The Perl Hardware Store

Tools You Didn't Know You Needed

Mark-Jason Dominus
Plover Systems
mjd-tpc-hardware@plover.com

19 August, 1998

1 Teflon Tape

A few months ago, I had a leaky shower head, so I went to the hardware store to get some putty or caulk or something to fix the leak. But when they found out what I wanted it for, they sold me teflon pipe thread tape instead. The tape fills up the gap between the head and the pipe just fine, it's waterproof, and it's very slippery, so it's easy to put the shower head on and take it off again. Also you can remove it easily, even years later. I had never heard of teflon pipe thread tape, but it was just what I needed, and much better than the caulk would have been.

I get two reactions from people I tell this story to. Some people say "Teflon pipe thread tape! Wow, I never heard of that before; I'll have to remember it." And some people interrupt me right at the beginning and say "You should use teflon tape for that, you know."

This talk is about teflon pipe thread tape. It's also about hose clamps, self-chalking plumb lines, mending plates, WD-40 in a spray can, and a whole bunch of other stuff. None of this stuff is specially magical; that's the point. Each of these tools will make half the people in the audience say "Well, of course! Everyone knows about that." But the other half will say "Wow! I never heard of that before, but now I don't know how I ever got along without it."

2 Schwartzian Transform

This first tool is already very well-known. But it's so useful that it's worth spending a few minutes on even if only a few people haven't seen it already. It's the Schwartzian transform, named after Randal Schwartz, who popularized it.

The problem it solves is when you want to sort a list of items, not by some apparent feature of the items, but by some hidden feature. For example, you have a list of the files in the current directory, and you want to sort them by modification date instead of by name.

The general idea is this: Construct a data structure that has both kinds of information, names and dates in it, sort by the appropriate one, and then throw it away again. Here's how it looks in Perl:

```
@names = readdir D;

@names_and_dates =
  map { { NAME => $_, DATE => -M $_ } }
  @names;

@sorted_names_and_dates =
  sort { $b->{DATE} <=> $a->{DATE} }
  @names_and_dates;

@sorted_names =
  map { $_->{NAME} }
  @sorted_names_and_dates;
```

I used hashes here, to make it clearer what was going on, but normally you use lists for efficiency. Perl hackers do this whole thing in one line and save some variables:

```
@sorted_names =
  map { $_->[0] }
  sort { $b->[1] <=> $a->[1] }
  map { [ $_, -M $_ ] }
  readdir D;
```

It's a little funny that I can be talking about this at all, because it's so common in Unix shell programming that it doesn't even have a name:

```
# Sort file names by file size
ls -l | sort -n +4 | awk '{print $NF}'

# Sort output of SOMETHING
# from most frequent to least
SOMETHING | uniq -c
  | sort -nr | awk '{$1=""; print}'
```

Of course, the Perl versions are better because you don't have to worry about mixing up the columns.

3 Manual Exporting

Most people know that there's something called the `Exporter`. Some people know what it does: It arranges for functions defined in one package to be *exported*, so they appear as if they'd been defined in some other package even though they haven't. A few people even know how this works.

The exporter is big and heavy, and often you don't want it. Knowing how to import stuff without it can be very useful.

Simplest Example: Import a Function from Another Package

Suppose you have a program whose code is divided among several modules. You want them all to use the same logging function. You could duplicate the code of the function into every package file, but that's asking for trouble. Or you could define the function once, in package `main`, and then use

```
package Foo;

main::LOG("Your shoelace is bleeding!");
```

That's a little ugly, and it's unnecessary. What you can do instead is *import* the `LOG` function into the modules where you want to use it. Doing that is a one-liner:

```
package Foo;

*LOG = \&main::LOG;    # Import

LOG("Look over there!  A dancing cow!");
# (Calls main::LOG)
```

Simple! Convenient!

Writing a Module that Exports a Function

Perhaps you have written a module whose only purpose is to define a new function and export it into the caller's namespace. You can use `Exporter` for this; sometimes it's just the thing. But `Exporter` is rather heavy; it has a lot of features that you don't really want, like `EXPORT_OK` and `EXPORT_TAGS` and I don't know what else. It's also five hundred lines long.

