

Tricks of the Wizards

Mark Jason Dominus

Plover Systems Co.

mjd-omniti-tricks@plover.com



v1.8 (April, 2005)

- If you don't understand this, you may not get much out of this class:

```
my $x = M->yup(13);
print $x->{V}, "\n";

sub M::yup { my ($p, $v) = @_ ; bless {V => $v}, $p }
```



What We'll See

- Magic
 - Dark hidden corners of Perl
 - Strange Incantations
- Specifically
 - Globs
 - More globs
 - Tie
 - Source Filters
 - Powerful uses of these things



Prerequisites

You Must Already be Conversant with:

- Packages
- References
- Objects
- Modules

If not, so sorry!

- Gildor says: "Do not meddle in the affairs of Wizards"



Warning

- These techniques are powerful but strange
- They might make your programs hard to understand
- 'Incantation' or 'Idiom'?
- The Mighty Marvel Wizard says: "With great power comes great responsibility"
- Everything looks 'obfuscated' the first time you see it
- No complaints about obfuscation, please

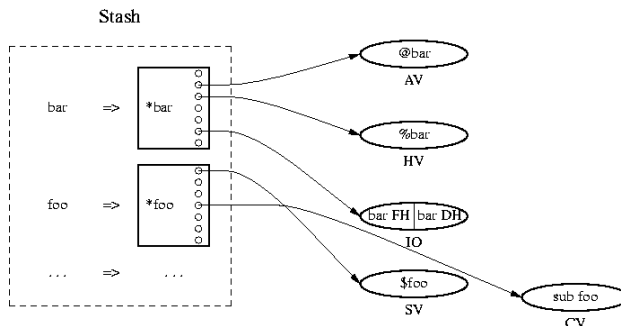


Warning #2

- Many of the techniques we'll see *directly* violate `strict refs` in the grossest and most blatant ways.
- That is not a flaw in the methods.
- `strict refs` is a safety feature.
- If you want to learn to use the Wand of Fireballs, you have to shut off the automatic sprinklers first.
- No complaints about `strict` failures, please.

Principles Of Magic

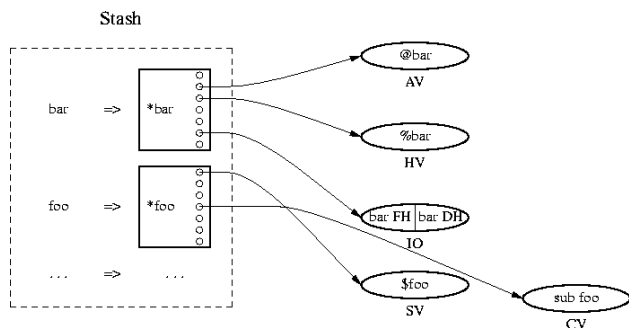
- Much magic is about making things appear to be what they're not
- First we have to understand what makes things appear as they are -- `$foo` for example?
- The Perl Symbol table:



- Several parts:
 - The stash
 - The globs
 - The SVs, AVs, HVs, etc.

The Magic Path to Enlightenment

- How is the value of `$foo` looked up?



- Figure out package name

```
Parse::RecDescent::foo --> Parse::RecDescent
Person::new --> Person
foo --> (Whatever is current)
```

- Look in stash for package, locate key `foo`
- Value is a glob. Extract SCALAR part of glob.
- Result is a pointer to an SV
- The SV is the value. (NULL pointer == undef)

The Magic Path to Enlightenment

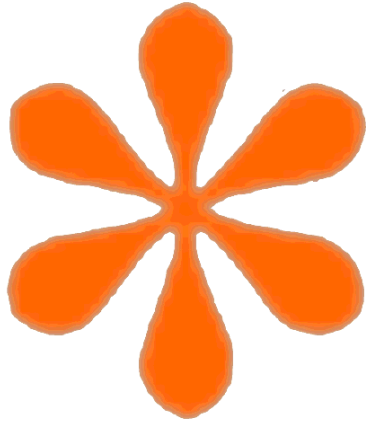
- The stash is a hash whose values are globs
- The values are pointers attached to the knobs of the globs
- Follow the knob of the glob in the hash for the stash



- All of these steps are interesting.
- We can benefit by enchanting any of them.
- Globs first.

