoded. The only other code points that have some sort of sameness across all 4 character sets are the pair 0xDC and 0xFC. Together these represent upper- and lowercase LATIN LETTER U WITH DIAERESIS, but which is upper and which is lower may be reversed: 0xDC is the capital in Latin1 and 0xFC is the small letter, while 0xFC is the capital in EBCDIC and 0xDC is the small one. This factoid may be exploited in writing case insensitive tests that are the same across all 4 character sets.

- Assuming the character set is just ASCII

  ASCII is a 7 bit encoding, but bytes have 8 bits in them. The 128 extra characters have different meanings depending on the locale. Absent a locale, currently these extra characters are generally considered to be unassigned, and this has presented some problems. This has being changed starting in 5.12 so that these characters can be considered to be Latin-1 (ISO-8859-1).

- Mixing #define and #ifdef

```
#define BURGLE(x) ... \
#ifdef BURGLE_OLD_STYLE        /* BAD */
... do it the old way ... \
#else
```

```
... do it the new way ... \
#endif
```

You cannot portably "stack" cpp directives. For example in the above you need two separate BURGLE() #defines, one for each #ifdef branch.

- Adding non-comment stuff after #endif or #else

```
#ifdef SNOSH
...
#else !SNOSH    /* BAD */
...
#endif SNOSH    /* BAD */
```

The #endif and #else cannot portably have anything non-comment after them. If you want to document what is going (which is a good idea especially if the branches are long), use (C) comments:

```
#ifdef SNOSH
...
#else /* !SNOSH */
...
#endif /* SNOSH */
```

The gcc option `-Wendif-labels` warns about the bad variant (by default on starting from Perl 5.9.4).

- Having a comma after the last element of an enum list

```
enum color {
  CERULEAN,
  CHARTREUSE,
  CINNABAR,     /* BAD */
};
```

is not portable. Leave out the last comma.

Also note that whether enums are implicitly morphable to ints varies between compilers, you might need to (int).

- Using //-comments

```
// This function bamfoodles the zorklator.   /* BAD */
```

That is C99 or C++. Perl is C89. Using the //-comments is silently allowed by many C compilers but cranking up the ANSI C89 strictness (which we like to do) causes the compilation to fail.

- Mixing declarations and code

```
void zorklator()
{
  int n = 3;
  set_zorkmids(n);    /* BAD */
  int q = 4;
```

That is C99 or C++. Some C compilers allow that, but you shouldn't.

The gcc option `-Wdeclaration-after-statements` scans for such problems (by default on starting from Perl 5.9.4).

- Introducing variables inside for()

```
for(int i = ...; ...; ...) {    /* BAD */
```

That is C99 or C++. While it would indeed be awfully nice to have that also in C89, to limit the scope of the loop variable, alas, we cannot.

- Mixing signed char pointers with unsigned char pointers

```
int foo(char *s) { ... }
...
unsigned char *t = ...; /* Or U8* t = ... */
foo(t);   /* BAD */
```

While this is legal practice, it is certainly dubious, and downright fatal in at least one platform: for example VMS cc considers this a fatal error. One cause for people often making this mistake is that a "naked char" and therefore dereferencing a "naked char pointer" have an undefined signedness: it depends on the compiler and the flags of the compiler and the underlying platform whether the result is signed or unsigned. For this very same reason using a 'char' as an array index is bad.

- Macros that have string constants and their arguments as substrings of the string constants

```
#define FOO(n) printf("number = %d\n", n)    /* BAD */
FOO(10);
```

Pre-ANSI semantics for that was equivalent to

```
printf("10umber = %d\10");
```

which is probably not what you were expecting. Unfortunately at least one reasonably common and modern C compiler does "real backward compatibility" here, in AIX that is what still happens even though the rest of the AIX compiler is very happily C89.

- Using printf formats for non-basic C types

```
IV i = ...;
printf("i = %d\n", i);    /* BAD */
```

While this might by accident work in some platform (where IV happens to be an `int`), in general it cannot. IV might be something larger. Even worse the situation is with more specific types (defined by Perl's configuration step in *config.h*):

```
Uid_t who = ...;
printf("who = %d\n", who);    /* BAD */
```

The problem here is that Uid_t might be not only not `int`-wide but it might also be unsigned, in which case large uids would be printed as negative values.

There is no simple solution to this because of printf()'s limited intelligence, but for many types the right format is available as with either 'f' or '_f' suffix, for example:

```
IVdf /* IV in decimal */
UVxf /* UV is hexadecimal */

printf("i = %"IVdf"\n", i); /* The IVdf is a string constant. */

Uid_t_f /* Uid_t in decimal */

printf("who = %"Uid_t_f"\n", who);
```

Or you can try casting to a "wide enough" type:

```
printf("i = %"IVdf"\n", (IV)something_very_small_and_signed);
```

See *"Formatted Printing of Size_t and SSize_t" in perlguts* for how to print those.

