Practical Perl Tools

Warning! Warning! Danger, Will Robinson!

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

In late January, Dick Tufeld, the voice talent behind The Robot in *Lost in Space* (and many other fine shows, *Thundarr the Barbarian* notwithstanding) died at age 85. I thought I would use the immortal phrase he spoke from that TV show as inspiration for this issue's column and dedicate it to Tufeld's memory. For this column, we're going to look at a number of ways you can (and perhaps should) handle warnings and errors within your Perl programs.

When approaching a Perl question, my first inclination is to review what is available in the language directly and what ships with the distribution (i.e., is "in core") before looking elsewhere for solutions. With that in mind, let's start with the two built-in functions for signaling that an error has occurred:

```
warn()
die()
```

The former lets you print an error message while letting the program continue, and the latter does the same but under most circumstances (more on this later) terminates the program run. Both take as an argument a list used for printing the error message. In most cases, that list contains a simple scalar, as in:

```
open my $FILE, '<', 'ookla' or die "Couldn't open the file ookla:$!";
open my $FILE, '<', 'ookla' or die "Couldn't open the file ookla:$!\n";
```

In both of these examples, I've used the common convention of including the $! variable in the error message. That variable gets set by Perl (to quote the docs) with "the current value of the C 'errno' variable, or in other words, if a system or library call fails, it sets this variable." A slightly more subtle difference between the two examples is whether the error message passed to die() or warn() ends in a newline ("\n"). If it does, the message is printed as written. If it doesn't, Perl appends the script name and line number where the error was encountered in your program to the message:

```
"Couldn't open the file ookla: no such file at column.pl line 1920"
```

The Perl documentation for die() recommends adding ", stopped" to your message if you plan to leave off the newline. The error message then would be:

```
"Couldn't open the file ookla: no such file, stopped at column.pl line 1920"
```

For most of your small-scale programming tasks, warn() and die() will do what you need. But when you start to write longer, more compartmentalized programs, you may find at least one of their standard properties may be less useful to you. Let

me show you a small example of what I mean. Let's say I'm writing a program that consists of a module I wrote and a script that calls the module. Here's the super simple module:

```
# module.pm
sub feel_better {

        open my $DATA, '<', 'meditation_of_the_day' or
          die "Can't open my data file: $!";

        print while (<$DATA>);

        close($DATA) or die "Unable to close data file:$!";
}
1;
```

and an equally simple script to call it:

```
# caller.pl
use module;

feel_better();
```

Let's run caller.pl without the data file module.pm needs. When we do so, it complains thusly:

```
Can't open my data file: No such file or directory at module.pm line 3.
```

If we were trying to debug this situation, we'd be a little stuck. We'd know that something in the module included by caller.pl reported an error, but we have no indication just where in caller.pl that module had been invoked. What if caller.pl was much longer and it called feel_better() in 20 different places in the code? How would we know which call failed? This is where the Carp module comes into play. Carp is a module shipped with Perl since version 5.00 was first released. It provides the following alternative error message routines:

```
carp()
croak()
confess()
cluck()
```

The first two, carp() and croak(), work just like warn() and die(), respectively, but instead of adding location information to the message when called without a newline based on where the error was encountered, they add it based on where the erroring code was called. So, if I change module.pm to read:

```
use Carp;
sub feel_better {

        open my $DATA, '<', 'meditation_of_the_day' or
          croak "Can't open my data file: $!";
...
```

the error message it throws in the absence of a data file is:

```
Can't open my data file: No such file or directory at module.pm line 4
        main::feel_better() called at caller.pl line 5
```

Now we know not only where the error occurred, but also have an idea of just which code got us to that point. If we wanted even more information, let's say because

we are trying to track down a situation where our program called code that called something else, we could use confess() and cluck(). The confess() call is like die()/croak() except that it will produce a full stack trace as it shuffles off this mortal coil. cluck() does a similar thing but, like warn(), it lets the program continue to run.

Before we move on to the next section, I think it is worth mentioning a number of riffs on the Croak module that are available. Most of these take the idea in a direction that makes them even more useful. For example, there are a number of modules that redirect the output of carp() in some way. The first is one I've used to great effect in the past: CGI::Carp. CGI::Carp modifies the croak()/confess()/carp() trio so that the Web server's error logs will contain useful error messages from your script. It also has a special subroutine used like this:

```
use CGI::Carp qw(fatalsToBrowser);
die "Can't open my data file: $!";
```

This subroutine will send that error message to the browser of the person encountering the message instead of sending it to the error log. The fatalsToBrowser subroutine handles all the magic necessary to make sure HTTP headers are sent before the error message. There is a related subroutine available in CGI::Carp called warningsToBrowser which will let you send the output of any warn() calls to the browser as HTML comments (so they are not seen by the user, but are still available for perusal if desired). Both of these routines are really useful while debugging a Web application during the development process.