Making Things Appear to Be What They're Not

Part I: Globs



Globs

- A glob is the glue between the symbol tables and the actual values.
- We're going to spend a lot of time on globs
- A glob has seven parts:
 - SCALAR
 - ARRAY
 - HASH
 - CODE
 - IO
 - FORMAT
 - GLOB



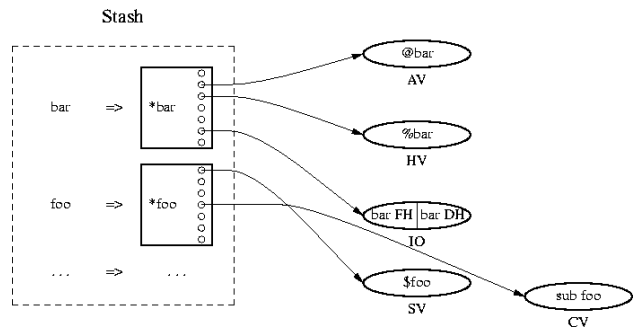
Accept no substitutes

- Despite the resemblance, globs have nothing to do with this:



Globs

- A glob has seven parts:



Globs

- When perl resolves a variable name, it goes through the glob
- Tinkering with the globs alters the way variables are looked up
- Glob notation in Perl:

*foo



Operations with Globs

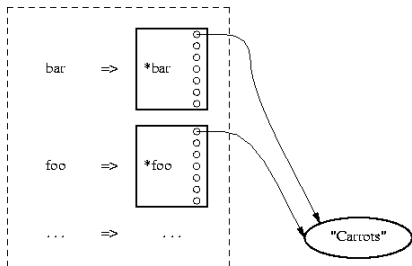
- Most useful:
 - *foo = REFERENCE
- The thing referred to is attached to the appropriate glob knob



Operations with Globs

- This performs installation into the symbol table

*bar = \ \$foo;



*bar = \ \$foo;

- The glob assignment *aliases* the value

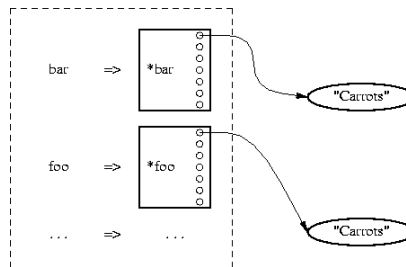


Aliasing

*bar = \ \$foo;

- Aliasing is different from assignment:

\$bar = \$foo;



- Assignment *copies* the SV and installs the *copy* into the symbol table

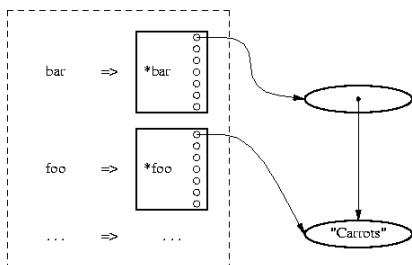


Aliasing

```
*bar = \$foo;
```

- Aliasing is different from assigning a reference:

```
$bar = \$foo;
```



- `\$foo` constructs a *new SV* with a *reference* value
- The *reference* is installed into the symbol table

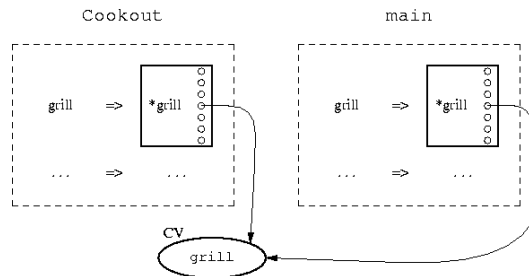
Exportation

```
package Cookout;
sub import {
    my $caller = caller;
    *{$caller . '::grill'} = \&grill;
}
sub grill {
    ...
}
```

- This module *exports* the function `grill` into the calling package:

```
use Cookout; # Calls Cookout->import()
grill('kebabs'); # Calls Cookout::grill('kebabs')
```

- `$caller . '::grill'` turns into `main::grill`



- Now you know how the Exporter works

Exportation

- Here's a slightly more full-featured exporter.

```
package Rings;
use Carp;

%exports = map {$_ => 1} qw(Narya Neny Vilya);

sub import {
    my $caller = caller;
    my $package = shift;
    for my $name (@_) {
        unless ($exports{$name}) {
            croak("Module $package does not export &$name; aborting");
        }
        *{$caller . '::' . $name} = \&{'Rings::' . $name};
    }
}

sub Narya { ... }
sub Neny  { ... }
sub Vilya { ... }
```