Also remember that the %p format really does require a void pointer:

```
U8* p = ...;
printf("p = %p\n", (void*)p);
```

The gcc option `-Wformat` scans for such problems.

- Blindly using variadic macros

  gcc has had them for a while with its own syntax, and C99 brought them with a standardized syntax. Don't use the former, and use the latter only if the HAS_C99_VARIADIC_MACROS is defined.

- Blindly passing va_list

  Not all platforms support passing va_list to further varargs (stdarg) functions. The right thing to do is to copy the va_list using the Perl_va_copy() if the NEED_VA_COPY is defined.

- Using gcc statement expressions

```
val = ({...;...;...});    /* BAD */
```

  While a nice extension, it's not portable. The Perl code does admittedly use them if available to gain some extra speed (essentially as a funky form of inlining), but you shouldn't.

- Binding together several statements in a macro

  Use the macros STMT_START and STMT_END.

```
STMT_START {
   ...
} STMT_END
```

- Testing for operating systems or versions when should be testing for features

```
#ifdef __FOONIX__     /* BAD */
foo = quux();
#endif
```

  Unless you know with 100% certainty that quux() is only ever available for the "Foonix" operating system **and** that is available **and** correctly working for **all** past, present, **and** future versions of "Foonix", the above is very wrong. This is more correct (though still not perfect, because the below is a compile-time check):

```
#ifdef HAS_QUUX
foo = quux();
#endif
```

  How does the HAS_QUUX become defined where it needs to be? Well, if Foonix happens to be Unixy enough to be able to run the Configure script, and Configure has been taught about detecting and testing quux(), the HAS_QUUX will be correctly defined. In other platforms, the corresponding configuration step will hopefully do the same.

  In a pinch, if you cannot wait for Configure to be educated, or if you have a good hunch of where quux() might be available, you can temporarily try the following:

```
#if (defined(__FOONIX__) || defined(__BARNIX__))
# define HAS_QUUX
#endif

...
```

```
#ifdef HAS_QUUX
foo = quux();
#endif
```

But in any case, try to keep the features and operating systems separate.

A good resource on the predefined macros for various operating systems, compilers, and so forth is *http://sourceforge.net/p/predef/wiki/Home/*

- Assuming the contents of static memory pointed to by the return values of Perl wrappers for C library functions doesn't change. Many C library functions return pointers to static storage that can be overwritten by subsequent calls to the same or related functions. Perl has light-weight wrappers for some of these functions, and which don't make copies of the static memory. A good example is the interface to the environment variables that are in effect for the program. Perl has `PerlEnv_getenv` to get values from the environment. But the return is a pointer to static memory in the C library. If you are using the value to immediately test for something, that's fine, but if you save the value and expect it to be unchanged by later processing, you would be wrong, but perhaps you wouldn't know it because different C library implementations behave differently, and the one on the platform you're testing on might work for your situation. But on some platforms, a subsequent call to `PerlEnv_getenv` or related function WILL overwrite the memory that your first call points to. This has led to some hard-to-debug problems. Do a *"savepv" in perlapi* to make a copy, thus avoiding these problems. You will have to free the copy when you're done to avoid memory leaks. If you don't have control over when it gets freed, you'll need to make the copy in a mortal scalar, like so:

```
if ((s = PerlEnv_getenv("foo") == NULL) {
   ... /* handle NULL case */
}
else {
    s = SvPVX(sv_2mortal(newSVpv(s, 0)));
}
```

  The above example works only if `"s"` is `NUL`-terminated; otherwise you have to pass its length to `newSVpv`.

Problematic System Interfaces

- malloc(0), realloc(0), calloc(0, 0) are non-portable. To be portable allocate at least one byte. (In general you should rarely need to work at this low level, but instead use the various malloc wrappers.)
- snprintf() - the return type is unportable. Use my_snprintf() instead.

Security problems

Last but not least, here are various tips for safer coding. See also *perlclib* for libc/stdio replacements one should use.

- Do not use gets()

  Or we will publicly ridicule you. Seriously.
- Do not use tmpfile()

  Use mkstemp() instead.
- Do not use strcpy() or strcat() or strncpy() or strncat()

  Use my_strlcpy() and my_strlcat() instead: they either use the native implementation, or Perl's own implementation (borrowed from the public domain implementation of INN).
- Do not use sprintf() or vsprintf()

  If you really want just plain byte strings, use my_snprintf() and my_vsnprintf() instead, which

will try to use snprintf() and vsnprintf() if those safer APIs are available. If you want something fancier than a plain byte string, use *Perl_form()* or SVs and *Perl_sv_catpvf()*.

Note that glibc `printf()`, `sprintf()`, etc. are buggy before glibc version 2.17. They won't allow a `%.s` format with a precision to create a string that isn't valid UTF-8 if the current underlying locale of the program is UTF-8. What happens is that the `%s` and its operand are simply skipped without any notice. *https://sourceware.org/bugzilla/show_bug.cgi?id=6530*.

- Do not use atoi()

  Use grok_atoUV() instead. atoi() has ill-defined behavior on overflows, and cannot be used for incremental parsing. It is also affected by locale, which is bad.

