Table of Contents

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

Chapter 3. Data Munging

Hacks 19-27

Perl has always been in love with data. No matter where you find it, Perl happily processes and extracts and reports on files, databases, web pages, spreadsheets, other programs, and anything that produces data. Perl's so happy to do this that it even overlooks brute-force, rough manipulations. Hey, pragmatism works!

Perl can be gentle, too. A little subtlety, a little style and finesse, and you can write maintainable, easy-to-understand code that's just as powerful as the wild-eyed forge-ahead-at-all-costs just-do-the-job code. Why? It's often faster and more correct—as well as more secure, more powerful, and shorter.

Sure, slinging data between sources sounds about as glamorous as slinging hash at the local diner, but it doesn't have to be that way. Here are several ideas to munge that yummy data with all of the elegance and style and power and clarity that you know you have.

Hack 19. Treat a File As an Array

Pretend a big stream of data on disk is a nice, malleable Perl data structure.

One of the big disappointments in programming is realizing that, although you can think of a text file as a long list of properly terminated lines, to the computer, it's just a big blob of ones and zeroes. If all you need to do is read the lines of a file and process them in order, you're fine. If you have a big file that you can't load into memory and can't process each line in order...well, good luck.

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

Fortunately, Mark Jason Dominus's `Tie::File` module exists, and is even in the core as of Perl 5.8.0. What good is it?

The Hack

Imagine you have a million-line CSV file of inventory data from a customer that's just not quite right. You can't import it into a spreadsheet, because that's too much data. You need to do some processing, inserting some lines and rearranging others. Importing the data into a little SQLite database won't work either because trying to get the queries right is too troublesome.

`Tie::File` won't help you write the rules for transforming lines, but it will take the pain out of manipulating the lines of a file. Just:

```
use Tie::File;

tie my @csv_lines, 'Tie::File', 'big_file.csv'
    or die "Cannot open big_file.csv: !$\\n";
```

Running the Hack

Suppose that your big CSV file contains a list of products and operations. That is, each line is either a list of product data (product id, name, price, supplier, et cetera) or some operation to perform on the previous *n* products. Operations take the form `opname:number`. Obviously the file would be easier to process if the operations appeared *before* the data on which to operate, but you can't always change customer data formats to something sane. In fact, this might be the easiest way to clean the data for other processes.

`Tie::File` makes this almost trivial:

```
for my $i ( 0 .. $#csv_lines )
{
    next unless my ($op, $num) = $csv_lines[ $i ] =~ /^(\\w+):(\\d+)/;
    next unless my $op_sub     = __PACKAGE__->can( 'op_' . $op );

    my $start                  = $i - $num;
    my $end                    = $i - 1;
    my @lines                  = @csv_lines[ $start .. $end ];
    my @newlines               = $op_sub->( @lines );

    splice @csv_lines, $start, $num + 1, @newlines;
}
```


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

Okay, there *is* a bit of cleverness in finding the right range of lines to modify, but consider how much trickier the code would have to be to do this *while* looping through the file a line at a time.

Of course, you can use all of the standard array manipulation operations (`push`, `pop`, `shift`, `unshift`, and `splice`) as necessary.

Hack 20. Read Files Backwards

Process the most recent lines of a file first.

Perl's position in system administration is stable and secure, due in no small part to its fast and flexible text-processing abilities. If you need to slice and dice log files, monitor services, and send out messages, you *could* glue together the perfect combination of shell and command-line utilities, or you could have Perl do it.

Of course, Perl is a general-purpose language and doesn't always provide every tool you might need by default. For example, if you find yourself processing system logs often, you might wish for a way to read files in reverse order, the most recent line first. Sure, you *could* slurp all of the lines into an array and read the last one, but on a busy system with lots of huge logs, that can be slow and memory-consuming.

Fortunately, there's more than one way to do it.