Instead of using `Exporter`, consider doing this:

```
package Foo;

# Always export 'foo':
sub import {
  # Get name of calling package
  my $caller = caller;

  *{$caller . '::foo'} = \&foo;
}
```


This bears careful study, because it's so useful and so simple. The `caller` function returns the name of the package that said `use Foo;`. We put that name into `$caller`; let's suppose it was package `main`. Then we construct the name that we want the function to have in the calling package; that's `main::foo`. Then we do the glob assignment just like before.

If we create a subclass of `Foo` called `Subclass` and let it inherit the `import` method, it'll still export `Foo::foo` regardless of whether or not there's an overriding function `Subclass::foo`. It's easy to fix that if we want to:

```
# If you subclass this package, it will
# export the overriden 'foo' from the
# subclass, if there is one.


sub import {
  my $package = shift;
  my $caller = caller;
  *{$caller . '::foo'}
    = $package->can('foo');
}
```

When you say

```
use Subclass ...;
```

that turns into a call to

```
import('Subclass', ...);
```

So an inherited `import` method can know what package it is acting on behalf of by examining its first argument, here called `$package`. This `import` uses `can` to ask Perl where there's a (possibly inherited) `foo` method defined for the calling package, and if there is one, `can` hands back a reference to that function, which we then export.

Constants

Now we know wnough to see how the `use constant` pragma works. Here it is, somewhat stripped down:

```
sub import {
  my $name = shift
    or return;
  my $value = shift;


  my $caller = caller;
  *{$caller . "::$name"}
    = sub () { $value };
}
```

The real module has error checking, and also handles list constants, but it's just the same. Check it out.

Exporting Variables

You don't have to export functions; you can export anything. THen you have one variable with two names in two different packages.

Here's a simple example: When a database call causes an error in the `DBI` module, `DBI` puts the error message into the string `$DBI::errstr`; then you can examine it, similar to the way you use `$!`. Now, maybe you wish you had a more convenient name for it.

No problem:

```
$DBerr = \$DBI::errstr
```

Now you can use `$DBerr` instead.

Exporting Variables, Part II

Here's a great trick:

```
package Evaluator;

sub import {
  my %hash;
  my $name = shift || 'eval';
  tie %hash => Evaluator;
  my $caller = caller;
  *{$caller . '::' . $name} = \%hash;
  1;
}
```



```
sub TIEHASH {
  bless {} => Evaluator;
}

sub FETCH {
  $_[0];
}
```

We created an anonymous hash, *tied* it, and exported the *tied* hash into the calling package. How do we use this?

```
use Evaluator;
print "6 * 7 + 5 = $eval{6 * 7 + 5}\n";

# Prints ''6 * 7 + 5 = 47''
```

or

```
use Evaluator ':';
print "6 * 7 + 5 = $:{6 * 7 + 5}\n";

# Prints ''6 * 7 + 5 = 47''
```

My hilarious `Interpolation` module lets you define arbitrary string interpolation semantics using this trick.

Clone the Exporter

It doesn't take five hundred lines of code to emulate the interesting part of `Exporter`. If you just want the simple features, it's trivial to do it yourself:

```
package Ziggurat;

my @EXPORTS = qw(this that other);

my %EXPORTED = map {$_ => 1} @EXPORTS;

sub import {
  my $pack = shift;
  my $caller = caller;
  foreach $func (@_) {
    croak "'$func' not exported"
      unless $EXPORTED{$func};
    *{$caller . '::' . $func} = \&$func;
  }
}
```

4 Adding a New Method to a Package

Suppose you're using someone else's module, and here's a method that you wished that it provided, but it doesn't. For example, consider the `Msql` module, which provides access to an SQL-driven database. After you make a query with the `query` method, you get back a 'statement handle' object, from which you can retrieve the results of the query, one record at a time, with `fetchhash`; typical code looks like this:

```
my $sth = $dbh->query(...);
error() unless defined $sth;

my $num_records = $sth->numrows();
for ($i=0; $i<$num_records; $i++) {
  # Get next record
  my %data = $sth->fetchhash();

  # Do something with %data
}
```

Now, you be doing many different queries, and you get tired of repeating the same error-checking code over and over, and moreover, it would be easier if there were a method that would just get *all* the records into one big list of hashes, instead of one at a time.