- Do not use strtol() or strtoul()

  Use grok_atoUV() instead. strtol() or strtoul() (or their IV/UV-friendly macro disguises, Strtol() and Strtoul(), or Atol() and Atoul() are affected by locale, which is bad.

DEBUGGING

You can compile a special debugging version of Perl, which allows you to use the `-D` option of Perl to tell more about what Perl is doing. But sometimes there is no alternative than to dive in with a debugger, either to see the stack trace of a core dump (very useful in a bug report), or trying to figure out what went wrong before the core dump happened, or how did we end up having wrong or unexpected results.

Poking at Perl

To really poke around with Perl, you'll probably want to build Perl for debugging, like this:

```
./Configure -d -DDEBUGGING
make
```

`-DDEBUGGING` turns on the C compiler's `-g` flag to have it produce debugging information which will allow us to step through a running program, and to see in which C function we are at (without the debugging information we might see only the numerical addresses of the functions, which is not very helpful). It will also turn on the `DEBUGGING` compilation symbol which enables all the internal debugging code in Perl. There are a whole bunch of things you can debug with this: *perlrun* lists them all, and the best way to find out about them is to play about with them. The most useful options are probably

```
l  Context (loop) stack processing
s  Stack snapshots (with v, displays all stacks)
t  Trace execution
o  Method and overloading resolution
c  String/numeric conversions
```

For example

```
$ perl -Dst -e '$a + 1'
....
(-e:1) gvsv(main::a)
    =>  UNDEF
(-e:1) const(IV(1))
    =>  UNDEF  IV(1)
(-e:1) add
    =>  NV(1)
```

Some of the functionality of the debugging code can be achieved with a non-debugging perl by using XS modules:

```
-Dr => use re 'debug'
-Dx => use O 'Debug'
```

Using a source-level debugger

If the debugging output of `-D` doesn't help you, it's time to step through perl's execution with a source-level debugger.

- We'll use `gdb` for our examples here; the principles will apply to any debugger (many vendors call their debugger `dbx`), but check the manual of the one you're using.

To fire up the debugger, type

```
gdb ./perl
```

Or if you have a core dump:

```
gdb ./perl core
```

You'll want to do that in your Perl source tree so the debugger can read the source code. You should see the copyright message, followed by the prompt.

```
(gdb)
```

`help` will get you into the documentation, but here are the most useful commands:

* run [args]

  Run the program with the given arguments.

* break function_name

* break source.c:xxx

  Tells the debugger that we'll want to pause execution when we reach either the named function (but see *"Internal Functions" in perlguts*!) or the given line in the named source file.

* step

  Steps through the program a line at a time.

* next

  Steps through the program a line at a time, without descending into functions.

* continue

  Run until the next breakpoint.

* finish

  Run until the end of the current function, then stop again.

* 'enter'

  Just pressing Enter will do the most recent operation again - it's a blessing when stepping through miles of source code.

* ptype

  Prints the C definition of the argument given.

```
(gdb) ptype PL_op
type = struct op {
    OP *op_next;
    OP *op_sibparent;
```

```
        OP *(*op_ppaddr)(void);
        PADOFFSET op_targ;
        unsigned int op_type : 9;
        unsigned int op_opt : 1;
        unsigned int op_slabbed : 1;
        unsigned int op_savefree : 1;
        unsigned int op_static : 1;
        unsigned int op_folded : 1;
        unsigned int op_spare : 2;
        U8 op_flags;
        U8 op_private;
    } *
```

* print

Execute the given C code and print its results. **WARNING**: Perl makes heavy use of macros, and *gdb* does not necessarily support macros (see later *gdb macro support*). You'll have to substitute them yourself, or to invoke cpp on the source code files (see *The .i Targets*) So, for instance, you can't say

```
print SvPV_nolen(sv)
```

but you have to say

```
print Perl_sv_2pv_nolen(sv)
```

You may find it helpful to have a "macro dictionary", which you can produce by saying `cpp -dM perl.c | sort`. Even then, *cpp* won't recursively apply those macros for you.

gdb macro support

Recent versions of *gdb* have fairly good macro support, but in order to use it you'll need to compile perl with macro definitions included in the debugging information. Using *gcc* version 3.1, this means configuring with `-Doptimize=-g3`. Other compilers might use a different switch (if they support debugging macros at all).

Dumping Perl Data Structures

One way to get around this macro hell is to use the dumping functions in *dump.c*; these work a little like an internal *Devel::Peek*, but they also cover OPs and other structures that you can't get at from Perl. Let's take an example. We'll use the `$a = $b + $c` we used before, but give it a bit of context: `$b = "6XXXX"; $c = 2.3;`. Where's a good place to stop and poke around?

