Table of Contents

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Chapter 4. Working with Modules

Hacks 28-42

Perhaps the greatest invention of Perl 5 is the idea of modules. They allow people to modify the language and reuse code far beyond what Larry and the Perl 5 porters ever envisioned. (Who could have predicted CPAN or `Acme::*`, for example?)

If you're doing any serious work with Perl, you'll spend a lot of time working with modules: installing them, upgrading them, loading them, working around weird and unhelpful features, and even distributing them. It makes a lot of sense to understand how Perl and modules interact and how to work with them effectively.

Here are several ideas that show off the varied ways that you can extend your programs. CPAN is only an arm's length away. Be ready.

Hack 28. Shorten Long Class Names

Type only what you need to type. You know what you mean.

Are you tired of using Perl classes with `Really::Long::Package::Names::You::Cant::Remember`? Use `aliased` and forget about them. This handy CPAN module creates short, easy-to-remember aliases for long class names.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The Hack

Given the hypothetical example just cited, use `aliased` to load the class and create an alias all at once:

```
use aliased 'Really::Long::Package::Names::You::Cant::Remember';

my $rem = Remember->new( );
```

When `aliased` loads a class, it automatically creates a constant subroutine, in the local name space named after the final part of the package name. This subroutine returns the full package name. Because it's a constant, it's actually very efficient; Perl will inline the package name, so that by the time your code has compiled, Perl sees it as if you had actually typed:

```
use aliased 'Really::Long::Package::Names::You::Cant::Remember';

my $rem = Really::Long::Package::Names::You::Cant::Remember->new( );
```

You gain simplicity and lose, well, nothing.

Resolve conflicts

Sometimes you might want to alias two classes that have the same final portion of their package names. In such cases, specify the alias that you want to use to disambiguate the two classes:

```
use aliased 'My::App::Contact';
use aliased 'My::App::Type::Contact' => 'ContactType';

my $contact_type = ContactType->new( );
my $contact      = Contact->new({ type => $contact_type });
```

Importing with aliased

Sometimes, even in object-oriented programming, you need to import symbols from a module. `aliased` allows you to do so while still creating an alias. The only wrinkle is that you *must* explicitly specify an alias. Why? Because then you pass in a list of import symbols, and if you didn't specify an alias name, the first symbol would be the alias! Here's how it works:

```
use aliased 'My::App::Contact' => 'Contact', qw( EMAIL PHONE );

my $contact = Contact->new({
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
    kind  => EMAIL,
    value => 'perlhacks@oreilly.com',
});
```

If you hadn't put that `'Contact'` there, then the alias would have been `EMAIL` and that wouldn't do what you meant .

Hack 29. Manage Module Paths

Keep your code where it makes sense to you, not just to Perl.

Perl's a flexible language and it tries to make few assumptions about your environment. Perhaps you're a system administrator with root access and a compiler and can install modules anywhere you want. Perhaps you only have shell access on a shared box and have to submit a change request to have something installed. Perhaps you want to test one set of modules against one program but not another.

Whatever the case, Perl gives you options to manage where it looks for modules. Suppose you have a program in your *~/work* directory that uses a module named `Site::User`. By default, Perl will search all of the directories in the special `@INC` variable for a file named *Site/User.pm*. That may not always include the directory you want (especially if, in this case, you want *~/work/lib*). What can you do?

Within Your Program

The simplest and most self-contained way to change Perl's search path is within your program by using the `lib` pragma. This happens at compile-time [Hack #70], as soon as `perl` encounters the statement, so put it before any `use` line for the module you want to load. For example:

```
use lib 'lib';
use Site::User;
```

adds the *lib/* directory—relative to the current directory—to the front of Perl's search path list. Similarly:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
no lib 'badlib';
use Site::User;
```

removes the *badlib/* directory from Perl's list of search paths. If you have two versions of `Site::User` installed and want to make sure that Perl doesn't pick up the wrong version from the wrong directory, exclude it.

From the Command Line

Sometimes you don't have the option or the desire to modify a program, though, especially when you're merely testing it. In that case, use the `lib` pragma from the command line when invoking the program by using `perl`'s `-M` switch:

```
$ perl -Mlib=lib show_users.pl
```

This is equivalent to `use lib 'lib'`. To exclude a path, prepend a hyphen to `lib`:

```
$ perl -M-lib=badlib show_users.pl
```

The `-I` flag also lets you include paths, but it does not let you *exclude* them.

With an Environment Variable

Of course, modifying every program or remembering to add a command-line switch to every invocation is a tremendous hassle. Fortunately, there's a third option: set the `PERL5LIB` environment variable to a colon-separated list of directories to add to the search path. Depending on your shell, this may be:

```
$ export PERL5LIB=/home/user/work/lib:/home/user/work_test/lib:$PERL5LIB

% setenv PERL5LIB /home/user/work/lib:/home/user/work_test/lib:$PERL5LIB
```

There's no good and easy way to exclude a directory for the search path here; put the correct directory at the front of the path.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

If you put the appropriate invocation in the appropriate startup file (such as */etc/profile* or the equivalent), users do not even have to know that this path is there. Of course, if they run programs from `cron` or another environment without these variables, some paths may not be present.

An easier option may be to write a simple shell script that sets the environment properly and then launches the actual `perl` binary, passing along the command-line options appropriately.

When Recompiling Perl

Your final recourse is to set the appropriate paths when compiling Perl [Hack #67]. This isn't as bad as it sounds, but it does take a little bit more dedication. Once you have downloaded and unpacked Perl, run the `Configure` script. Answer all of the questions appropriately until it asks:

```
Enter a colon-separated set of extra paths to include in perl's @INC
search path, or enter 'none' for no extra paths.

Colon-separated list of additional directories for perl to search? [none]
```

Type there the list of directories to add to Perl's built-in `@INC`. Note that `perl` will search this directory *after* it searches its core directories, so if you want to load something in place of a core module, you must manipulate the path with one of the other techniques.

Hack 30. Reload Modified Modules

Update modules in a running program without restarting.

Developing a long-running program can be a tedious process, especially when starting and stopping it can take several seconds or longer. This is most painful in cases where you just need to make one or two little tweaks to see your results. The Ruby on Rails web programming toolkit in development mode gets it right, automatically noticing when you change a library and reloading it in the running server without you having to do anything.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Perl can do that too.

The Hack

All it takes is a simple module named `Module::Reloader`:

```
package Module::Reloader;

use strict;
use warnings;

my %module_times;

INIT
{
    while (my ($module, $path) = each %INC)
    {
        $module_times{ $module } = -M $path;
    }
}

sub reload
{
    while (my ($module, $time) = each %module_times)
    {
        my $mod_time    = -M $INC{$module};
        next if $time = = $mod_time;

        no warnings 'redefine';
        require ( delete $INC{  $module } );
        $module_times{ $module } = $mod_time;
    }
}