The Hack

Yes, you *could* look up `perldoc -F -X` and find a file's size and read backwards until you find the appropriate newline and then read forward...but just install `File::ReadBackwards` from the CPAN instead.

Suppose you have a server process that continually writes its status to a file. You only care about its current status (at least for now), not its historical data. If its status is `up`, everything is happy. If its status is `down`, you need to panic and notify everyone, especially if it's 3 a.m. on Boxing Day.

Simulate the program that writes its logs with:

```
#!/usr/bin/perl

use strict;
use warnings;

use Time::HiRes 'sleep';

local $| = 1;

for ( 1 .. 10000 )
{
    my $status = $_ % 10 ? 'up' : 'down';
    print "$status\\n";
    sleep( 0.1 );
}
```

This program writes a status message to STDOUT ten times a second; nine of those are `up` and the last is `down`. Run it and redirect it to a file such as *status.log*. In `bash`, this is:

```
$ perl write_fake_log.pl > status.log &
```

Running the Hack

With *status.log* continually growing with newer information, finding the most recent status is easy with `File::ReadBackwards`:

```
use FileReadBackwards;

my $bw = File::ReadBackwards->new( 'status.log' )
    or die "Cannot read 'status.log': $!\\n";

exit( 0 ) if $bw->readline( ) =~ /up/;

# panic( ) ...
```

The program is straightforward. Create a `File::ReadBackwards` object by passing the name of the file you want to read. Then, every time you call `readline( )` on the object, you'll receive the previous line of the file, starting with the last line and working backwards to the first line.

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

Hacking the Hack

Note that the current version of the module (1.04) does not `flock` the file it reads, so you may read a partial line. Also, you may get a partial line, depending on how large your filesystem's buffers are and how much the process has written to it. If either is important to you, the source code is short, full of good comments, and easy to modify—but if you just need the last *n* lines of a file, this is the easy way.

Hack 21. Use Any Spreadsheet As a Data Source

Make your data analysis independent of the spreadsheet program.

Spreadsheets are useful for holding structured data, usually based on columns and rows. Most often part of the data is calculated data from other cells in the same spreadsheet.

If you want to work with that data, you face the problem of too many standards and programs. Writing a script that has to read the data from the spreadsheet is more writing an interface to the spreadsheet than actually working with the interesting data.

Accessing Cell Data

The `Spreadsheet::Read` module gives you a single interface to the data of most spreadsheet formats available, hiding all the troublesome work that deals with the parsers and the portability stuff, yet being flexible enough to get to the guts of the spreadsheet.

It's easy to use:

```
use Spreadsheet::Read;

my $ref  = ReadData( 'test.xls' );
my $fval = $ref->[1]{A3};
my $uval = $ref->[1]{cell}[1][3];
```

Here `$ref` is a reference to a structure that represents the *data* from the spreadsheet (*test.xls*). The reference points to a list (the worksheets) of hashes (the data).

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

Every cell has two representations: either access it by its name (`A3`), in which case the interface gives you the formatted value, or the `cell` hash, in which case you get the unformatted value of the cell.

Do I need Spreadsheet::Read for that?

No you don't, but it makes life easier. Setting aside all the good things of the various user interfaces for the available spreadsheets (95% probably Excel or OpenOffice.org), coding access to the cell data in the available native parsers is not always as easy as it should be. These interfaces try to give you full control, but you have no easy way to access the data.

Some examples for native equivalences of the previous code snippet are:

- Microsoft Excel
- OpenOffice.org
- Comma Separated Values

They all show you the contents of cell `A3`, where you can interpret the CSV file as a collection of rows (the lines) and columns (the fields).

`Spreadsheet::Read` gives the same interface to all of these, but uses the native parser in the background. The only thing you have to alter if you change spreadsheet formats is the `ReadData( )` call:

```
my $ref = ReadData( 'test.xls' );

# or
my $ref = ReadData( 'test.sxc' );

# or
my $ref = ReadData( 'test.csv' );
```

which will make your code much more readable, maintainable, and portable. Your code won't depend on the spreadsheet format used by the people shipping you the data.