What about `pp_add`, the function we examined earlier to implement the + operator:

```
(gdb) break Perl_pp_add
Breakpoint 1 at 0x46249f: file pp_hot.c, line 309.
```

Notice we use `Perl_pp_add` and not `pp_add` - see *"Internal Functions" in perlguts*. With the breakpoint in place, we can run our program:

```
(gdb) run -e '$b = "6XXXX"; $c = 2.3; $a = $b + $c'
```

Lots of junk will go past as gdb reads in the relevant source files and libraries, and then:

```
Breakpoint 1, Perl_pp_add () at pp_hot.c:309
309         dSP; dATARGET; tryAMAGICbin(add,opASSIGN);
(gdb) step
311           dPOPTOPnnrl_ul;
(gdb)
```

We looked at this bit of code before, and we said that `dPOPTOPnnrl_ul` arranges for two `NV`s to be placed into `left` and `right` - let's slightly expand it:

```
#define dPOPTOPnnrl_ul  NV right = POPn; \
                        SV *leftsv = TOPs; \
                        NV left = USE_LEFT(leftsv) ? SvNV(leftsv) : 0.0
```

`POPn` takes the SV from the top of the stack and obtains its NV either directly (if `SvNOK` is set) or by calling the `sv_2nv` function. `TOPs` takes the next SV from the top of the stack - yes, `POPn` uses `TOPs` - but doesn't remove it. We then use `SvNV` to get the NV from `leftsv` in the same way as before - yes, `POPn` uses `SvNV`.

Since we don't have an NV for `$b`, we'll have to use `sv_2nv` to convert it. If we step again, we'll find ourselves there:

```
(gdb) step
Perl_sv_2nv (sv=0xa0675d0) at sv.c:1669
1669        if (!sv)
(gdb)
```

We can now use `Perl_sv_dump` to investigate the SV:

```
(gdb) print Perl_sv_dump(sv)
SV = PV(0xa057cc0) at 0xa0675d0
REFCNT = 1
FLAGS = (POK,pPOK)
PV = 0xa06a510 "6XXXX"\0
CUR = 5
LEN = 6
$1 = void
```

We know we're going to get 6 from this, so let's finish the subroutine:

```
(gdb) finish
Run till exit from #0  Perl_sv_2nv (sv=0xa0675d0) at sv.c:1671
0x462669 in Perl_pp_add () at pp_hot.c:311
311           dPOPTOPnnrl_ul;
```

We can also dump out this op: the current op is always stored in `PL_op`, and we can dump it with `Perl_op_dump`. This'll give us similar output to *B::Debug*.

```
(gdb) print Perl_op_dump(PL_op)
{
13  TYPE = add  ===> 14
    TARG = 1
    FLAGS = (SCALAR,KIDS)
    {
        TYPE = null  ===> (12)
          (was rv2sv)
        FLAGS = (SCALAR,KIDS)
        {
11          TYPE = gvsv  ===> 12
            FLAGS = (SCALAR)
            GV = main::b
        }
    }
```

finish this later

Using gdb to look at specific parts of a program

With the example above, you knew to look for `Perl_pp_add`, but what if there were multiple calls to it all over the place, or you didn't know what the op was you were looking for?

One way to do this is to inject a rare call somewhere near what you're looking for. For example, you could add `study` before your method:

```
study;
```

And in gdb do:

```
(gdb) break Perl_pp_study
```

And then step until you hit what you're looking for. This works well in a loop if you want to only break at certain iterations:

```
for my $c (1..100) {
    study if $c == 50;
}
```

Using gdb to look at what the parser/lexer are doing

If you want to see what perl is doing when parsing/lexing your code, you can use `BEGIN {}`:

```
print "Before\n";
BEGIN { study; }
print "After\n";
```

And in gdb:

```
(gdb) break Perl_pp_study
```

If you want to see what the parser/lexer is doing inside of `if` blocks and the like you need to be a little trickier:

```
if ($a && $b && do { BEGIN { study } 1 } && $c) { ... }
```

SOURCE CODE STATIC ANALYSIS

Various tools exist for analysing C source code **statically**, as opposed to **dynamically**, that is, without executing the code. It is possible to detect resource leaks, undefined behaviour, type mismatches, portability problems, code paths that would cause illegal memory accesses, and other similar problems by just parsing the C code and looking at the resulting graph, what does it tell about the execution and data flows. As a matter of fact, this is exactly how C compilers know to give warnings about dubious code.

lint

The good old C code quality inspector, `lint`, is available in several platforms, but please be aware that there are several different implementations of it by different vendors, which means that the flags are not identical across different platforms.

There is a `lint` target in Makefile, but you may have to diddle with the flags (see above).

Coverity

Coverity (*http://www.coverity.com/*) is a product similar to lint and as a testbed for their product they periodically check several open source projects, and they give out accounts to open source

developers to the defect databases.