1;
```

At the end of compile time [Hack #70], the module caches the name and modification time of all currently loaded modules. Its `reload( )` function checks the current modification time of each module and reloads any that have changed since the last cache check.

Running the Hack

Use the module as usual. Then, when you want to reload any loaded modules, call `Module::Reloader::reload( )`. In a long-running server process, such as a pure-Perl web server running in development mode for a framework, this is easy to do right before processing a new incoming request.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Provided that the modules being modified don't keep around any weird state between requests, the request will see the new behavior.

Hacking the Hack

The module as written *does* attempt to avoid spurious warning messages by suppressing `Subroutine %s redefined at...` error messages, but a compilation error in a module may cause strange behavior and necessitate a server restart. This is for development purposes only; it's very difficult to write code that behaves perfectly in a production environment—too many things could go wrong.

Changing the definition of classes while you have active instances of those classes can do scary things. It may be worthwhile to exclude certain modules, perhaps by specifying filters for modules to include or to exclude.

This module currently does not erase the symbol tables of reloaded modules; that may be useful in certain circumstances. (It may be hazardous in others, where multiple modules affect symbols in a given package.)

Hack 31. Create Personal Module Bundles

Create a personal bundle of your favorite modules.

It never fails. I'm working on a new computer, a friend's computer, or a work computer and I've installed my favorite modules and written some code.

```
use My::Favorite::Module;
My::Favorite::Module->washes_the_dishes( );
```

Then I run the program.

```
Can't locate My/Favorite/Module.pm in @INC (@INC contains ...
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

I did it again. I forgot to install the one module I really needed. Hopefully it's the last one. Of course, even if you never forget to install your favorites, it's still a pain to laboriously install a bunch of modules every time you have a new Perl installation.

That's where personal bundles come in.

The Hack

A personal bundle is very easy to make. Just create a normal CPAN distribution. You don't even need to write tests for it: the modules you list will (hopefully) test themselves.

Instead, create an empty package with the modules you want listed in your POD contents section [Hack #32]. For example, suppose that you're a testing fanatic. You want to install your favorite testing modules, so you decide to call your bundle `Bundle::Personal::`*`Mine`* (where *Mine* is your PAUSE ID).

```
package Bundle::Personal::Mine;

$VERSION = '0.42';

1;

__END__

=head1 NAME

Bundle::Personal::Mine - My favorite testing modules

=head1 SYNOPSIS

perl -MCPAN -e 'install Bundle::Personal::Mine'

=head1 CONTENTS

Test::Class

Test::Differences

Test::Exception

Test::MockModule

Test::Pod

Test::Pod::Coverage

Test::WWW::Mechanize

=head1 DESCRIPTION
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
My favorite modules.

... rest of POD, if any ...
```

Then just package up your tarball and stow it in a safe place (or even upload it to the CPAN).

Running the Hack

From then on, to install all of your favorite modules, just type **`cpanp i Bundle::Personal::Mine`** for CPANPLUS, **`perl -MCPAN -e 'install Bundle::Personal::Mine'`** for `CPAN`, or whatever your favorite module installation incantation is.

Hacking the Hack

When preparing a personal bundle, be selective about what you include. If you include a module that routinely fails tests, the entire bundle installation might fail. If that happens, try to install the errant module manually and return to installing the bundle. It's generally a bad idea to force the installation of a module with failing tests until you understand why they fail. This is especially true when working on a new machine.

Other uses for such bundles include software development kits, corporate bundles, and application support modules. The CPAN already has bundles for `Bundle::Test`, `Bundle::BioPerl`, `Bundle::MiniVend`, and so on. Go to your favorite CPAN mirror and search for bundles. The bundle you want to create may already exist.

Should you really upload your own bundle to the CPAN? It depends. If you maintain a redistributable application that requires several CPAN modules, creating an installation bundle can help users install it and packagers package it. If you're the only person using your bundle, it probably won't do anyone else much good.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hack 32. Manage Module Installations

Bundle up required modules to make installations easier.

Embracing the Perl way means taking advantage of the CPAN when possible. There are thousands of reusable, easily installable modules that do almost anything you can imagine—including making your coding life much easier and simpler.

Some day you'll have to distribute your software, upgrade Perl, or do something else that means that you can't rely on having all of your existing modules available. Never fear; just create a bundle that the `CPAN` module can use to install all of the necessary modules for you!

The Hack

The `CPAN` module doesn't only download and install modules. It can also give you a catalog of what you have installed on your system. The `autobundle` command takes this list and writes it to a bundle file—a very simple, mostly POD module that `CPAN` can use later (or elsewhere) to install necessary modules.

If you only support one application, you can use a technique such as in "Trace All Used Modules" [Hack #74] to figure out everything you need to install.

All you have to do is launch the shell, issue the `autobundle` command, and note where it creates the bundle file:

```
$ cpan

cpan shell -- CPAN exploration and modules installation (v1.7601)
ReadLine support enabled

cpan> autobundle

# time passes...
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
Wrote bundle file
  /usr/src/.cpan/Bundle/Snapshot_2005_11_13_00.pm
```

Running the Hack

Copy or move the bundle file from its current location. Then when you upgrade or reinstall Perl, or when you move to another box, move the bundle file to the *Bundle/* directory beneath the `CPAN` module's working directory. Then, from the `CPAN` shell in the new machine or installation:

```
$ cpan

cpan shell -- CPAN exploration and modules installation (v1.7601)
ReadLine support enabled

cpan> install Bundle::Snapshot_2005_11_13_00

# time really  passes...
```

It will go through the bundle list in order, intstalling all modules as necessary. At least, it will try.

Hacking the Hack

If you look at the bundle file, you might notice that it includes lots and lots of modules—maybe more than you need and certainly plenty of core modules. Worse yet, depending on how you've configured your `CPAN` and how well the modules you want to install mark their dependencies, you may need to babysit the installation to get it to succeed. You may even have to restart it a few times.

If possible, set `CPAN` to follow all prerequisites without asking when configuring it for the first run. (You can always change it back later.) That will help. The next best thing to do is to prune the module list. When possible, try to arrange dependencies appropriately. (Modules change enough that it's unlikely you'll be able to do this perfectly.)