Another Carp redirection module that tries even harder to let you direct error messages in a useful fashion is Carp::Notify. Carp::Notify can actually email you the error messages. It will also let you designate certain global variables in your program as "storable," meaning you want it to tell you the value of all of those variables when it is reporting an error. It is very helpful to have the runtime context of an error at hand when examining an error report.

If you like that last idea, you are really going to like Carp::REPL. Carp::REPL will modify your die() (and, optionally, warn()) calls so that when you call die(), instead of having the program quit, you will find yourself in an interactive Perl session. In this session you can poke and prod at the current state of the program to determine just where and how things went awry.

Death Be Proud?

Some people prefer a programming style where Perl functions and their connected system calls automatically throw errors if they fail. Instead of having to write:

```
open my $DATA, '<', 'meditation_of_the_day' or
       die "Can't open my data file: $!";
 ...
close($DATA) or die "Unable to close data file:$!";
```

they prefer to write:

```
use autodie qw(open close);
open my $DATA, '<', 'meditation_of_the_day';
 ...
close($DATA);
```

safe in the notion that if open() or close() calls fail in some way, that will cause the program to stop. The plus of this approach is your code looks a little cleaner because it isn't littered with tests for success/failure every step of the way. The autodie distribution also contains a Fatal module, which is how people pulled this trick off before autodie came along. It is worthwhile reading the docs for both modules so you will be aware of their individual gotchas.

This isn't my preferred programming style, but there is one context where I do make use of a similar behavior: DBI programming. If you open a connection to a database as follows:

```
$dbh = DBI->connect("DBI:mysql:$database",
                    $username, $pw,{RaiseError => 1});
```

DBI will automatically call die() if the connect() or other subsequent DBI calls return an error. There's an equivalent PrintError parameter that performs a warn() instead of a die() if that is more to your liking.

Death Be Not Proud

The flip side of the previous topic is that sometimes when your program encounters a "fatal" error, à la die(), you may have a different idea for what the program should do at that moment. If you are able to write code that can recover from an error like this, you need a way to catch the error and proceed from that point with your recovery. The usual way to do this is to wrap the operation that could potentially fail in an eval()/eval{} block, as in this example from the DBI docs:

```
eval {

        ...
        $sth->execute();
        ...
};
if ($@) {
        # $sth->err and $DBI::err will be true if error was from DBI
        warn $@; # print the error
}
```

If the execute() fails, the eval() will trap its failure and set $@ accordingly. Eval() also gets used like this when you want to test for a potentially fatal condition, e.g., if a module you plan to use isn't available:

```
eval ('use Mondok;');
warn "Mondok module not available on this machine, skipping..." if ($@);
```

But eval() has a few issues (especially before Perl version 5.14, where a key one was addressed). These issues mostly surround $@, which can be cleared, clobbered, and generally messed with in ways that don't necessarily make it a reliable semaphore. The easiest way to get around these problems is to use a module called Try::Tiny (whose docs contain a lovely litany of eval() complaints). Try::Tiny lets you write code of this form:

```
try { something } catch { the results } finally { perform some clean up }
```

Real Perl code using Try::Tiny would look like this:

```
try { this_might_die() }
catch { warn "had a problem: $_";}
finally { $error_count++ if ($@); }
```

If you like the try-catch-finally construct—perhaps you miss it from other programming languages—you might want to check out Try::Tiny's more powerful sibling, TryCatch. TryCatch has a larger list of dependencies, but it lets you do things like put "return()" in your catch blocks, something Try::Tiny by itself cannot do. To get that specific functionality from Try::Tiny you would need to add another module, Error::Return.

But Maybe What You Really Want Is an Object

We're almost out of time for this column, but I wanted to at least mention one other approach to handling errors that you may wish to explore if you are writing larger and more extensive OOP-based programs. There's a good argument to be made for passing around exception objects instead of the simple scalar error messages we've seen throughout this column. With an exception object, you can define a persistent interface for reporting back more detail on errors. This means that instead of having to parse:

```
"Ran out of memory: 20k"
```

and then rewrite how you parse it when you decide the message should be:

```
"FATAL: memory exceeded max allocated by 20k"
```

you might want to use an object that can be queried for the error message and the amount of memory separately. Here's some example code:

```
use Exception::Class ( 'ColumnException' => { fields => ['memory_used'] } );

# we wrap this in an eval because throw() does a die() with the given object
eval {ColumnException->throw(
       error=> 'FATAL: memory exceeded max allocated',
       memory_used => '20000'
)};

my $e  = Exception::Class->caught('ColumnException');
print $e->error,": ";
print $e->memory_used,"\n";
```

There are a few modules that help make creating exception objects pretty painless. I'd recommend you look at Exception::Class (used in the previous example) or Exception::Base. If you decide to use the latter, you may want to check out Exception::Died, because it will automatically hook all of the die() calls in your program and cause them to return exception objects by default.

With that, we have to bring this column to an end. Thank you Mr. Tufeld for making my TV-watching days richer with your voice. Take care, and I'll see you next time.