There is Coverity setup for the perl5 project: *https://scan.coverity.com/projects/perl5*

HP-UX cadvise (Code Advisor)

HP has a C/C++ static analyzer product for HP-UX caller Code Advisor. (Link not given here because the URL is horribly long and seems horribly unstable; use the search engine of your choice to find it.) The use of the `cadvise_cc` recipe with `Configure ... -Dcc=./cadvise_cc` (see cadvise "User Guide") is recommended; as is the use of `+wall`.

cpd (cut-and-paste detector)

The cpd tool detects cut-and-paste coding. If one instance of the cut-and-pasted code changes, all the other spots should probably be changed, too. Therefore such code should probably be turned into a subroutine or a macro.

cpd (*http://pmd.sourceforge.net/cpd.html*) is part of the pmd project (*http://pmd.sourceforge.net/*). pmd was originally written for static analysis of Java code, but later the cpd part of it was extended to parse also C and C++.

Download the pmd-bin-X.Y.zip () from the SourceForge site, extract the pmd-X.Y.jar from it, and then run that on source code thusly:

```
java -cp pmd-X.Y.jar net.sourceforge.pmd.cpd.CPD \
 --minimum-tokens 100 --files /some/where/src --language c > cpd.txt
```

You may run into memory limits, in which case you should use the -Xmx option:

```
java -Xmx512M ...
```

gcc warnings

Though much can be written about the inconsistency and coverage problems of gcc warnings (like `-Wall` not meaning "all the warnings", or some common portability problems not being covered by `-Wall`, or `-ansi` and `-pedantic` both being a poorly defined collection of warnings, and so forth), gcc is still a useful tool in keeping our coding nose clean.

The `-Wall` is by default on.

The `-ansi` (and its sidekick, `-pedantic`) would be nice to be on always, but unfortunately they are not safe on all platforms, they can for example cause fatal conflicts with the system headers (Solaris being a prime example). If Configure `-Dgccansipedantic` is used, the `cflags` frontend selects `-ansi -pedantic` for the platforms where they are known to be safe.

Starting from Perl 5.9.4 the following extra flags are added:

- `-Wendif-labels`
- `-Wextra`
- `-Wdeclaration-after-statement`

The following flags would be nice to have but they would first need their own Augean stablemaster:

- `-Wpointer-arith`
- `-Wshadow`
- `-Wstrict-prototypes`

The `-Wtraditional` is another example of the annoying tendency of gcc to bundle a lot of warnings under one switch (it would be impossible to deploy in practice because it would complain a lot) but it

does contain some warnings that would be beneficial to have available on their own, such as the warning about string constants inside macros containing the macro arguments: this behaved differently pre-ANSI than it does in ANSI, and some C compilers are still in transition, AIX being an example.

Warnings of other C compilers

Other C compilers (yes, there **are** other C compilers than gcc) often have their "strict ANSI" or "strict ANSI with some portability extensions" modes on, like for example the Sun Workshop has its `-Xa` mode on (though implicitly), or the DEC (these days, HP...) has its `-std1` mode on.

MEMORY DEBUGGERS

NOTE 1: Running under older memory debuggers such as Purify, valgrind or Third Degree greatly slows down the execution: seconds become minutes, minutes become hours. For example as of Perl 5.8.1, the ext/Encode/t/Unicode.t takes extraordinarily long to complete under e.g. Purify, Third Degree, and valgrind. Under valgrind it takes more than six hours, even on a snappy computer. The said test must be doing something that is quite unfriendly for memory debuggers. If you don't feel like waiting, that you can simply kill away the perl process. Roughly valgrind slows down execution by factor 10, AddressSanitizer by factor 2.

NOTE 2: To minimize the number of memory leak false alarms (see *PERL_DESTRUCT_LEVEL* for more information), you have to set the environment variable PERL_DESTRUCT_LEVEL to 2. For example, like this:

```
env PERL_DESTRUCT_LEVEL=2 valgrind ./perl -Ilib ...
```

NOTE 3: There are known memory leaks when there are compile-time errors within eval or require, seeing `S_doeval` in the call stack is a good sign of these. Fixing these leaks is non-trivial, unfortunately, but they must be fixed eventually.

NOTE 4: *DynaLoader* will not clean up after itself completely unless Perl is built with the Configure option `-Accflags=-DDL_UNLOAD_ALL_AT_EXIT`.

valgrind

The valgrind tool can be used to find out both memory leaks and illegal heap memory accesses. As of version 3.3.0, Valgrind only supports Linux on x86, x86-64 and PowerPC and Darwin (OS X) on x86 and x86-64. The special "test.valgrind" target can be used to run the tests under valgrind. Found errors and memory leaks are logged in files named *testfile.valgrind* and by default output is displayed inline.

Example usage:

```
make test.valgrind
```

Since valgrind adds significant overhead, tests will take much longer to run. The valgrind tests support being run in parallel to help with this:

```
TEST_JOBS=9 make test.valgrind
```

Note that the above two invocations will be very verbose as reachable memory and leak-checking is enabled by default. If you want to just see pure errors, try:

```
VG_OPTS='-q --leak-check=no --show-reachable=no' TEST_JOBS=9 \
    make test.valgrind
```

Valgrind also provides a cachegrind tool, invoked on perl as:

```
VG_OPTS=--tool=cachegrind make test.valgrind
```

As system libraries (most notably glibc) are also triggering errors, valgrind allows to suppress such errors using suppression files. The default suppression file that comes with valgrind already catches a lot of them. Some additional suppressions are defined in *t/perl.supp*.