Finally, you can prune out all of the core modules by running the bundle file through Module::CoreList [Hack #73]. That way, you have a somewhat smaller list of modules to install.

```
use Module::CoreList;

my ($bundle, $version) = @ARGV;
$version             ||= $];
@ARGV                  = $bundle;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
my $core_list          = $Module::CoreList::version{ $version };
die "Unknown version $version\\n" unless $core_list;

# find module list
while (<>)
{
    print;
    last if $_ eq "=head1 CONTENTS\\n";
}

print "\\n";

# process only module/version lines
while (<>)
{
    if ( $_ eq "=head1 CONFIGURATION\\n" )
    {
        print;
        last;
    }

    chomp;
    next unless $_;

    my ($module, $version) = split( /\\s+/, $_ );
    $version = 0 if $version eq 'undef';

    next if exists $core_list->{ $module }
                and $core_list->{ $module } >= $version;

    print "$module $version\\n\\n";
}

# print everything else
print while <>;
```

Run this program, passing the name of the bundle file and, optionally, the version of Perl against which to check. Redirect the output to a new bundle file:

```
$ perl prune_bundle.pl Snapshot_2005_11_03_00.pm > PrunedSnapshot.pm
$
```

Now you have an easier time deciding which modules you really need to install.

Hack 33. Presolve Module Paths

Make programs on complex installations start more quickly.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

In certain circumstances, one of Perl's major strengths can be a weakness. Even though you can manipulate where Perl looks for modules (@INC) at runtime according to your needs [Hack #29], and even though you can use thousands of modules from the CPAN, your system has to find and load these modules.

For a short-running, repeated program, this can be expensive, especially if you have many paths in `@INC` from custom testing paths, sitewide paths, staging servers, business-wide repositories, and the like. Fortunately, there's more than one way to solve this. One approach is to resolve all of the paths just once, and then use your program as normal.

The Hack

"Trace All Used Modules" [Hack #74] shows how putting a code reference into `@INC` allows you to execute code every time you `use` or `require` a module. That works here, too.

```
package Devel::Presolve;

use strict;
use warnings;

my @track;

BEGIN { unshift @INC, \\&resolve_path }

sub resolve_path
{
    my ($code, $module) = @_;
    push @track, $module;
    return;
}

INIT
{
    print "BEGIN\\n{\\n";

    for my $tracked (@track)
    {
        print "\\trequire( \\$INC{'$tracked'} = '$INC{$tracked}' );\\n";
    }

    print "}\\n1;\\n";
    exit;
}

1;
```

`Devel::Presolve`'s `resolve_path( )` captures every request to load a module, stores the module name, and returns. Thus Perl attempts to load the module as normal. After the entire program has finished compiling, but before it starts to run [Hack #70], it prints to

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

STDOUT a `BEGIN` block that loads all of the modules by absolute filepath then exits the program.

Running the Hack

Put `Devel::Presolve` somewhere in your path. Then run your slow-starting program while loading the module. Redirect the output to a file of your choosing:

```
$ perl -MDevel::Preload slow_program.pl > preload.pm
```

preload.pm will contain something similar to:

```
BEGIN
{
    require( $INC{'CGI.pm'}      = '/usr/lib/perl5/5.8.7/CGI.pm' );
    require( $INC{'CGI/Util.pm'} = '/usr/lib/perl5/5.8.7/CGI/Util.pm' );
    require( $INC{'vars.pm'}     = '/usr/lib/perl5/5.8.7/vars.pm' );
    require( $INC{'constant.pm'} = '/usr/lib/perl5/5.8.7/constant.pm' );
    require( $INC{'overload.pm'} = '/usr/lib/perl5/5.8.7/overload.pm' );
}

1;
```

You can either include the contents of this file at the start of *slow_program.pl* or load it as the first module. If you do the latter, put the file in a directory at the *front* of `@INC`, lest you erase any performance gains.

Note that the trick of assigning to `%INC` within the `require` avoids a potentially nasty module-reloading bug, where Perl doesn't see `require '/usr/lib/perl5/5.8.7./CGI.pm'` as loading the same file as `use CGI;` does.

Hacking the Hack

Pre-resolving paths likely won't help long-running programs. For short-running programs where startup time can dwarf calculation time, it may, depending on how complex your `@INC` is. Be especially careful that upgrading Perl or installing new versions of modules may invalidate this cache—it *is* a cache—and cause strange errors. This technique may work better only when you want to deploy a program to a production system, but likely not when you're merely developing or testing.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hack 34. Create a Standard Module Toolkit

Curb your addiction to explicit `use` statements.

Most experienced Perl programmers rely on a core set of modules and subroutines that they use in just about every application they create. For example, if you work with XML documents on a daily basis (and you certainly have our deepest sympathy there), then you probably use either `XML::Parser` or `XML::SAX` or `XML::We::Built::Our::Own::Damn::Solution` all the time.

If those documents contain lists of files that you need to manipulate, then you probably use `File::Spec` or `File::Spec::Functions` as well, and perhaps `File::Find` too. Maybe you need to verify and manipulate dates and times on those files, so you regularly pull in half a dozen of the `DateTime` modules.

If the application has an interactive component, you might continually need to use the prompt( ) subroutine from IO::Prompt [Hack #14]. Likewise, you might frequently make use of the efficient `slurp( )` function from `File::Slurp`. You might also like to have Smart::Comments instantly available [Hack #54] to simplify debugging. Of course, you always specify `use strict` and `use warnings`, and probably `use Carp` as well.

A Mess of Modules

This adds up to a tediously long list of standard modules, most of which you need to load every time you write a new application:

```
#! /usr/bin/perl

use strict;
use warnings;
use Carp;
use Smart::Comments;
use XML::Parser;
use File::Spec;
use IO::Prompt qw( prompt );
use File::Spec::Functions;
use File::Slurp qw( slurp );
use DateTime;
use DateTime::Duration;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
use DateTime::TimeZone;
use DateTime::TimeZone::Antarctica::Mawson;
# etc.
# etc.
```

It would be great if you could shove all these usual suspects in a single file:

```
package Std::Modules;

use strict;
use warnings;
use Carp;
use Smart::Comments;
use XML::Parser;
use File::Spec;
use IO::Prompt qw( prompt );
use File::Spec::Functions;
use File::Slurp qw( slurp );
use DateTime;
use DateTime::Duration;
use DateTime::TimeZone;
use DateTime::TimeZone::Antarctica::Mawson;
# etc.

1;
```

and just use that one module instead:

```
#! /usr/bin/perl

use Std::Modules;
```

Of course, that fails dismally. Using a module that uses other modules isn't the same as using those other modules directly. In most cases, you'd be importing the components you need into the wrong namespace (into `Std::Modules` instead of `main`) or into the wrong lexical scope (for `use strict` and `use warnings`).