Accessing a data column

Accessing a single field is good if you know the field you need to access, but quite often, your script has to analyze the data (structure) itself. For that you need a full set of data. Spreadsheet user interfaces always refer to the data location as a `(column, row)` pair, where (Perl) programmers more often use the `(row, column)` way of indexing. Perl starts indexing at

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

0, where spreadsheets usually start with 1. Spreadsheet::Read starts with 1 for the data and uses the zeroth field for internal control data. To fetch a complete column:

```
# Fetch me column "B"
my @colB = @{ $ref->[1]{cell}[2] };
shift @colB;
```

or:

```
my @colB = @{$ref->[1]{cell}[2]}[1..$#{$ref->[1]{cell}[2]}];
```

Accessing a row of data

Likewise for fetching a complete row:

```
# Fetch me row 4
my @row4 = map { $ref->[1]{cell}[$_][4] } 1..$ref->[1]{maxcol};
```

Using programmer-style indexing

If you need to go over and through the complete set of data and prefer to have the data in a list of rows, instead of a list of columns, indexed from 0 not 1, Spreadsheet::Read offers a function to convert that for you:

```
use Spreadsheet::Read qw( rows );

# Get all data in a row oriented list
my @rows = rows( $ref->[1] );

# A3 is now in $rows[2][0]
```

Showing all data in a spreadsheet

Want to show all of the data in a spreadsheet?

```
use Spreadsheet::Read;

my $file        = 'test.xls';
my $spreadsheet = ReadData( $file )         or die "Cannot read $file\\n";
my $sheet_count = $spreadsheet->[0]{sheets} or die "No sheets in $file\\n";

for my $sheet_index (1 .. $sheet_count)
{
    # Skip empty worksheets
```


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

```
    my $sheet = $spreadsheet->[$sheet_index] or next;

    printf( "%s - %02d: [ %-12s ] %3d Cols, %5d Rows\\n", $file,
        $sheet_index, $sheet->{label}, $sheet->{maxcol}, $sheet->{maxrow} );

    for my $row ( 1 .. $sheet->{maxrow} )
    {
        print join "\\t" => map {
        $sheet->{cell}[$_][$row] // "-" } 1 .. $sheet->{maxcol};
        print "\\n";
    }
}
```

The output will be something like:

```
test.xls - 01: [ Sheet1       ]   4 Cols,     4 Rows
A1      B1      -       D1
A2      B2      -       -
A3      -       C3      D3
A4      B4      C4      -
test.xls - 02: [ Second Sheet ]   5 Cols,     3 Rows
x       -       x       -       x
-       x       -       x
x               x               x
```

Note that the example uses the `defined-or` operator (`//`) from Perl 6. This is available as a patch for Perl 5.8.x and will be available in Perl 5.10.

Empty cells are often `undef` values, which is not the same as an empty string `""`. If you use the above code with `strict` and `warnings`, there will be a warning for every empty cell if you do not use the defined-or. Showing empty fields as – is more visibly attractive than using whitespace.

Written in more portable code, this is equivalent to:

```
print join "\\t" => map
{
    my $val = $sheet->{cell}[$_][$row];
    defined $val ? $val : "-";
} 1 .. $sheet->{maxcol};
```

How It Works

`Spreadsheet::Read` does no parsing of the spreadsheets itself, instead using the native parsers to do the hard work. For Microsoft Excel, it uses `Spreadsheet::ParseExcel`, for OpenOffice.org, `Spreadsheet::ReadSXC`, and for CSV, `Text::CSV_XS`.

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

Using `Spreadsheet::Read`, you do not have to worry about spreadsheet internal formats, or the way the native parser presents the data to the programmer. The interface is the same and is independent of the spreadsheet you use. If you need to get to the guts for anything this interface does not (yet) support, you can always fall back to the real parser, because without it, `Spreadsheet::Read` does not work anyway.