- In the main program, calls like this:

```
use Rings qw(Narya Neny Fred);
```

- Turn into this:

```
Rings->import(qw(Narya Neny Fred));
```

croak

- Consider: `main.pl`

```
#!/usr/bin/perl
use Rings qw(Narya Neny Fred);
```

- And `Rings.pm`

```
...
die "Module $package does not export &$name; aborting";
...
```

- This yields

```
... does not export &Fred; aborting at line 379 of Rings.pm.
```

- Not very useful

- With `croak` instead of `die`

```
... does not export &Fred; aborting at line 2 of main.pl.
```

- Similarly `carp` instead of `warn`



Forced Importing / Aliasing

- Module `VeryLongName` contains function `SomeFunction`.
- Instead of calling `VeryLongName::SomeFunction` many times:

```
{ local *F = \&VeryLongName::SomeFunction;
  F(...);
}
```

- A more realistic example:

```
*ERR = \&$DBI::errstr;
```



Forced Importing / Aliasing

- Another real example: `Module` has a function you want, but the name is wrong:

```
use Module 'function';
```

- Perhaps this is not good because it overlaps some other function that you need

- For example:

```
sub get { ... } # Clobbered by LWP::Simple::get
use LWP::Simple; # Ouch --- exports 'get' by default
```

- Do this instead:

```
use LWP::Simple (); # Load, but don't import anything
BEGIN { *webget = \&LWP::Simple::get }
```

- use `Module ()` is a weird special case
- It loads `Module` but does not call `import` at all



(No) Globs in Perl 6

- Everyone seems to know that Perl 6 won't have globs

○ (Even people who don't know what globs are)

- In Perl 5, globs are essential to exporting

○ How will exportation be handled in Perl 6?

- Exportation is an aliasing operation
- Perl 6 has an explicit aliasing operator `:=`

```
$new := $old;
@new := @old;
%new := %old;

@new := $oldref;
%new := $oldref;
```

- These will work even if `new` is a lexical variable



(No) Globs in Perl 6

- For exportation to another package one will use:

```
%Cookout::{ '&grill' } := \&grill;
```

- Stashes in Perl 6 are still hashes

- They have names that end in `::`

- The key `&grill` in a stash is associated with the function object

- The Exporter itself will do something like

```
my $calling_package = caller().package;
my %Exporter::To:: := %{$calling_package _ '::'}
...
%Exporter::To::{ $name } := %Exporter::From::{ $name };
```



Passing Filehandles

- In The Beginning, filehandles weren't first-class values
- Consider code like this:

```
open FH, ...;
print FH ...;
$z = <FH>;
close FH;
```

- Here FH is actually a *literal string* (a 'bareword')
- Almost as if you had written something like this:

```
open "FH", ...;
print "FH" ...;
$z = <"FH">;
close "FH";
```

- All Perl's I/O functions expect to get strings
 - They then resolve the string to a glob in the usual way
 - Then they extract the filehandle part of the glob

Passing Filehandles

- This method for filehandles causes some problems

```
open FH, ...;
$data = read_block(FH);

package My::IO;

sub read_block {
    my $fh = shift;
    my $buf;
    read $fh, $buf, $BLOCKSIZE;
    $buf;
}
```

- Here the read function is given the string FH
 - But FH means My::IO::FH, not main::FH
 - Function doesn't work

Passing Filehandles

```
$data = read_block(FH);          # Doesn't work

package My::IO;

sub read_block {
    my $fh = shift;
    my $buf;
    read $fh, $buf, $BLOCKSIZE;
    $buf;
}
```

- Solution 1:


```
$data = read_block(main::FH);
```
- Solution 2:


```
$data = read_block(\*FH);
```
- Perl's I/O functions all will glob references
 - They then access the glob through the reference instead of through the stash

Passing Filehandles

- Similarly:


```
open my $fh, ...;
```
- This now creates a new filehandle and stores it in \$fh
- What is actually created?


```
print "$fh\n";
GLOB(0x80f7b0c)
```
- A glob reference
 - It's a glob that's not part of the symbol table
 - There are no aliasing effects on assignment
- In Perl 6, open will simply return a filehandle object:


```
my $fh = open $filepath : mode=>'rw'
```
- It will probably stringify as something like IO(0x436c1d)

Globjjects

- You're probably familiar with using a blessed hash as an object
- Hash elements are like C++ class members
- Many people suggest using an array for space and time efficiency
 - See Greg Bacon's TPJ article
 - This trick was codified in 5.005's *pseudohash* feature
 - Which was subsequently removed