The Hack

What you really need is a way to create a far more cunning module: one that cuts-and-pastes any `use` statements inside it into any file that uses the module. The easiest way to accomplish that kind of sneakiness is with the `Filter::Macro` CPAN module. As its name suggests, this module is a source filter that converts what follows it into a macro. Perl then replaces any subsequent `use` of that macro-ized module with the contents of the module. For example:

```
package Std::Modules;
use Filter::Macro;     # <-- The magic happens here
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
use strict;
use warnings;
use Carp;
use Smart::Comments;
use XML::Parser;
use File::Spec;
use IO::Prompt qw( prompt );
use File::Spec::Functions;
use File::Slurp qw( slurp );
use DateTime;
use DateTime::Duration;
use DateTime::TimeZone;
use DateTime::TimeZone::Antarctica::Mawson;
# etc.
# etc.

1;
```

Now, whenever you write:

```
#! /usr/bin/perl

use Std::Modules;
```

all of those other `use` statements inside `Std::Modules` are pasted into your code, in place of the `use Std::Modules` statement itself.

Hacking the Hack

There's also a more modular and powerful variation on this idea available. The `Toolkit` module (also on CPAN) allows you to specify a collection of standard module inclusions as separate files in a standard directory structure. Once you have them set up, you can automatically use them all just by writing:

```
#! /usr/bin/perl

use Toolkit;
```

The advantage of this approach is that you can also set up "conditional usages"—files that tell `Toolkit` to import specific subroutines from specific modules, but only when something actually uses those subroutines. For example, you can tell `Toolkit` not to always load:

```
use IO::Prompt qw( prompt );
use File::Slurp qw( slurp );
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

but only to load the `IO::Prompt` module if something actually uses the `prompt( )` subroutine, and likewise to defer loading `File::Slurp` for `slurp( )` until actually necessary.

That way, you can safely specify dozens of handy subroutines and modules in your standard toolkit, but only pay the loading costs for those you actually use.

Hack 35. Write Demos from Tutorials

Give tutorial readers example code to run, tweak, and examine.

Reading code is one thing. Running code is another. Example code is wonderfully useful, but nothing beats playing with it—changing values, moving subroutines around, and seeing what happens if you touch just one more thing.

You'll never escape writing documentation. You *can* escape having to explain the basics over and over again if that documentation includes working, runnable code that people can customize for their needs. If you've already realized that including pure-POD modules is a great way to document programs, take the next step and make the tutorials themselves write out their examples.

The Hack

Writing a POD-only tutorial is easy. For example, the basic `SDL::Tutorial` shows how to create a screen using Perl and the SDL bindings:

```
use SDL::App;

# change these values as necessary
my  $title                  = 'My SDL App';
my ($width, $height, $depth) = ( 640, 480, 16 );

my $app = SDL::App->new(
    -width  => $width,
    -height => $height,
    -depth  => $depth,
    -title  => $title,
);
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
# your code here; remove the next line
sleep 2;
```

Running the Hack

Better yet, if you run the tutorial from the command line, it writes out this program to a file of your choosing:

```
$ perl -MSDL::Tutorial=sdl_demo.pl -e 1
```

Looking at the tutorial itself [Hack #2], it's only a `use` statement, a heredoc, and the documentation. How does `Pod::ToDemo` know to write the file and exit? Further, what if someone accidentally uses `SDL::Tutorial` as a module within a real program—will it write or overwrite a file and throw an error?

Nope; that's part of the magic.

Inside the Hack

`Pod::ToDemo` has two tricks. The first is writing code that will execute when you run the demo file from the command line only. `caller( )` isn't just for checking the calling subroutine—it walks the entire call stack. The module's `import( )` method has this code:

```
my @command = caller( 3 );

return if @command and $command[1] ne '-e';
```

That is, look three levels up the call stack. If the filename of that call frame is *not* `-e` (the correct command-line invocation to write a demo file), then someone has accidentally used a demo module in a real program, and the `import( )` method returns without doing anything.

Why three levels? The previous level is the implicit BEGIN block surrounding the use call in the demo module [Hack #70]. The next one up is the load of the demo module itself (`-M` on the command line creates its own block). The top level is the command-line invocation itself.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The rest of the `import( )` method itself merely installs another method into the demo module, calling it `import( )`. By the time the rest of the module finishes compiling, when Perl goes to call that module's `import( )`, it'll be there—and it can write the file as necessary.

Hacking the Hack

This is the easy way to use `Pod::ToDemo`. There's also a more difficult way. Consider if you already show the example code within the tutorial, perhaps in one large chunk and perhaps not. Duplicating the code within the string to pass to `Pod::ToDemo` and the tutorial itself would be a mistake. In that case, generate the code however you like, pulling it out of the POD, and pass a subroutine reference to `Pod::ToDemo`. The module will call that instead, when appropriate, letting you write the demo file as you like.

This trick would also work to parameterize the demo file based on command-line arguments.

Hack 36. Replace Bad Code from the Outside

Patch buggy modules without touching their source code.

Until computers finally decide to do what we mean, not what we say, programs will have bugs. Some you can work around. Others are severe enough that you have to modify source code.

When the bugs are in code you don't maintain and you don't have a workaround, Perl's dynamic nature can be an advantage. Instead of keeping local copies of externally managed code, sometimes patching that code from the outside is the simplest way to make your code work again.

The Hack

Imagine that you're building a large application that uses a hypothetical `Parser` module that, for whatever reason, calls `exit( )`, not `die( )`, when it fails. The relevant code might look something like:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
package Parser;

sub parse
{
    my ($class, $text) = @_;
    validate_text( $shift );
    bless \\$text, $class;
}

sub validate_text
{
    my $text = shift;
    exit 1 unless $text =~ /^</;
}

1;
```

You might normally expect to use this module with code such as:

```
use Parser;

my $parser = eval { Parser->parse( 'some example text' ) };
die "Bad input to parser: $@\\n" if $@;
```

However because of the `exit( )`, your program will end. It may be perfectly legitimate that the text to parse in this example is invalid, so `Parser` can't handle it, but the `exit( )` is just wrong—it gives you no opportunity to alert the user or try to fix the problem. If `validate_text( )` were a method, you could subclass the module and override it, but you don't have this option.

Fortunately, you *can* override the `exit( )` keyword with a function of your own, if you do it at the right time:

```
package Parser;
use subs 'exit';
package main;

use Parser;
sub Parser::exit{die shift;}
```

Before Perl can parse the `Parser` package, you must tell it to consider all occurrences of `exit( )` as calls to a user-defined function, not the built-in operator. That's the point of switching packages and using the `subs` pragma.

Back in the `main` package, the odd-looking subroutine declaration merely declares the actual implementation of that subroutine. Now instead of exiting, all code that calls `exit( )` in `Parser` will throw an exception instead.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hacking the Hack

If you don't really care about validation, if you prefer a sledgehammer solution, or if you don't want to replace `exit( )` in the entire package, you can replace the entire `validate_text ( )` function:

```
use Parser;
local *Parser::validate_text;
*Parser::validate_text = sub
{
    my $text = shift;
    die "Invalid text '$text'\\n" unless $text =~ /^</;
};
```

Doing this in two steps avoids a warning about using a symbol name only once. Using `local` replaces the code only in your current dynamic scope, so any code you call *from* this scope will use this function instead of the old version.