`Spreadsheet::Read` tries to achieve a set of commonly supported features of all of the parsers it can use and aims to extend that in the future to make the use of the native parsers unnecessary (such as color attributes, display formats, font face and sizes, and character encoding). All native parsers support that in a different way, if they support it at all. For example, CSV does not have a defined way of identifying the character encoding of the data.

Hacking the Hack

The more spreadsheet formats this module supports, the more value it gains in portability and eventually for your script's maintainability.

Currently, the module supports a hook for `Spreadsheet::Perl`, but there is no parser support for it yet. It would probably do this module well to isolate the conversions for the different parsers in separate modules, such as `Spreadsheet::Read::Excel`, to avoid cluttering the main interface.

The module comes with one conversion script: `xlscat`, which takes a file in any of the supported spreadsheet formats and converts it to either readable ASCII or CSV. Use `xlscat -?` to see the supported options. If you have useful scripts, they may be worth bundling with this module.

The module does not die if any of the parsers is not installed, making it useful if you only use OpenOffice.org and do not yet bother with Excel (or vice versa). It is quite easy and valuable to add your own parser support and supply a patch to the author to include it in future releases.

Hack 22. Factor Out Database Code

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

Separate SQL and Perl to make your life easier.

Small scripts have a way of growing up into essential programs. Unfortunately, they don't always mature design-wise. Far too often a business-critical program starts life as a quick-and-dirty just-get-it-done script and evolves mostly by accretion, not the clear and thoughtful hand of good design.

This is especially true in programs that work with data in various formats or which embed other languages such as HTML or SQL. Fortunately, it only takes a little bit of discipline—if no small amount of work—to clean up this mess and make your code somewhat easier to maintain.

The Hack

The only trick is to remove all SQL from within the code and to isolate it in its own module. You don't have to abstract away or factor out all of the database access code or the various means by which you fetch data or bind to parameters—just untangle the Perl and non-Perl code.

Be strict. Store *every* instance of SQL in the module. For example, if you have a subroutine such as:

```
sub install_nodemethods
{
    my $dbh = shift;

    my $sth = $dbh->prepare(<<'END_SQL');
SELECT
    types.title AS class, methods.title AS method, nodemethod.code AS code
FROM
    nodemethod
LEFT JOIN
    node AS types ON types.node_id = nodemethod.supports_nodetype
END_SQL

    $sth->execute( );

    # ... do something with the data
}
```

store the SQL in the SQL module in its own subroutine:

```
package Lots::Of::SQL;

use base 'Exporter';
use vars '@EXPORT';
```

```
@EXPORT = 'select_nodemethod_attributes';

sub select_nodemethod_attributes ( )
{
    return <<'END_SQL';
    SELECT
        types.title     AS class,
        methods.title   AS method,
        nodemethod.code AS code
    FROM
        nodemethod
    LEFT JOIN
        node AS types ON types.node_id = nodemethod.supports_nodetype
    END_SQL
}
```

Running the Hack

Now call the query from the refactored original subroutine:

```
use Lots::Of::SQL;

sub install_nodemethods
{
    my $dbh = shift;

    my $sth = $dbh->prepare( select_nodemethod_attributes( ) );
    $sth->execute( );

    # ... do something with the data
}
```

Putting the empty prototype on the SQL abstraction function tells Perl that it can inline the (constant) return value whenever other code calls this function. You get the benefit of hiding all that SQL behind a readable name without paying a runtime price.

Hacking the Hack

Of course, stuffing all of that code into one potentially huge module isn't exactly the end result of refactoring—but with a bit more polish, it's a good step. Exporting all of the SQL subroutines is overkill that doesn't really balance out the niceness of being able to maintain the same SQL just once for any application that uses the database.

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

Why not export just what you need?