To get valgrind and for more information see

```
http://valgrind.org/
```

AddressSanitizer

AddressSanitizer is a clang and gcc extension, included in clang since v3.1 and gcc since v4.8. It checks illegal heap pointers, global pointers, stack pointers and use after free errors, and is fast enough that you can easily compile your debugging or optimized perl with it. It does not check memory leaks though. AddressSanitizer is available for Linux, Mac OS X and soon on Windows.

To build perl with AddressSanitizer, your Configure invocation should look like:

```
sh Configure -des -Dcc=clang \
   -Accflags=-faddress-sanitizer -Aldflags=-faddress-sanitizer \
   -Alddlflags=-shared\ -faddress-sanitizer
```

where these arguments mean:

* -Dcc=clang

  This should be replaced by the full path to your clang executable if it is not in your path.

* -Accflags=-faddress-sanitizer

  Compile perl and extensions sources with AddressSanitizer.

* -Aldflags=-faddress-sanitizer

  Link the perl executable with AddressSanitizer.

* -Alddlflags=-shared\ -faddress-sanitizer

  Link dynamic extensions with AddressSanitizer. You must manually specify `-shared` because using `-Alddlflags=-shared` will prevent Configure from setting a default value for `lddlflags`, which usually contains `-shared` (at least on Linux).

See also *http://code.google.com/p/address-sanitizer/wiki/AddressSanitizer*.

PROFILING

Depending on your platform there are various ways of profiling Perl.

There are two commonly used techniques of profiling executables: *statistical time-sampling* and *basic-block counting*.

The first method takes periodically samples of the CPU program counter, and since the program counter can be correlated with the code generated for functions, we get a statistical view of in which functions the program is spending its time. The caveats are that very small/fast functions have lower probability of showing up in the profile, and that periodically interrupting the program (this is usually done rather frequently, in the scale of milliseconds) imposes an additional overhead that may skew the results. The first problem can be alleviated by running the code for longer (in general this is a good idea for profiling), the second problem is usually kept in guard by the profiling tools themselves.

The second method divides up the generated code into *basic blocks*. Basic blocks are sections of code that are entered only in the beginning and exited only at the end. For example, a conditional jump starts a basic block. Basic block profiling usually works by *instrumenting* the code by adding *enter basic block #nnnn* book-keeping code to the generated code. During the execution of the code the basic block counters are then updated appropriately. The caveat is that the added extra code can skew the results: again, the profiling tools usually try to factor their own effects out of the results.

Gprof Profiling

gprof is a profiling tool available in many Unix platforms which uses *statistical time-sampling*. You can build a profiled version of *perl* by compiling using gcc with the flag `-pg`. Either edit *config.sh* or re-run *Configure*. Running the profiled version of Perl will create an output file called *gmon.out* which contains the profiling data collected during the execution.

quick hint:

```
$ sh Configure -des -Dusedevel -Accflags='-pg' \
    -Aldflags='-pg' -Alddlflags='-pg -shared' \
    && make perl
$ ./perl ... # creates gmon.out in current directory
$ gprof ./perl > out
$ less out
```

(you probably need to add `-shared` to the <-Alddlflags> line until RT #118199 is resolved)

The *gprof* tool can then display the collected data in various ways. Usually *gprof* understands the following options:

* -a

  Suppress statically defined functions from the profile.

* -b

  Suppress the verbose descriptions in the profile.

* -e routine

  Exclude the given routine and its descendants from the profile.

* -f routine

  Display only the given routine and its descendants in the profile.

* -s

  Generate a summary file called *gmon.sum* which then may be given to subsequent gprof runs to accumulate data over several runs.

* -z

  Display routines that have zero usage.

For more detailed explanation of the available commands and output formats, see your own local documentation of *gprof*.

GCC gcov Profiling

basic block profiling is officially available in gcc 3.0 and later. You can build a profiled version of *perl* by compiling using gcc with the flags `-fprofile-arcs -ftest-coverage`. Either edit *config.sh* or re-run *Configure*.

quick hint:

```
$ sh Configure -des -Dusedevel -Doptimize='-g' \
    -Accflags='-fprofile-arcs -ftest-coverage' \
    -Aldflags='-fprofile-arcs -ftest-coverage' \
    -Alddlflags='-fprofile-arcs -ftest-coverage -shared' \
    && make perl
$ rm -f regexec.c.gcov regexec.gcda
$ ./perl ...
$ gcov regexec.c
```

```
$ less regexec.c.gcov
```

(you probably need to add `-shared` to the <-Alddlflags> line until RT #118199 is resolved)

Running the profiled version of Perl will cause profile output to be generated. For each source file an accompanying *.gcda* file will be created.

