;login:

THE MAGAZINE OF USENIX & SAGE

April 2003 • volume 28 • number 2

inside:

USENIX & SAGE

The Advanced Computing Systems Association & The System Administrators Guild

practical Perl
fixing broken modules

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long- time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

Introduction

I recently worked on a project where I needed to make some bug fixes to some locally written Perl modules. To make my changes, I fixed and tested a local copy of these modules, modifying Perl's search path to find my copy of them instead of the buggy versions. Modifying Perl's search path is an excellent way to test experimental code without the hassle of installing each fix, or impacting other users on the same system.

Dynamic languages like Perl, Python, and Ruby all have one very important feature in common: Programs written in these languages are distributed in source form. This is a great boon to software developers who need to debug a system after it is installed. A Perl programmer can examine a large Perl application and see exactly how it works and even make changes, if necessary. Debugging an installed application written in Perl is trivial. Debugging a program written in C, C++, or Java is more difficult, especially if the original source code is lost, misplaced, or otherwise unavailable.

Access to a program's source code eases repair. Once I find a bug in a Perl program that needs fixing, I can easily copy the program, make a change, and test the updated program. I can then replace the original program with my fixed version if I have write access to the appropriate directories. If I do not have sufficient permissions to upgrade the program, I can maintain the updated program in a local portion of my path while I wait for installation issues to be sorted out. In the worst case, I can update my PATH environment variable to find my updated program before the existing version.

Fixing a broken library module is a similar process, but slightly more complicated. If I do not have write permission to the library module directories, then I cannot install the update. Even if I could, there are good reasons not to update a module that could be used by many programs and users on a system. It is entirely possible that my blindly overwriting an existing module with my updated version will fix my program, but it will also break a great many other programs in use.

If I want to install an updated module locally, I need to update Perl's module search path to find my update before the preexisting version. Because this is Perl, there's more than one way to do it. Which technique I will use will depend on the situation.

Perl's Module Search Path

Perl processes programs in two phases: compile time and runtime. During compile time, Perl ensures that your program is syntactically correct and performs other operations, like loading modules. Once this process is complete, the runtime phase begins and Perl starts to execute your program statements.

During compile time, Perl includes external modules by looking through a list of directories named by @INC until it finds an appropriate file to load. To include modules from a specific directory, update @INC during compile time and before use statements are processed. Modifying @INC at runtime will have no impact since the program has been compiled and all use statements have already been processed.

By default, Perl looks in one of two general file-system areas when a module is to be loaded. Core modules (the ones that are bundled with Perl) are stored in $PREFIX/lib/perl5, where $PREFIX is the base of the Perl installation, like /usr, /usr/local, or /opt. Additional modules, like those installed from CPAN, are stored in $PREFIX/lib/site_perl. These directories may also include subdirectories containing Perl's version number (e.g., 5.005_03, 5.6.1, or 5.8.0) and subdirectories containing the current platform architecture (i386-freebsd, darwin, etc.).

If a module is not found in any of these locations, Perl will look in the current directory. If Perl still cannot find a module, it will terminate processing the program and report a fatal error.

Whenever Perl is loading a module, it looks for the first matching module encountered in the list that is the search path. If I want to override a previously installed module, I must place it in a directory that will appear before the normal module search directories.

Updating the Module Search Path

There are many ways to update the module search path. One way is to use the PERL5LIB environment variable. Using environment variables to change Perl's behavior is generally discouraged, because "opaque" settings are hard to notice, especially by a casual maintainer; besides, they can vary on a per-user or per-terminal basis. Sometimes, setting PERL5LIB is the best way to update the module search path, like communicating with Perl subprocesses.

PERL5LIB adds new library directories to the front of the module search path. It can contain a series of colon-delimited directories:

```
[ziggy@duvel ~]$ mkdir newlib1 newlib2
[ziggy@duvel ~]$ export PERL5LIB=newlib1:newlib2
[ziggy@duvel ~]$ perl -le 'print join("\n", @INC)' newlib1
newlib2
/opt/lib/5.8.0/darwin
/opt/lib/5.8.0
/opt/lib/site_perl/5.8.0/darwin
/opt/lib/site_perl/5.8.0
/opt/lib/site_perl
```

Whenever a directory in PERL5LIB contains a subdirectory that matches the current platform architecture, that platform-specific directory will also be added to the search path. This is also the location where compiled C extensions will be installed.

```
[ziggy@duvel ~]$ mkdir newlib2/darwin
[ziggy@duvel ~]$ export PERL5LIB=newlib1:newlib2
[ziggy@duvel ~]$ perl -le 'print join("\n", @INC)' newlib1
newlib2/darwin
newlib2
/opt/lib/5.8.0/darwin
/opt/lib/5.8.0
/opt/lib/site_perl/5.8.0/darwin
/opt/lib/site_perl/5.8.0
/opt/lib/site_perl
```