To replace the subroutine globally, use `Parser` as normal, but remove the line that starts with `local`. Replace it with `no warnings 'redefine';` to avoid a different warning.

If you need to switch behavior, make the replacement `validate_text( )` into a closure, setting a lexical flag to determine which behavior to support. This variant technique is highly useful in testing code.

Hack 37. Drink to the CPAN

Play London.pm's CPAN Drinking Game—but responsibly.

The CPAN drinking game tests your knowledge of the CPAN. The goal of the game, depending on who you ask, is either to prove that you have an incredibly deep knowledge of the CPAN or to get incredibly drunk. An alternate goal is to learn about modules you never even knew existed. Just try to remember them.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Running the Hack

The first player, Audrey, takes a drink and names a CPAN module: `Devel::Cover`. Play passes to Barbie, who's sitting immediately to Audrey's right. Barbie needs to drink and then come up with a released module which starts with `C`, the first letter of the *last* part of Audrey's module. If he can't, he drinks and play passes to the next player.

If Barbie names a module with three parts, perhaps `Crypt::SSLeay::X509`, play skips over chromatic, who's sitting to his right. The same applies if he managed to pull out a module name with four, five, or more parts.

Domm picks up with `X`. He drinks and pulls out `XML::XPath`. Because the last part starts with the same letter as the first part, the direction of play reverses and it's chromatic's turn.

chromatic drinks and, sadly, can't come up with anything and has to pass. He's now out of the game. Audrey drinks and names `XML::Simple`. Play continues counterclockwise to Domm, who needs to come up with something starting with `S`.

The winner is the last remaining player.

Hacking the Hack

Try whiskey!

Seriously, as bar-rific as the game sounds, you don't have to drink alcohol. Try another beverage—hot tea is good, root beer is good, and anything with caffeine can change the rule for losing in interesting ways.

Some variants of the game require Barbie to drink *until* he can name a module. This can take a while.

The author recommends never challenging Audrey to the CPAN drinking game.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hack 38. Improve Exceptional Conditions

Die with style when something goes wrong.

Perl's exception handling is sufficiently minimal. It's easy to recover when things go wrong without having to declare every possible type of error you might possibly encounter. Yet there are times when you know you can handle certain types of exceptions, if not others. Fortunately, Perl's special exception variable `$@` is more special than you might know—it can hold objects.

If you can stick an object in there, you can do just about anything.

The Hack

How would you like more context when you catch an exception? Sure, if someone uses the `Carp` module you can sometimes get a stack trace. That's not enough if you want to know exactly what went wrong.

For example, consider the canonical example of a system call gone awry. Try to open a file you can't touch. Good style says you should `die( )` with an exception there. Robust code should catch that exception—but there's so much useful information that the exception string could hold, why should you have to parse the message to figure out which file it was, for example, or what the error was, or how the user tried to open the file?

`Exception::Class` lets you throw an exception as normal while making all of the information available through instance methods.

Suppose you've factored out all of your file opening code into a single function:

```
use File::Exception;

sub open_file
{
    my ($name, $mode) = @_;

    open my $fh, $mode, $name or
        File::Exception->throw( file => $name, mode => $mode, error => $! );
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
    return $fh;
}
```

Instead of calling `die( )`, the function `throw( )`s a new `File::Exception` object, passing the file name, mode, and system error message. `File::Exception` subclasses `Exception::Class::Base` to add two more fields and a friendlier error message:

```
package File::Exception;

use SUPER;
use Exception::Class;

use base 'Exception::Class::Base';

sub Fields
{
    my $self = shift;
    return super( ), qw( file mode );
}

sub file { $_[0]->{file} }
sub mode { $_[0]->{mode} }

sub full_message
{
    my $self = shift;
    my $msg  = $self->message( );

    my $file = $self->file( );
    my $mode = $self->mode( );

    return "Exception '$msg' when opening file '$file' with mode '$mode'";
}

1;
```

The only curious piece of the code is the `Fields( )` method. `Exception::Class::Base` uses this to initialize the object with the proper attributes.

`full_message( )` creates and returns the string used as the exception message. This is what `$@` would contain if this were a normal exception. As it is, Exception::Class::Base overrides object stringification [Hack #99] so the objects appear as normal `die( )` messages to users who don't realize they're objects.

Running the Hack

Call `open_file( )` as usual—within an `eval( )` block:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
my $fh;
$fh = eval { open_file( '/dev/null', '<' ) };
warn $@ if $@;

$fh = eval { open_file( '/dev', '>' ) };
warn $@ if $@;
```

Reading from */dev/null* is okay (at least on Unix-like systems), but writing to */dev* or any other directory is a problem:

```
Exception 'Is a directory' when opening file '/dev' with mode '>'
    at directory_whacker.pl line 10.
```

The real power comes when you treat the object as an object:

```
$fh = eval { open_file( '/dev', '>' ) };

if (my $error = $@)
{
    warn sprintf "Tried to open %s '%s' as user %s at %s: %s\\n",
        $error->mode( ), $error->file( ), $error->uid( ),
        scalar( localtime( $error->time( ) ) ),
        $error->error( );
}
```

What are the other methods? They're methods available on all `Exception::Class` objects.

Make a copy of `$@` as soon as possible, lest another `eval( )` block somewhere overwrite your object out from underneath you.

Now instead of having to parse the string for potentially useful information, you can debug and, if possible, recover with better debugging information:

```
Tried to open > '/dev' as user 1000 at Tue Jan 17 21:58:00 2006:
    Is a directory
```

Hacking the Hack

`Exception::Class` objects are objects—so they can have relationships with each other. You can subclass them and make an entire hierarchy of exceptions, if your application needs them. You can also catch and redispatch them based on their type or any other characteristic you want.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Best of all, if someone doesn't want to care that you're throwing objects, she doesn't have to. They still behave just like normal exceptions.

Hack 39. Search CPAN Modules Locally

Search the CPAN without leaving the command line.

Websites such as http://search.cpan.org/ are fantastic for finding the Perl module you need from the CPAN, but firing up a web browser, navigating to the page, and waiting for the results can be slow.

Similarly, running the CPAN or CPANPLUS shell and doing i `search term` is also slow. Besides that, you might not even have a network connection.

The Hack

The last time the CPAN or CPANPLUS shell connected to a CPAN mirror it downloaded a file listing every single module—*03modlist.data.gz*. You can see the file at ftp://cpan.org/modules/03modlist.data.gz. Because you have that local copy, you can parse it, check the modules that match your search terms, and print the results.

Additionally you can check to see if any of them are installed already and highlight them.