To display the results you use the *gcov* utility (which should be installed if you have gcc 3.0 or newer installed). *gcov* is run on source code files, like this

```
gcov sv.c
```

which will cause *sv.c.gcov* to be created. The *.gcov* files contain the source code annotated with relative frequencies of execution indicated by "#" markers. If you want to generate *.gcov* files for all profiled object files, you can run something like this:

```
for file in `find . -name \*.gcno`
do sh -c "cd `dirname $file` && gcov `basename $file .gcno`"
done
```

Useful options of *gcov* include `-b` which will summarise the basic block, branch, and function call coverage, and `-c` which instead of relative frequencies will use the actual counts. For more information on the use of *gcov* and basic block profiling with gcc, see the latest GNU CC manual. As of gcc 4.8, this is at *http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro*

MISCELLANEOUS TRICKS

PERL_DESTRUCT_LEVEL

If you want to run any of the tests yourself manually using e.g. valgrind, please note that by default perl **does not** explicitly cleanup all the memory it has allocated (such as global memory arenas) but instead lets the exit() of the whole program "take care" of such allocations, also known as "global destruction of objects".

There is a way to tell perl to do complete cleanup: set the environment variable PERL_DESTRUCT_LEVEL to a non-zero value. The t/TEST wrapper does set this to 2, and this is what you need to do too, if you don't want to see the "global leaks": For example, for running under valgrind

```
env PERL_DESTRUCT_LEVEL=2 valgrind ./perl -Ilib t/foo/bar.t
```

(Note: the mod_perl apache module uses also this environment variable for its own purposes and extended its semantics. Refer to the mod_perl documentation for more information. Also, spawned threads do the equivalent of setting this variable to the value 1.)

If, at the end of a run you get the message *N scalars leaked*, you can recompile with `-DDEBUG_LEAKING_SCALARS`, (`Configure -Accflags=-DDEBUG_LEAKING_SCALARS`), which will cause the addresses of all those leaked SVs to be dumped along with details as to where each SV was originally allocated. This information is also displayed by Devel::Peek. Note that the extra details recorded with each SV increases memory usage, so it shouldn't be used in production environments. It also converts `new_SV()` from a macro into a real function, so you can use your favourite debugger to discover where those pesky SVs were allocated.

If you see that you're leaking memory at runtime, but neither valgrind nor `-DDEBUG_LEAKING_SCALARS` will find anything, you're probably leaking SVs that are still reachable and will be properly cleaned up during destruction of the interpreter. In such cases, using the `-Dm` switch can point you to the source of the leak. If the executable was built with `-DDEBUG_LEAKING_SCALARS`, `-Dm` will output SV allocations in addition to memory allocations. Each SV allocation has a distinct serial number that will be written on creation and destruction of the

SV. So if you're executing the leaking code in a loop, you need to look for SVs that are created, but never destroyed between each cycle. If such an SV is found, set a conditional breakpoint within `new_SV()` and make it break only when `PL_sv_serial` is equal to the serial number of the leaking SV. Then you will catch the interpreter in exactly the state where the leaking SV is allocated, which is sufficient in many cases to find the source of the leak.

As `-Dm` is using the PerlIO layer for output, it will by itself allocate quite a bunch of SVs, which are hidden to avoid recursion. You can bypass the PerlIO layer if you use the SV logging provided by `-DPERL_MEM_LOG` instead.

PERL_MEM_LOG

If compiled with `-DPERL_MEM_LOG` (`-Accflags=-DPERL_MEM_LOG`), both memory and SV allocations go through logging functions, which is handy for breakpoint setting.

Unless `-DPERL_MEM_LOG_NOIMPL` (`-Accflags=-DPERL_MEM_LOG_NOIMPL`) is also compiled, the logging functions read $ENV{PERL_MEM_LOG} to determine whether to log the event, and if so how:

```
$ENV{PERL_MEM_LOG} =~ /m/           Log all memory ops
$ENV{PERL_MEM_LOG} =~ /s/           Log all SV ops
$ENV{PERL_MEM_LOG} =~ /t/           include timestamp in Log
$ENV{PERL_MEM_LOG} =~ /^(\d+)/      write to FD given (default is 2)
```

Memory logging is somewhat similar to `-Dm` but is independent of `-DDEBUGGING`, and at a higher level; all uses of Newx(), Renew(), and Safefree() are logged with the caller's source code file and line number (and C function name, if supported by the C compiler). In contrast, `-Dm` is directly at the point of `malloc()`. SV logging is similar.

Since the logging doesn't use PerlIO, all SV allocations are logged and no extra SV allocations are introduced by enabling the logging. If compiled with `-DDEBUG_LEAKING_SCALARS`, the serial number for each SV allocation is also logged.