Consider that every operation on a table is its own exporter group, then create an exporter tag for that operation. For example, if you have the tables `users`, `stories`, and `comments`, group each *type* of SQL query into a tag:

```
package Lots::Of::SQL;

use base 'Exporter';
use vars qw( @EXPORT_OK %EXPORT_TAGS );

@EXPORT_OK = qw(
    select_user    insert_user    update_user
    select_story   insert_story   update_story
    select_comment insert_comment
    select_stories
    select_user_stories
    select_user_comments
);

%EXPORT_TAGS = (
    user    =>
    [ qw(
        select_user insert_user update_user select_user_stories
        select_user_comments
    )],
    story   =>
    [ qw(
        select_story insert_story update_story select_user_stories
        select_stories
    )],
    comment => [ qw( select_comment insert_comment select_user_comments )],
);
```

Then a hypothetical `User` module can `use Lots::Of::SQL ':user';` and receive only the SQL it needs.

This isn't the end of the story. Suppose you want DBAs or non-Perl types to edit and reuse the SQL. "Build a SQL Library" [Hack #23] has ideas.

Perhaps maintaining those export lists by hand is too much work. Using attributes [Hack #45] could simplify your life.

Maybe static SQL written for a single database isn't your style. Try generating it with a templating system or using an abstract, Perlish representation [Hack #24] instead. You might even switch to a persistence or object-relational mapping module such as `Class::DBI`. There are plenty of options, once you untangle SQL from Perl.

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

Hack 23. Build a SQL Library

Store queries where non-programmers can maintain them.

Most serious programmers know the dangers of mixing their user interface code (HTML, GUI, text) with their business logic. When you have a designer making things pretty, it's too much work for any programmer to integrate change after change to font size, placement, and color.

If you have a DBA, the same goes for your SQL.

Why not keep your queries where they don't clutter up your code and where your DBA can modify and optimize them without worrying about a misplaced brace or semicolon breaking your software? If you use `SQL::Library` with a plain text file under version control, you can.

The Hack

Install `SQL::Library` from the CPAN. Extract all of the SQL from your code into one place [Hack #22], and then put it all in a plain text file in INI format:

```
[select_nodemethod_attributes]
SELECT    types.title     AS class,
          methods.title   AS method,
          nodemethod.code AS code
FROM      nodemethod
LEFT JOIN node            AS types
ON        types.node_id = nodemethod.supports_nodetype
```

The section title (the names in square brackets) is the name of the query and the rest is the SQL. Save the file (for example, *nodemethods.sql*). Then from your code, create a `SQL::Library` object:

```
use SQL::Library;

my $library = SQL::Library->new({ lib => 'nodemethods.sql' });
```


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

Running the Hack

Whenever you need a query, retrieve it by name from the library:

```
my $sth = $dbh->prepare( $library->retr( 'select_nodemethod_attributes' ) );
```

From there, treat it as normal.

Hacking the Hack

This isn't very exciting until you get to more complex queries—where the order of joins is important, where the exact nature of queries changes, or where there's lots of manipulation and editing going on. Being able to modify the SQL without touching the code is very handy.

For example, consider a reporting application. Choose a filename to hold the queries. Write a bit of code that processes the queries and feeds them to a library to produce graphs or spreadsheets. (The trick with `NAME_lc` in "Bind Database Columns" [Hack #25] is very useful here.) Then just loop through all of the queries in the library, preparing and executing them, and processing the results:

```
use SQL::Library;

my $library = SQL::Library->new({ lib => 'daily_reports.sql' });

for my $query ( $library->elements( ) )
{
    my $sth = $dbh->prepare( $query );
    my %columns;

    $sth->bind_columns( \\@columns{ @{ $sth->{NAME_lc} } } );
    $sth->execute( );

    process_report( \\%columns );
}
```

Now whenever your users want another query, just write it and store it in the appropriate library file. You never have to touch the reporting program (as long as it can draw its pretty graphs correctly)—and if you can teach your users to write their own queries, you can make your job that much easier.