```
#!perl -w

# import merrily
use strict;
use IO::Zlib;
use Parse::CPAN::Modlist;

# get the search pattern
my $pattern    = shift || die "You must pass a pattern\\n";
my $pattern_re = qr/$pattern/;

# munge our name
my $self       = $0; $self =~ s!^.*[\\\\/]!!;

# naughty user
die ("usage : $self <query>\\n") unless defined $pattern;
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
# get where the local modulelist is from CPAN(PLUS?)::Config
my $base;
eval { require CPANPLUS::Config; CPANPLUS::Config->import( ); };
unless ($@)
{
    my $conf = CPANPLUS::Config->new( );
    # different versions have the config in different places
    for (qw(conf _build))
    {
        $base = $conf->{$_}->{base} if exists $conf->{$_};
    }
}

goto SKIP if defined $base;

eval { require CPAN::Config; CPAN::Config->import( ) };

unless ($@)
{
    local $CPAN::Config;
    $base = $CPAN::Config->{'keep_source_where'}."/modules/";
}

goto SKIP if defined $base;

die "Couldn't find where you keep your CPAN Modlist\\n";

SKIP:
my $file     = "${base}/03modlist.data.gz";

# open the file and feed it to the mod list parser
my $fh       = IO::Zlib->new($file, "rb")  or die "Cannot open $file\\n";
my $ml       = Parse::CPAN::Modlist->new(join "", <$fh>);

# by default we want colour
my $colour   = 1;

# check to see if we have Term::ANSIColor installed
eval { require Term::ANSIColor };

# but if we can't have it then we can't have it
$colour      = 0 if $@;

# now do the actual checking

my $first    = 0;

# check each module
for my $module (map { $ml->module($_) } $ml->modules( ))
{
    my $name = $module->name( );
    my $desc = $module->description( );

    # check to see if the pattern matches the name or desc
    next unless  $name =~ /$pattern_re/i or $desc =~ /$pattern_re/i;

    # aesthetics
    print "\\n-- Results for '$pattern' --\\n\\n" unless $first++;

    # check to see if it's installed
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
        eval  "require $name";

        # print out the title - coloured if possible
        if ( $colour && !$@ )
        {
              print Term::ANSIColor::color('red'),
                  "$name\\n",
                  Term::ANSIColor::color('reset');
        }
        elsif (!$@)
        {
            print "!! $name\\n";
        }
        else
        {
            print "$name\\n";
        }

        # print out the name and description
        print "- $desc\\n\\n";
    }

    exit 0;
```

First, the code tries to find the local module list. This can be in several places. It initially checks for `CPANPLUS`, assuming that anyone who has that installed will use it over the less featureful `CPAN`. Different versions of `CPANPLUS` store the file in different locations, so the code checks both.

If that fails, the program performs the same check for `CPAN`. If that doesn't work, the program ends.

If the file is present, the code uncompresses it with `IO::Zlib` and passes it to `Parse::CPAN::Modlist` to parse it.

The next part checks to see if `Term::ANSIColor` is available. If so, it can highlight installed modules.

The `Parse::CPAN::Modlist::modules( )` method returns only the names of modules in the list, so the code must load the appropriate `Module` object to get at the other metadata. Using `map { }` in the `for` loop is incredibly convenient.

For efficency, there's an early check if the name or description matches the input pattern. Notice how the results banner (`Results for '$pattern'`) only prints if there is at least one result.

The code attempts to `require` the module to see if it is available. If so, the program must highlight the name with color, if available, or exclamation marks otherwise. Finally, the program prints the description and tries the next module.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hacking the Hack

There are plenty of ways to improve this program.

Currently it assumes that the `CPANPLUS` module list is the most up to date. It should probably check both `CPANPLUS` and `CPAN` if possible, look for the appropriate *03modlist.data.gz* in each case, and push it onto a list of potential candidates before using the most recently modified version.

This hack also relies on *03modlist.data.gz* being up to date. If you don't use the CPAN or CPANPLUS shell regularly, this might not be the case.

There are several possible solutions.

First, the program could just die if the module list is too old. This is the simplest (and most defeatist) solution.

Second, you could write a cron job that periodically updates the module list. This has the advantage that even if you have no network currently available, you know it's still reasonably fresh.

Finally, you could check to see whether the module list is older than a certain threshold. If so, you could warn or force the chosen provider to download a newer one. This has the disadvantage of not working if you cannot connect to your CPAN mirror.

Currently, the code checks both the name and the description—which can produce a lot of useless results. It should be possible to build a more complicated query parser that gives users finer-grained control over the results.

Finally, the code doesn't necessarily have to `require` modules to see if they exist. It could use logic similar to perldoc -l [Hack #2] to find their locations.

Hack 40. Package Standalone Perl Applications

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Distribute a full Perl application to users.

The three main ways to distribute an application are via an installer, via a standalone executable, or via source. These choices vary a lot across platforms. Windows users prefer installers, especially *.msi* files. Mac fans are quite happy with *.app* files, which usually come in disk images. Most Linux variants use installers (*.deb* and *.rpm*) but others prefer source.

What if your application is a Perl program?

Perl may seem like an atypical GUI language, but it does have bindings for GUI toolkits including Tk, wxWidgets, Qt, and GTK. Perl can be useful in the GUI realm as a rapid-development foundation or simply to add a couple of dialogs to a mostly background process. One great entry barrier, however, is that most platforms do not bundle these GUI toolkits with Perl—and some platforms do not bundle Perl at all. Though there are packaged distributions of Perl itself, the add-on modules that usually accompany any sophisticated Perl project are typically source code. This poses a problem for most Windows users and many Mac users for whom this is too low-level a task. Only the sysadmin-rich world of Linux and Unix regularly tolerates `sudo cpan install Foo` commands.

The Hack

The PAR project attempts to to create a solution to bundling the myriad files that usually compose a Perl application into a manageable monolith. PAR files are simply ZIP files with manifests. If you have PAR installed on your computer, you can write Perl code that looks like:

```
#!perl -w

use PAR 'foo.par';
use Foo;
...
```

and if *Foo.pm* is inside the *foo.par* file, `perl` will load it as if it were a normal installed module. Even more interestingly, you can write:

```
#!perl -w

use PAR 'http://www.example.com/foo.par';
use Foo;
...
```

which will download and cache the *foo.par* archive locally. How's that for a quick update?

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

You may have noticed the sticky phrase above "If you have PAR installed..." That is a catch-22 of sorts. PAR helps users to skip the software installation steps, but first they have to...install software!

To get around this, PAR takes another page from the ZIP playbook: self-extracting executables. The PAR distibution comes with a program called `pp` that allows a developer to wrap the core of Perl and any additional project-specific Perl modules into a PAR file with a *main.pl* and an executable header to bootstrap the whole thing. This produces something like `/usr/bin/perl` with all of its modules embedded inside.

Running the Hack

Consider a basic `helloworld.pl` application:

```
#!perl -w

use strict;
use Tk;

my $mw = MainWindow->new( );

$mw->Label(-text => 'Hello, world!')->pack( );
$mw->Button(-text => 'Quit', -command => sub { exit })->pack( );

MainLoop( );
```

To run this, you have to have Perl and Tk installed[1] and perhaps X11 running (via `open /Applications/Utilities/X11.app`). Run `perl helloworld.pl` to see a window like that in Figure 4-1.

[1] On my Mac OS X 10.4 box, I do this via `fink install tk-pm586`

Figure 4-1. "Hello, world" in Perl/Tk

Now suppose that you want to give this cool new application to other Mac users. Telling them to first install Fink, Tk, and X11 just for "Hello, World!" is ludicrous. Instead, build an executable with `pp`:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
% pp -o helloworld helloworld.pl
```

That creates a 3 MB executable, *helloworld*, which includes the entirety of both Perl and Tk. Send it to a friend who has a Mac (and X11, because this version of Tk isn't Aqua-friendly) and she can run it. If you were to make a Windows version it would be even easier on end users—on Windows, Tk binds directly to the native GUI, so X11 is not a prerequisite.

Aside from portability, another PAR benefit is version independence. The example executable, though built against Perl 5.8.6 on Mac OS X 10.4, should also work well on 10.3 or 10.2, even though those OSes shipped with older versions of Perl. This is because PAR included every part of 5.8.6 that the example needed in the executable.

Hacking the Hack

If you download that executable, you can open it with any zip tool:

```
% zipinfo helloworld
Archive:  helloworld   3013468 bytes   689 files
drwxr-xr-x  2.0 unx        0 b- stor 23-Oct-05 14:21 lib/
drwxr-xr-x  2.0 unx        0 b- stor 23-Oct-05 14:21 script/
-rw-r--r--  2.0 unx    20016 b- defN 23-Oct-05 14:21 MANIFEST
-rw-r--r--  2.0 unx      210 b- defN 23-Oct-05 14:21 META.yml
-rw-r--r--  2.0 unx     4971 b- defN 23-Oct-05 14:21 lib/AutoLoader.pm
-rw-r--r--  2.0 unx     4145 b- defN 23-Oct-05 14:21 lib/Carp.pm
... [snipped 679 lines] ...
-rw-r--r--  2.0 unx    12966 b- defN 23-Oct-05 14:21 lib/warnings.pm
-rw-r--r--  2.0 unx      787 b- defN 23-Oct-05 14:21 lib/warnings/register.pm
-rw-r--r--  2.0 unx      186 t- defN 23-May-05 22:22 script/helloworld.pl
-rw-r--r--  2.0 unx      262 b- defN 23-Oct-05 14:21 script/main.pl
689 files, 2742583 bytes uncompressed, 1078413 bytes compressed:  60.7%
```

You may see that the file sizes don't match. That's because the EXE also contains the whole Perl interpreter outside of the ZIP portion. That adds an extra 200% to file size in this case.

Is it fast? No. Perl must unzip the file prior to use (which happens automatically, of course). Is it compact? No, 3 MB for Hello World is almost silly. Is it convenient? Yes—and that is often the most important quality when shipping software to users.

An interesting consequence of this distribution model is that the executable contains all of the source code. For some companies this may represent a problem (with some possible solutions listed at http://par.perl.org/). On the other hand it is also a benefit in that you might satisfy any GPL requirements without having to offer a separate source download.

An important note for Windows is that, thanks to ActiveState, you do not need a C compiler to build Perl yourself. They provide an installable package that includes Tk pre-built. See links on http://par.perl.org/ for pre-compiled installers for PAR.

Hack 41. Create Your Own Lexical Warnings

Add your own warnings to the `warnings` pragma.

Perl 5.6 added a useful pragma called `warnings` that expanded and enhanced upon the `-w` and `-W` switches. This pragma introduced warnings scoped lexically. Within a lexical scope you can enable and disable warnings as a whole or by particular class.

For example, within a `say( )` function emulating the Perl 6 operator, you could respect the current value of `$,` (the output field separator) and not throw useless warnings about its definedness with:

```
use warnings;

# ... more code here...

sub say
{
    no warnings 'uninitialized';
    print join( $,, @_ ), "\\n";
}
```

See `perllexwarn` for a list of all of the types of warnings you can enable and disable.

When you write your own module, you can even *create* your own warnings categories for users of your code to enable and disable as they see fit. It's easy.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The Hack

To create a warning, use the `warnings::register` pragma in your code. That's it. The `UNIVERSAL::can` module[2] does this.

[2] Which detects, reports, and attempts to fix the anti-pattern of people calling `UNIVERSAL::can( )` as a function, not a method.

Within the module, when it detects code that uses `UNIVERSAL::can( )` as a function, it checks that the calling code has enabled warnings, then uses `warnings::warn( )` to report the error:

```
if (warnings::enabled( ))
{
    warnings::warn( "Called UNIVERSAL::can( ) as a function, not a method" );
}
```

Running the Hack

How does this look from code that merely uses the module? If the calling code doesn't use warnings, nothing happens. Otherwise, it warns as normal. To enable or disable the *specific* class of warning, use:

```
# enable
use warnings 'UNIVERSAL::can';

# disable
no warnings 'UNIVERSAL::can';
```

Hacking the Hack

You can also re-use existing warnings categories. For example, if you want to mark a particular interface as deprecated, write a wrapper for the new function that warns when users use the old one:

```
sub yucky_function
{
    my ($package, $filename, $line) = caller( );

    warnings::warnif( 'deprecated',
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
            "yucky_function( ) is deprecated at $filename:$line\\n" );

    goto &yummy_function;
}
```

This version of `goto` replaces the original call in the call stack by calling the new function with the current contents of `@_`.

Now when users use the `warnings` pragma with no arguments (or enable `deprecated` warnings), they'll receive a warning suggesting where and how to update their code.

Hack 42. Find and Report Module Bugs

Fix problems in CPAN modules.

In an ideal world, all software is fully tested and bug free. Of course that's rarely the case.

Using Perl modules offers many advantages, including more thoroughly validated routines, tested and optimized solutions, and the fact that someone has already done part of your job for you. Sometimes, though, you may find that the shiny module that does exactly what you need actually does something different than it should have.