Everyone knows how to add a method to a class; it's called subclassing. And yet subclassing in this case seems like rather a pain; you'd have to make up a new namespace, and put it in a file, and set up `ISA`, and change all your programs to use the new module instead of `use Msql`, and you're hardly adding much to the class anyway. But the alternative seems to be to hack on the `Msql.pm` file itself, which you certainly don't want to do.

The solution is incredibly simple and lightweight. Here it is: In the main program file, put

```
sub Msql::myquery {
  my $dbh = shift;
  # Just stick the method here, dummy.
}
```

Now you can write

```
@list_of_hashes = $dbh->myquery(...);
```

just as if `myquery` were like any other method in the `Msql` package, because it *is* just like any other method in the `Msql` package.

If `Msql` does happen to have subclasses, they'll inherit your new `myquery` method too.

The only thing wrong with this is that you and the programmer in the next cubicle might both try to add incompatible `Msql::myquery` functions. But big projects have different rules than small ones, and in a small project, this lightweight tactic for class extension is a huge win.

People who live in languages that have stricter namespace boundaries than Perl can probably come up with some other reasons why this is a terrible thing to do. But you should be able to ignore them without too much trouble.

If you find yourself adding lots and lots of new methods, you should probably think about putting them into a subclass.

This isn't really an 'importing' trick, but I threw it in anyway, because it's useful.

5 Semaphore Files

Let's write a program that will either print out a file to a web browser, or update that file:

```
$FILE = '/data/hitcounter';

print "Content-type: text/html\n\n";
if (something()) {
  open F, $FILE or die ...;
  print while <F>;
} else {
  open F, "> $FILE" or die ...;
  my $data = qx{some command};
  print F $data;
}

close F;
```

There's a little problem here called a race condition. If one process is reading the file while the other one is writing it, the reading process might get garbage. Worse, if two processes try to update the file at the same time, the file will probably get mangled. For a hit counter, nobody really cares, but if it's the file that records your bank balance, you might get a little upset.

The solution, which everyone knows, is to use *file locking*:

```
if (something()) {
  open F, $FILE or die ...;
  flock F, LOCK_SH;
  print while <F>;
} else {
  open F, "> $FILE" or die ...;
  flock F, LOCK_EX;
  my $data = qx{some command};
  print F $data;
}
```

Unfortunately, this is wrong. The 'writing' branch has already clobbered the contents of the file before it acquired the lock. If a process was reading, it might read garbage.

Let's see, we can fix that—we'll lock the file before we open it...No, that doesn't work, because we can only lock an open filehandle, not a file.

Well, we could have the writer open the file in append mode, lock the file, then seek back to the beginning, overwrite the contents, and truncate the file?

Blecch. Here's a simpler solution: Use a semaphore file which is unrelated to the data file, and lock the semaphore file instead:

```
$FILE = '/data/hitcounter';
$SEMAPHORE = $FILE . '.lck';


print "Content-type: text/html\n\n";


open S, "> $SEMAPHORE" or die ...;


if (something()) {
  flock S, LOCK_SH;
  open F, $FILE or die ...;
  print while <F>;
} else {
  flock S, LOCK_EX;
  open F, "> $FILE" or die ...;
  my $data = qx{some command};
  print F $data;
}


close F;  close S;
```

All sorts of problems just go away when the semaphore file is unrelated to the data file. For example, you can use one semaphore to lock an entire directoryful of data if you want to. On some systems, you can't get an exclusive lock on a file you've opened only for reading; the semaphore file approach solves that problem also: You can open the semaphore for writing, and exclusive-lock it, regardless of whether you opened the data file for writing.

You can also arrange to have the process leave its PID in the semaphore file so you know who has the resource locked, and maybe some other useful information. Can't do that when you lock the file with the real data.

It's dumb to use the same file for two unrelated purposes. Just do it this way; you'll be glad.