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

Hack 24. Query Databases Dynamically Without SQL

Write Perl, not SQL.

SQL is a mini-language with its own tricks and traps. Embedded SQL is the bane of many programs, where readability and findability is a concern. Generated SQL isn't always the answer either, with all of the quoting rules and weird options.

In cases where you don't have a series of fully baked SQL statements you always run—where query parameters and even result field names come from user requests, for example—let `SQL::Abstract` do it for you.

The Hack

Create a new `SQL::Abstract` object, pass in some data, and go.

Suppose you have a reporting application with a nice interface that allows people to view any list of columns from a set of tables in any order with almost any constraint. Assuming a well-factored application, the model might have a method resembling:

```
use SQL::Abstract;

sub get_select_sth
{
    my ($self, $table, $columns, $where) = @_;

    my $sql           = SQL::Abstract->new( );
    my ($stmt, @bins) = $sql->select( $table, $columns, $where );
    my $sth           = $self->get_dbh( )->prepare( $stmt );

    $sth->execute( );
    return $sth;
}
```

`$table` is a string containing the name of the table (or view, preferably) to query, `$columns` is an array reference of names of columns to view, and `$where` is a hash reference associating columns to values or ranges.

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

If a user wants to query the `users` table for `login_name`, `last_accessed_on`, and `email_address` columns for all users whose `signup_date` is newer than `20050101`, the calling code might be equivalent to:

```
my $table   = 'users';
my $columns = [qw( login_name last_accessed_on email_address )];
my $where   = { signup_date => { '>=', '20050101' } };
my $sth     = $model->get_select_sth( $table, $columns, $where );
```

The returned `$sth` is a normal iterable `DBI` statement handle, suitable for passing to a templating system or other user interface view component. This is very useful for selecting only the interesting parts of a table or view.

Hacking the Hack

There's no reason you have to let users select the kind of information they want to view. Perhaps you have system administrators who should be able to see (and update) any non-key column in the `users` table, managers who should be able to see and update most personnel-related columns, and normal users who should only see demographic information.

You can use the same underlying model to fetch information from the database—just add a layer over it to exclude requested columns that the particular user of the system shouldn't see. Assuming that you have an object representing the user type with a method that returns the allowed columns for a particular table, call `restrict_columns( )` before `get_select_sth( )`:

```
sub restrict_columns
{
    my ($self, $user, $table, $columns) = @_;
    my $user_columns                    = $user->get_columns_for( $table );
    return [ grep { exists $user_columns->{ $_ } } ] @$columns;
}
```

Instead of maintaining separate SQL queries for each type of user accessing the system, you can maintain a list somewhere of appropriate view and update columns for each type of user, reusing the query generator. If you keep the list of types and allowed columns in the database or in a configuration file somewhere, you have data-driven programming and an easy-to-maintain system.

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

Hack 25. Bind Database Columns

Use placeholders for data retrieved from the database, not just sent to it.

Experienced database programmers know the value of placeholders in queries. (Inexperienced database programmers will soon find out why they're important, when unquoted data breaks their programs.) When you execute a query and pass in values, the database automatically quotes and inserts them into the prepared query, usually making for faster, and always making for safer, code.

Perl's `DBI` module has a similar feature for retrieving data from the database. Instead of copying column after column into variables, you can *bind* variables to a statement, so that they will contain the appropriate values for each row `fetch( )`ed.

Of course, this technique appears less flexible than retrieving hashes from the DBI, as it relies on the order of data returned from a query and loads of scalar variables...or does it?