Another way to extend the module search path is to use Perl's -I command line switch. Adding multiple -I switches when invoking Perl will add multiple directories (and version-specific subdirectories) to @INC. Note that adding -I switches on the command line will prepend directories to @INC, while using -I on the shebang (#!) line will append directories to the end of @INC:

```
[ziggy@duvel ~]$ cat > test.pl
#!/usr/bin/perl -lw -Inewlib2
print join("\n", @INC);
^D
[ziggy@duvel ~]$ perl -Inewlib1 test.pl
newlib1
/opt/lib/5.8.0/darwin
/opt/lib/5.8.0
/opt/lib/site_perl/5.8.0/darwin
/opt/lib/site_perl/5.8.0
/opt/lib/site_perl
.
newlib2/darwin
newlib2
[ziggy@duvel ~]$
```

With this behavior, using -I on the shebang line is sufficient for adding a module directory to @INC to find modules that are not stored in the core or site module directories. If you want to include a directory in @INC to supersede the modules installed elsewhere on the system, you must specify -I on the command line when invoking Perl.

A third way to add directories to @INC is to modify @INC directly at compile time. One way to do this is to modify @INC in a BEGIN block, so that it will be modified before modules are loaded:

```
#!/usr/bin/perl -lw
BEGIN {unshift(@INC, "newlib1", "newlib2"); } print
join("\n", @INC);
```

No output is produced.

Using BEGIN blocks and directly tweaking @INC works, but it is ugly and obscure. A better way to perform the same task is to use a use lib; declaration instead. This declaration is processed at compile time, before modules are loaded. The use lib; declaration does exactly what it says — prepends another library path to the list of module search paths.

```
[ziggy@duvel ~]$ perl -lw
use lib qw(newlib1 newlib2);
print join("\n", @INC);
^D
newlib1
newlib2
/opt/lib/5.8.0/darwin
/opt/lib/5.8.0
/opt/lib/site_perl/5.8.0/darwin
/opt/lib/site_perl/5.8.0
/opt/lib/site_perl
.
```

Note that use lib; declarations can insert directories at the front of @INC, but each directory must be explicitly added. Only PERL5LIB and perl -I can automatically add a platform-specific subdirectory to @INC to find compiled modules.

Using Local Module Directories

Updating the module search path has a great many uses. The most common use is to load modules from an application-specific library directory. Installing modules in a local library makes it easy to quickly modify an application's modules when it is in development. This eliminates the need to go through the module-install process for each update.

If you cannot or do not want to add modules to the standard module library, you can install modules elsewhere and find them with a use lib; declaration. This is a great way to test modules before installing them, or install modules in a central location on a system where you cannot install a module in the normal site-wide location.

Fixing a Broken Module

One of the lesser-known ways to use a local library directory is to fix a broken module. Because I can create a local module library directory and configure Perl to look there for modules, I can make a copy of a broken module and fix it. I can update programs to look for this fixed module, or invoke Perl using PERL5LIB or perl -I to find my fixed module.

Consider this little module, which misbehaves and emits an obnoxious number of status messages:

```
package Sample;
use strict;
$|++;   ## Turn on auto flushing
sub new {
      print STDOUT "Creating a new object\n";
      my $object = {};
      bless $object, __PACKAGE__;
      $object->load_config();
      return $object;
}
sub load_config {
      print STDOUT "Loading configuration file\n";
                      my %config;
      $/ = "\n";   ## Read in a series of lines
      open(F, "/etc/Sample.conf");
      while (<F>) {
          chomp;
          s/#.*$//;
          s/s+/ /;
          next unless m/^(.*?)=(.*)$/;
          $config{$1} = $2;
      }
      return %config;
}
...
1;
```

This module contains a few errors I want to fix. I start by copying Sample.pm from its location in /opt/lib/perl5/5.8.0 to my local module directory, ~/fixed-modules/. I then make my changes to ~/fixed-modules/Sample.pm.

In some cases, I could fix this module by writing a new module and inheriting from Sample.pm. However, the problems that I want to fix are endemic and cannot be fixed effectively without rewriting the entire module.

Another approach would be to create a fix to this module and rename it as FixedSample.pm. If I have several programs that use the Sample module, then I will have a lot of programs to update to use FixedSample instead. By making a fixed version of Sample available, I can continue to use existing programs with little or no modifications to those programs.