Here's some code that creates a proxy object `FooProxy`. When you create an instance of this proxy object, it should behave just like an instance of the original `Foo` object, but `FooProxy` could modify specific behavior of the `Foo` object, perhaps to log method calls or check access [Hack #48], without altering the `Foo` package itself:

```
package FooProxy;

sub new
{
    my $class = shift;
    my $foo   = Foo->new( @_ );
    bless \\$foo, $class;
}

sub can
{
```

```
    my $self = shift;
    return $$self->can( @_ );
}

1;
```

Here's some code that instantiates a `FooProxy` object, and being paranoid, attempts to double-check that the created object looks just like a `Foo` object:

```
# Create a proxy object
my $proxy = FooProxy->new( );

# Make sure the proxy acts like a Foo
if ($proxy->isa('Foo'))
{
    print "Proxy is a Foo!\\n";
}
else
{
    die "Proxy isn't a Foo!";
}
```

When you run this script, you might notice a problem. When you call a `Foo` method on the `$fooproxy` object, the method complains that the object isn't `Foo`. What's going on?

Instead of diving straight into the debugger or throwing print statements throughout the code, step back and take a logical approach [Hack #53]. Here's the `Foo` definition:

```
package Foo;
use UNIVERSAL::isa;

sub new
{
    my $class = shift;
    bless \\my $foo, $class;
}

sub isa
{
    1;
}

1;
```

`Foo` uses the CPAN module `UNIVERSAL::isa` to protect itself against people calling the method `UNIVERSAL::isa( )` as a function.[3] When someone calls `UNIVERSAL::isa ( $some_foo, 'Class' )`, `UNIVERSAL::isa` should detect the `isa( )` method of the `Foo` object, and call that. In this case, though, `isa( )` is executing in the context of `FooProxy`. This looks like a problem with the `UNIVERSAL::isa` module; you should file a bug report!

[3] `Foo` defines its own `isa( )`, so you *must* call `$some_foo->isa( )` instead.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Write a Test

Instead of just reporting the bug generically and leaving the author to diagnose, fix, and verify, give the author an excellent head start by writing a test. Taking it one step further, you can even add this test directly to the module's own collection of tests. After downloading and unpacking *UNIVERSAL-isa-0.05.tar.gz*, look for its *t/* subdirectory. Each *.t* file in this directory represents a unit test with one to many subtests. Add a new test to the package by creating a new *.t* file. `UNIVERSAL::isa`, however, already includes a *bugs.t* file, so you can just add the new test there.

You could rewrite the example code and add it to *bugs.t*. Just don't forget to increment the test count appropriately, because you're adding tests:

```
# really delegates calls to Foo
{
    package FooProxy;

    sub new
    {
        my $class = shift;
        my $foo   = Foo->new( @_ );
        bless \\$foo, $class;
    }

    sub can
    {
        my $self = shift;
        return $$self->can( @_ );
    }
}

my $proxy = FooProxy->new( );
isa_ok( $proxy, 'Foo' );
```

Run the test and make sure it fails. If so, it's a good test; it demonstrates what you consider to be a real bug.

Running the test is usually as simple as:

```
$ prove -lv t/bugs.t
# test output here...
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Submitting a bug report

Now you've done a lot of the work for the author. Not only have you narrowed the problem down to a particular module, you have produced a test case that he or she can include with the module to ensure that the bug gets fixed and stays fixed in future revisions. Instead of submitting a bug report that merely explains what you think the problem is (or just the symptom), you can provide an implemented test case that demonstrates the problem and will prove that the ultimate fix really works.

It's helpful to have the Perl community review your findings to confirm your analysis. Perl Monks (http://www.perlmonks.org/) is a free community for Perl programmers. Many of the best-known names in the Perl community—authors, instructors, and even language designers—frequent Perl Monks and dispense their wisdom freely. It's easy to be sure that you've found a legitimate bug, only to find out that you misunderstood the expected behavior. Further, you might get more useful feedback, such as a pointer that the module you're using is outdated, and there's a much better replacement, or that another module more closely meets your needs.

Once you have confirmation that this is a bug, submit your report to the CPAN Request Tracker at http://rt.cpan.org/.[4] This site provides a simple interface to submit bug reports to the appropriate package maintainer, and then check the status of the report. This site supports user accounts (including your existing PAUSE ID) that are useful for tracking your numerous bug reports, but you don't have to create an account. If you choose to continue without an account, you may specify an email address with the bug report, and you'll receive updates when the module maintainer updates your ticket.

[4] Of course, the author might prefer another means of reporting. Check the module's documentation to be sure.

On the site, first search for distributions. This will give you a form where you can enter the package distribution name, `UNIVERSAL::isa`, and find a list of active bugs against it. From here, you can report your new bug, assuming someone else hasn't already submitted it!

In the submission form, fill in the requested information. For the subject, please be specific and concise. Instead of `UNIVERSAL::isa is broken`, consider `isa( ) reports incorrect package type (?)`. Choose an appropriate severity, and indicate the module version or versions in which you observed the defect. There's a box to describe in more detail what you observed and how this behavior differs from your expectation. Note the comments on the submission page that suggest other useful information to include.

There is also a place to attach a file. Along with the basic bug report, you can submit a patch to the module to add your test case. To create the patch, extract the package, creating a

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

versioned directory with the pure downloaded form. Next, copy that package directory to another directory without the version number:

```
$ cp -r UNIVERSAL-isa-0.05 UNIVERSAL-isa
```

Make your changes (incorporate the test script) to the files in *UNIVERSAL-isa*, and then make a patch against the official release. First, in each package directory, do `make clean` to clean up any build-related files. Now, in the directory above both package directories, run `diff` with the **u**nified and **r**ecursive flags, to make the file readable and to pick up all of the changed files:

```
$ diff -ur UNIVERSAL-isa-0.05 UNIVERSAL-isa > isa_misbehaving.patch
```

This command will produce a patch that, when applied to the files in *UNIVERSAL-isa-0.05/*, will reproduce the changes you made to the module's test file. Simply include this patch with your bug report, and you'll give the package maintainer a huge head start on fixing the problem.

Attach the patch you created, and submit the form.

Check your email or the site periodically for the status of your bug. Obviously, if there's a fix, you will want to grab the new version quickly, but you also need to see if the author has rejected your bug. If so, research the issue more to determine whether the issue is truly where you thought it was, or if you need to debug your own code further.

Hacking the Hack

You can do more than merely submitting a bug report. With a well-written test case in hand, it's not as daunting a task to fix the bug yourself. Along with the patch that adds your unit tests, you could even submit a patch against the entire package source. The package tests, including the one you added, will verify that the code change is correct, so the maintainer just has to review the changes and apply them.

As it turns out, the bug is that the particular version of `UNIVERSAL::isa` called the method `UNIVERSAL::can( )` as a function, not a method. Oops.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Regardless of whether you provide a fix to the package maintainer, submitting a good bug report with effective unit tests adds value to CPAN for all its users.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006