The Hack

Suppose that you have a templating application that needs to retrieve some fields from a table[1] and wants to contain the results in a hash. You could write a subroutine named `bind_hash( )`:

[1] Or, better, a view or stored procedure....

```
sub bind_hash
{
    my ($dbh, $hash_ref, $table, @fields) = @_;

    my $sql = 'SELECT ' . join(', ', @fields) . " FROM $table";
    my $sth = $dbh->prepare( $sql );

    $sth->execute( );
    $sth->bind_columns( \\@$hash_ref{ @{ $sth->{NAME_lc} } } );

    return sub { $sth->fetch( ) };
}
```


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

The only really tricky part of the code is using the reference operator (\\) on a hash slice. When fed a list, this operator produces a list of references to the values in the list—and a hash slice returns a list of the values, themselves scalars. The NAME_lc property of an active statement handle contains an anonymous array of lowercased field names that the statement will retrieve. This can improve portability.

Running the Hack

Suppose that you have a users table[2] and you want to retrieve the names, birthdays, and shoe sizes of all of the users, and print them nicely. That's easy:

[2] Or view or stored procedure....

```
# assume you already have $dbh connected

my %user;

my $user_fetch = bind_hash( $dbh, \\%user, qw( users name dob shoe_size ) );

while ($user_fetch->( ))
{
    print "$user{name}, born on $user{dob}, wears a size " .
          "$user{shoe_size} shoe\\n";
}
```

This hack only works well when you're fetching a row at a time. It's also not the right way to build a quick and easy object-relational mapper, because by the time you need a new hash for each row, you've already bound it. That's okay—it's still very fast and flexible and lends itself well to the iterator technique [Hack #26].

Hack 26. Iterate and Generate Expensive Data

Hide lists, streams, and expensive data structures behind a simple interface.

Perl's fundamental aggregate data types—hashes and arrays—are wonderfully flexible and often just what you want. That's often, not always. Sometimes you really need to process data that's expensive to calculate, part of a huge list that won't fit into memory, or just never ends.

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

When that happens, use a function reference as a data structure. Seriously.

The Hack

Imagine that you've just taken a job as a network administrator, replacing someone who completely failed to do any documentation. You know that you have all sorts of devices on the network with static IP addresses and you have a rough idea of the network blocks, but you don't know which addresses are in use.

Rather than finding every device, checking its settings, and reassigning things, you can write a little program to loop through each address and try to contact the device. It's a good first approximation. How do you check every netblock though? Use `Net::Netmask` to generate a list of IP addresses.

That could get messy though—do you really want to loop over a list of potentially millions of addresses? This is a good place to use a generator.

```
use Net::Netmask;

sub create_generator
{
    my @netmasks;

    for my $block (@_)
    {
        push @netmasks, Net::Netmask->new( $block );
    }

    my $nth = 1;

    return sub
    {
        return unless @netmasks;
        my $next_ip = $netmasks[0]->nth( $nth++ );

        if ( $next_ip eq $netmasks[0]->last( ) )
        {
            shift @netmasks;
            $nth = 1;
        }

        return $next_ip;
    }
}
```


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

Running the Hack

Pass `create_generator( )` a list of IP network blocks and netmasks and it will return a function reference that, when called, returns either the next address in the series or the undefined value if you've exhausted everything. It does this by closing over two variables, the list of `Net::Netmask` objects in `@netmasks` and a counter variable `$nth`. The latter represents the current position in the list of available addresses for the current `Net::Netmask` object.

To test an IP address for an active device, just pull a new address from the iterator by executing it:

```
my $next_address   = create_generator( '192.168.1.0/8', '10.0.0.0/16' );

while (my $address = $next_address->( ))
{
    # try to communicate with machine at $address
}
```

If you have a huge group of addresses to check, this is much more memory- and time-friendly than generating a list of hundreds of thousands of addresses all at once.

Hacking the Hack

With a generator as large as this one and the inevitable delay for network communication, you might want a way to suspend and resume from a certain point. If you turned the generator function reference into an object, you could add a `serialize( )` or `store ( )` method that saves the current state. Then you can resume from almost any point. All you need to save is the `base( )` and `bits( )` information from each active `Net::Netmask` object (presumably in the proper order) and the current value of `$nth`.