The first error I need to fix is the modification of the $| special variable at the beginning of the module. This global variable controls “auto flushing,” or automatically flushing data sent to STDOUT instead of buffering it. Modification to $| is usually a global change in program behavior. Methods in this module may want to have output to STDOUT flushed immediately, but other portions of this program may rely on STDOUT being buffered. Changing global behavior like this is bad style when writing a module.

A better way to flush output to STDOUT within a module is to modify the value of $| locally. This can be done by making a local copy of $| within a sub and modifying that copy:

```
sub new {
      ## Turn on autoflushing only within this sub
      local $| = 1;
      print STDOUT "Creating a new object\n";
      ...
}
```

There is a similar problem in load_config(). It modifies the $/, or input record separator. Within load_config(), this variable needs to be a newline character (the default value). Another portion of the program may need a different behavior, like reading in an entire file at once, or reading in one block at a time. These behaviors are defined by setting $/ to undef or the empty string. Calling load_config() will change this global behavior and may inadvertently impact other portions of the program.

Fixing this buggy behavior also requires making a local modification to the value of $/:

```
sub load_config {
      ## turn on autoflushing locally
      local $| = 1;
      print STDOUT "Loading configuration file\n";
                      my %config;
      ## Read in a series of lines within this sub
      local $/ = "\n";
      ...
}
```

Another problem I want to fix deals with opening the configuration file. Remember that file handles in Perl are global variables. The configuration file is opened using the file handle fh, which is a very common name for a file handle. If my program already has an open file handle called fh, then load_config() will close it and open up another file instead. I could fix this problem by choosing a better name for my file handle. A better solution would be to use a lexical file handle, or a file handle that exists only within this sub:

```
sub load_config {
      ...
      open (my $fh, "/etc/Sample.conf");
      while (<$fh>) {
          ...
      }
      ...
}
```

After I make these changes, my copy of the Sample module will be well behaved. But it still emits an obnoxious number of log messages. I don't need to see them, and I would like to eliminate them. Here is what my fixed version of this module looks like after I've removed the unnecessary print statements:

```
package Sample;
use strict;
sub new {
      my $object = {};
      bless $object, __PACKAGE__;
      $object->load_config();
      return $object;
}
sub load_config {
      my %config;
      local $/ = "\n";
      open(my $fh, "/etc/Sample.conf");
      while (<$fh>) {
          chomp;
          s/#.*$//;
          s/s+/ /;
          next unless m/^(.*?)=(.*)$/;
          $config{$1} = $2;
      }
      return %config;
}
...
1;
```

With an updated version of my module, all I need to do now is use it in my programs. I can do this in a number of ways. I can modify programs that I explicitly want to use this module by adding a use lib '$ENV{HOME}/fixed-modules';. If I do not want to modify my Perl programs, I can use perl -I~/fixed-modules when invoking programs that use this module, or set PERL5LIB to include ~/fixed-modules. Any of these techniques will allow me to override Sample.pm with my copy.

Annotating Unfamiliar Modules

Periodically, I need to fix a module I have never seen before. I might be able to isolate a problem using the Perl debugger, but any insights I gain will probably be lost by the time I finish a debugging session. I could write them down on paper or online somewhere, but there is no guarantee I'll have that paper or file available when I need it again. Usually, it will be sitting on my desk at home when I am at work, or sitting on my desk at work when I am at home.

A better solution would be to comment on the program's source code directly. In this situation, I probably do not want to update the installed module directly, but I can comment on a module if I create a local copy and annotate that copy. If my changes are useful, then I can easily create a patch to send back to the module's author.

Another modification I can make locally is to normalize a module's coding style. Many module suites are written by multiple programmers over a period of time. Sometimes the style of each author will vary, or the coding style is so different from my own that it makes the module difficult to read.

I'd rather fix a problem than complain about coding style. If I make a local copy of a module, I can process it using a code formatter like perltidy and apply a single, readable style to one or more modules. My goal here is to understand how a module works, not to start a flame war over style. By loading my reformatted version of the modules, I can also use them when debugging a program. Once I understand how the module works, I can patch the original version of the module.

Conclusion

Because Perl programs and Perl modules are distributed as source code, it is easy to take an existing program and modify it to quickly fix a bug. This kind of fast turnaround helps you solve a problem before the boss fires you.

Fixing modules takes a little more effort than fixing a broken program, but it can be done. The key to fixing a broken module rests with loading modules from a local directory, and updating Perl's module search path, @INC, to find the updated modules.