6 Memoizing

A *pure* function is one with no side effects whose return value depends only on the values of its arguments. For example:

```
sub factorial {
  my $n = shift;
  $n == 0 ? 1 : $n * factorial($n-1);
}
```

You can sometimes make pure functions faster by *memoizing* them. A memoized function caches its return values; then if you call it with the same arguments as before, it just looks up the answer in the cache:

```
{ my @fact = (1);
    sub factorial {
    my $n = shift;


    return $fact[$n]
      if defined $fact[$n];


    $fact[$n] =
      ($n == 0 ? 1
               : $n * factorial($n-1));
  }
}
```

In this case that isn't so impressive. But for some applications, memoizing can be a huge win.

Highly Recursive Functions

```
# Compute nth Fibonacci number
sub fib {
  my $n = shift;
  if ($n < 2) { $n }
  else        { fib($n-1) + fib($n-2) }
}
```

This function is very slow. It takes quite a while to compute `fib(20)`, because it first wants to compute `fib(19)` and `fib(18)`, and add the results. But to compute `fib(19)`, it first has to compute `fib(18)` and `fib(17)`, and then it comes back and computes `fib(18)` all over again even though the answer is the same. And both of the times that it wants to compute `fib(18)`, it has to compute `fib(17)` from scratch, and then it has to do it again each time it wants to compute `fib(19)`. This function does so much recomputing of old results that it takes a really long time to run—`fib(20)` makes about 22,000 recursive calls, to compute and recompute things that it already computed.

Memoizing here is a big win; you reduce those 22,000 recursive calls to about 40. If you ask for `fib(20)` again, it doesn't make any recursive calls at all.

Automatic Memoization

I like memoization a lot; it's useful for all kinds of things. Like all tools, it's better when you don't have to think about it, so I buried it in a module. If you have the module, you can say

```
use Memoize;
memoize 'fibo';
```

and it'll automcatically memoize the `fibo` function. Very handy, and available from CPAN.

Functions that Take a Long Time to Compute

If the function will take a long time, you can cache the results in a file. Once the results database is populated, the function will run almost instantaneously.

You can populate the database overnight, as a batch job, and then when you come back in the morning you've turned a slow function into a fast function.

Profiling Execution Speed

If your program is too slow, and you want to speed it up, the first thing you should do is profile it, to find out which functions are taking the most time. Otherwise you might waste effort trying to speed up a function that only contributes to 1% of the program's entire run time. Profiling can be a pain, and the Perl profiler isn't as well-developed as you might like it to be.

You can use memoizing as an alternative, and it's sometimes preferable. Suppose you have a function `f` which you think is a possible candidate to be rewritten to be faster. Run your program three times: Once with no memoizing, once with `f` memoized (this will populate the cache), and once with `f` memoized and the cache already populated. This last run will simulate the speed of the program with `f`'s contributions removed. If the run time with the cache populated is 98% of the run time with no memoizing, then no possible rewriting of `f` can speed up the program more than about 2%—so you'll know to look elsewhere.

'Orcish Maneuver'

Memoizing is very useful for sort comparison functions, which tend to get called over and over again with the same arguments.

Suppose you have a bunch of strings in the form `"May 14, 1997"` and you want to sort them into chronological order. The obvious way is:

```
%m2n =
   ( jan => 0, feb => 1,  mar => 2,
     apr => 3, may => 4,  jun => 5,
     jul => 6, aug => 7,  sep => 8,
     oct => 9, nov => 10,dec => 11, );

sub compare_dates {
  my ($am, $ad, $ay) =
    ($a =~ /(\w{3}) (\d+), (\d+)/);

  my ($bm, $bd, $by) =
    ($b =~ /(\w{3}) (\d+), (\d+)/);

             $ay  <=>         $by
  || $m2n{lc $am} <=> $m2n{lc $bm}
  ||         $ad  <=>         $bd;
}
```

Now, suppose you have a list of 1,000 of these strings. You're going to make about 8,700 calls to `compare_dates`, so about 17,500 splits, and that means that the typical date string was split 17.5 times, of which the last 16.5 were a complete waste.