Of course, in a program that probably has network communication as its most significant bottleneck, you may want to check several addresses in parallel. "Pull Multiple Values from an Iterator" [Hack #27] can help.

Mark Jason Dominus's *Higher Order Perl* (Morgan Kaufmann, 2005) shows how to use functional programming techniques in Perl, including iterators and generators. This book is worth studying in detail.

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

Hack 27. Pull Multiple Values from an Iterator

Make your iterators and generators highly context-sensitive.

Iterators and generators are fantastically useful for data that takes too long to generate, may never run out, or costs too much memory to keep around. Not every problem works when reading one item at a time though, and finding a nice syntax for pulling only as many items as you need can be tricky.

Perl's notion of context sets it apart from many other programming languages by doing what you mean. When you want a single item, it will give you a single item. When you want nothing, it can give you nothing. When you want a list, it will oblige. That power is yours through the `wantarray( )` operator, too.

Wouldn't it be nice if Perl could tell how many items you want from an iterator or generator without you having to be explicit? Good news—it can.

Better Context than wantarray( )

Robin Houston's `Want` module extends and enhances `wantarray( )` to give more details about the calling context of a function. Besides distinguishing between void and scalar context, `Want`'s `howmany( )` function can tell *how many* list items the calling context wants, from one to infinity.

The Code

Consider a simple generator that implements a counter. It takes the initial value, the destination value, and an optional step size (which defaults to 1). When it reaches the destination, it returns the undefined value.

```
sub counter
{
    my ($from, $to, $step)  = @_;
    $step                  ||= 1;
```


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

```
    return sub
    {
        return if $from > $to;
        my $value       = $from;
        $from          += $step;
        return $value;
    };

}
```

Creating and using a counter, perhaps one that counts from 1 to 10 by threes, is easy:

```
my $counter = counter( 1, 10, 3 );
my $first   = $counter->( );
```

What if you want the next three steps though? You could call it in a loop, but wouldn't it be nicer to call it with:

```
my ($first, $second, $third) = $iterator->( );
```

That's where `multi_iterator( )` comes in. Feed it an iterator or generator and it returns a function that acts as a drop-in replacement for the iterator or generator but respects the calling context:

```
use Want 'howmany';

sub multi_iterator
{
    my ($iterator) = @_;

    return sub
    {
        my $context = wantarray( );

        return              unless defined $context;
        return $iterator->( ) unless         $context;
        return map { $iterator->( ) } 1 .. howmany( );
    };
}
```

The multi-iterator first must check for void context (so it returns nothing and never kicks the contained iterator), then scalar context (so it can kick the iterator once). Then it kicks the iterator as many times as necessary to produce the number of expected values. Whatever the behavior of the contained iterator or generator when it exhausts its possible values, the multi-iterator will pass along to the caller.

This takes one more step than before, but the results speak for themselves:

```
my $counter          = counter( 1, 10, 3 );
my $iterator         = multi_iterator( $counter );
```


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

```
# multiple variables, list context
my ($first, $second) = $iterator->( );

# void context
$iterator->( );

# single variable, scalar context
my $third            = $iterator->( );

# single variable, list context
my ($fourth)         = $iterator->( );
```

`$first` contains the value 1 and `$second` the value 4. So far so good. `$third` contains 7 and `$fourth` 10. All subsequent accesses will contain `undef`.

Hacking the Hack

Being able to iterate over multiple iterators in parallel would be very useful. That's doable here.

This technique works outside of iterators as well; in any place you distinguish between list and scalar context and may need to know more about one-element list context versus *n*-element list context, `howmany( )` is useful.

`Want` has many other interesting context-related features; it's worth exploring further on its own. Fortunately, its documentation is very useful.

Be careful about assigning the results of the iterator call to an array, which effectively has infinite elements. It may not do what you want if you have an infinite generator or iterator (unless you want an infinitely large array consuming infinite amounts of memory and taking infinite time to complete).

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