DDD over gdb

Those debugging perl with the DDD frontend over gdb may find the following useful:

You can extend the data conversion shortcuts menu, so for example you can display an SV's IV value with one click, without doing any typing. To do that simply edit ~/.ddd/init file and add after:

```
! Display shortcuts.
Ddd*gdbDisplayShortcuts: \
/t ()   // Convert to Bin\n\
/d ()   // Convert to Dec\n\
/x ()   // Convert to Hex\n\
/o ()   // Convert to Oct(\n\
```

the following two lines:

```
((XPV*) (())->sv_any )->xpv_pv  // 2pvx\n\
((XPVIV*) (())->sv_any )->xiv_iv // 2ivx
```

so now you can do ivx and pvx lookups or you can plug there the sv_peek "conversion":

```
Perl_sv_peek(my_perl, (SV*)()) // sv_peek
```

(The my_perl is for threaded builds.) Just remember that every line, but the last one, should end with \n\

Alternatively edit the init file interactively via: 3rd mouse button -> New Display -> Edit Menu

Note: you can define up to 20 conversion shortcuts in the gdb section.

C backtrace

On some platforms Perl supports retrieving the C level backtrace (similar to what symbolic debuggers like gdb do).

The backtrace returns the stack trace of the C call frames, with the symbol names (function names), the object names (like "perl"), and if it can, also the source code locations (file:line).

The supported platforms are Linux, and OS X (some *BSD might work at least partly, but they have not yet been tested).

This feature hasn't been tested with multiple threads, but it will only show the backtrace of the thread doing the backtracing.

The feature needs to be enabled with `Configure -Dusecbacktrace`.

The `-Dusecbacktrace` also enables keeping the debug information when compiling/linking (often: `-g`). Many compilers/linkers do support having both optimization and keeping the debug information. The debug information is needed for the symbol names and the source locations.

Static functions might not be visible for the backtrace.

Source code locations, even if available, can often be missing or misleading if the compiler has e.g. inlined code. Optimizer can make matching the source code and the object code quite challenging.

Linux

> You **must** have the BFD (-lbfd) library installed, otherwise `perl` will fail to link. The BFD is usually distributed as part of the GNU binutils.
>
> Summary: `Configure ... -Dusecbacktrace` and you need `-lbfd`.

OS X

> The source code locations are supported **only** if you have the Developer Tools installed. (BFD is **not** needed.)
>
> Summary: `Configure ... -Dusecbacktrace` and installing the Developer Tools would be good.

Optionally, for trying out the feature, you may want to enable automatic dumping of the backtrace just before a warning or croak (die) message is emitted, by adding `-Accflags=-DUSE_C_BACKTRACE_ON_ERROR` for Configure.

Unless the above additional feature is enabled, nothing about the backtrace functionality is visible, except for the Perl/XS level.

Furthermore, even if you have enabled this feature to be compiled, you need to enable it in runtime with an environment variable: `PERL_C_BACKTRACE_ON_ERROR=10`. It must be an integer higher than zero, telling the desired frame count.

Retrieving the backtrace from Perl level (using for example an XS extension) would be much less exciting than one would hope: normally you would see `runops`, `entersub`, and not much else. This API is intended to be called **from within** the Perl implementation, not from Perl level execution.

The C API for the backtrace is as follows:

get_c_backtrace

free_c_backtrace

get_c_backtrace_dump

dump_c_backtrace

Poison

If you see in a debugger a memory area mysteriously full of 0xABABABAB or 0xEFEFEFEF, you may be seeing the effect of the Poison() macros, see *perlclib*.

Read-only optrees

Under ithreads the optree is read only. If you want to enforce this, to check for write accesses from buggy code, compile with `-Accflags=-DPERL_DEBUG_READONLY_OPS` to enable code that allocates op memory via `mmap`, and sets it read-only when it is attached to a subroutine. Any write access to an op results in a `SIGBUS` and abort.

This code is intended for development only, and may not be portable even to all Unix variants. Also, it is an 80% solution, in that it isn't able to make all ops read only. Specifically it does not apply to op slabs belonging to `BEGIN` blocks.

However, as an 80% solution it is still effective, as it has caught bugs in the past.

When is a bool not a bool?

On pre-C99 compilers, `bool` is defined as equivalent to `char`. Consequently assignment of any larger type to a `bool` is unsafe and may be truncated. The `cBOOL` macro exists to cast it correctly; you may also find that using it is shorter and clearer than writing out the equivalent conditional expression longhand.

On those platforms and compilers where `bool` really is a boolean (C++, C99), it is easy to forget the cast. You can force `bool` to be a `char` by compiling with `-Accflags=-DPERL_BOOL_AS_CHAR`. You may also wish to run `Configure` with something like

```
-Accflags='-Wconversion -Wno-sign-conversion -Wno-shorten-64-to-32'
```

or your compiler's equivalent to make it easier to spot any unsafe truncations that show up.

The `TRUE` and `FALSE` macros are available for situations where using them would clarify intent. (But they always just mean the same as the integers 1 and 0 regardless, so using them isn't compulsory.)

The .i Targets

You can expand the macros in a *foo.c* file by saying

```
make foo.i
```

which will expand the macros using cpp. Don't be scared by the results.

AUTHOR

This document was originally written by Nathan Torkington, and is maintained by the perl5-porters mailing list.