One way out of this is to use the Schwartzian Transform. Another is to define an auxiliary function that turns a date into a number, and then memoize it:

```
use Memoize;

sub compare_dates {
  to_number($a) <=> to_number($b);
}

# Convert "Apr 2, 1969" to "196990402"
sub to_number {
  my ($m, $d, $y) =
    ($_[0] =~ /(\w{3}) (\d+), (\d+)/);

  sprintf("%04d%02d%02d",
          $y, $m2n{$m}, $d);
}

memoize 'to_number';
```

Now you only do 1,000 splits, which is a big improvement.

In sort comparators, you often need more speed, and the slow part of this comparator is the two calls to `to_number`. We've replaced 17,500 calls to `split` with 17,500 calls to `to_number`, which is an improvement, but not as much of an improvement as we'd like. So instead of using the `Memoize` module, we can just inline the memoization, like this:

```
{ my %cache;

  sub compare_dates {
    ($cache{$a} ||= to_number($a)) <=>
    ($cache{$b} ||= to_number($b))
  }
}
```

Now we have the best of both worlds. If the numeric version is already in the cache, we get it immediately; otherwise we compute it and put it in the cache. Result: Exactly 1,000 calls to `to_number`.

Joseph Hall, author of *Effective Perl Programming*, dubbed this the 'Orcish Maneuver', because the notable features of this approach are the `||` and the cache.

Dynamic Programming

You might have heard of 'dynamic programming'; it's a technique where you break down a problem into smaller subproblems, solve them separately, and then build up the solutions into a solution to the big problem. Merge sort is a good example. For a more interesting example, we'll look at the *partition problem*.

In the partition problem, you have list of numbers. You want to divide the list into two groups so that the sum of the numbers in each group is the same.

If you learned about dynamic programming in school (assuming you went to school) you probably spent a lot of time working out the details of how to represent and locate the solutions to the subproblems. But you can think of memoization as an automatic dynamic programming technique.

We're going to solve the partition problem by writing a recursive function. Suppose the list has five elements, whose sum is 30. Then what we really need to do is find some subset of those elements whose sum is 15. If we can do that, or prove there is no such subset, we've solved the partition problem.

Now, suppose the first element is an 8. This 8 might be part of the subset we want to find, or it might not. If it is part of the subset, then the remaining elements of the subset must add up to 7; if it isn't, then they add up to 15. So throw away the 8, and recursively inquire if some subset of the remaining elements can be made to add up to 7 or 15.

Repeat this until you've thrown away all but one of the elements, at which point the solution is trivial; it's either equal to the target sum, or it isn't.

Our recursive function `T` will take a list of numbers and a target sum, and it'll try to see whether there's a subset of the numbers that adds up to the target sum. Here's the code:

```
# Take a list of numbers and a target
# sum.  return a sublist that add up to
# the target, if possible, or undef
# otherwise.
```

```
sub T {
  my ($list, $target) = @_;
  my $answer;

  if (@$list == 0) {
    return ($target == 0) ? [] : undef
  }

  my $first = shift @$list;

  $solution = T($list, $target);
  unless (defined $solution) {
    $solution = T($list, $target - $first);
    if (defined($solution)) {
      unshift @$solution, $first;
    }
  }

  # restore @$list
  unshift @$list, $first;

  return $solution;
}
```

Now let's ask it if it can find a way to split the elements of (8,2,7,3,10) into two lists each of which adds up to 15. The call looks like this:

```
T([8,2,7,3,10], 15)
```

And sure enough, the function returns (2,3,10).

Actually this function is too slow to use for large problems; it takes exponential time just like the Fibonacci number function did. And again, we can solve the problem by memoizing. We just add two lines to the top of the program:

```
use Memoize;
memoize 'T';
```

Suddenly it's an efficient solution instead of a slow one.

Here's a driver function that you can use for playing with it:

```
sub partition {
  my ($answer, $sum);
  map { $sum += $_ /2 } @_;

  $answer = T(\@_, $sum);

  if (defined $answer) {
    local $" = ' + ';
    print "@$answer = $sum\n";
  } else {
    print "No solution exists.\n";
  }
}
```

Try partition(8,2,7,3,10).

7 Ouch

I'm sure I'm out of time by now. I wanted to show you how to roll your own semantics for OO programming if you don't like the way Perl does it. Be sure to lobby the conference folks to invite me back if you want to hear about it.