Globjjects

- Base object on array? Or hash?
- There are tradeoffs here
- What if you need both? Use a glob!
- A glob contains a hash *and* an array
- And also a filehandle

Globjjects

- The biggest win is using the filehandle part
- Perl accepts a glob reference anywhere it normally expects a filehandle
- If your object is a blessed glob reference, people can use it like a filehandle
- Let's write an object that looks like a regular filehandle
 - But it supports a `flush` method that flushes any buffered data

Globjjects

```
package IO::Flushable;
sub new {
    my ($package, $mode, $filename) = @_;
    open my $fh, $mode, $filename or return;
    bless $fh => $package;
}
```

- People can use this object just like a filehandle:

```
my $fh = IO::flushable->new(">", "logfile") or die ...;
print $fh "Blah blah blah\n";
syswrite $fh, $logentry;
close $fh;
```

- It also closes automatically when it is destroyed.

Globjoints

- I promised a flush operation
- `$fh->flush()` will flush the handle

```
sub flush {
    my ($self) = @_;           # $self is a GLOB reference
    my $ofh = select $self;
    my $rc;
    { local $| = 1;
      $rc = print $self "";
    }
    select $ofh;
    return $rc;
}
```

Globjoints

- Here's a more interesting example
- It's like a regular filehandle
 - But it has a `remember` operation that remembers the current file position
 - And a `gobackto` operation that goes back to a saved position
- Changing positions is accomplished with Perl's `seek` and `tell` functions

```
my $pos = tell FH;
seek FH, $pos, 0;
```

Globjoints

- The constructor is similar to the previous example:

```
package IO::Remembers;
sub new {
    my ($package, $filename) = @_;
    open my $fh, $filename or return;
    bless $fh => $package;
}
```

- Once again it can be used like a regular filehandle:

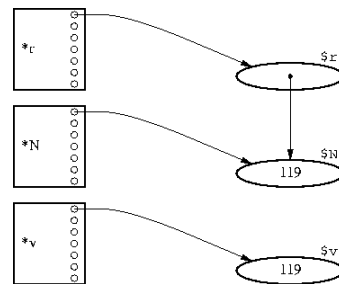
```
my $fh = IO::Remembers->new('input');
my $line = <$fh>;
read $fh, $bytes, 1024;
close $fh;
```

Globjoints

- Before we see `remember` and `gobackto`, here's some syntax
- You can use a glob as if it were a reference to any sort of thing.
- For example:

```
$N = 119;
$v = ${*N};
$r = \${*N};
```

- `$v` now contains 119
- `$r` now refers to `$N` so that `$$r` is 119.
- In particular, if `*g` is a glob, `*$g` is its hash
 - `*g->{key}` looks up `key` in the hash
- If `$gr` is a globref, `*$gr` is the glob
 - `**$gr` is the glob's hash
 - `*$gr->{key}` looks up `key` in the hash



Globjoints

- `$fh->remember('hippo')` will save the current position under the key `hippo`.

```
sub remember {
    my ($self, $key) = @_; # $self is a GLOB reference
    *$self->{$key} = tell $self;
}
```



- `$fh->gobackto('hippo')` will return to the saved position.

```
sub gobackto {
    my ($self, $key) = @_;
    seek $self, *$self->{$key}, 0;
}
```



- Future reads from the 'filehandle' will continue from the old position

Globjoints

- In Perl 6, this will be more straightforward
- Filehandles will just be objects from class `IO`
 - Built-in functions like `print` and `<>` will be method calls
- So just subclass `IO` and add the methods you want

Wrappers

- Suppose we'd like to trace execution of the functions in a package
- To do that, we'll replace each function with a 'wrapper'
 - The wrapper will announce that the function is being called
 - Then call the real function
- Basic idea:

```
my $real_func = \&*$func_name;
*$func_name = sub {
    print "$func_name(@_)\n";
    $real_func->(@_);
};
```

Wrappers

```
package Trace;

sub import {
    my $caller = caller;
    my @functions = @_ ? @_ : all_functions($caller);
    for my $func_name (@functions) {
        my $real_func = \&*$func_name;
        *{$caller . "::$func_name"} = sub {
            print "$func_name(@_)\n";
            $real_func->(@_);
        };
    }
}
```

Stash Walking

```
my @functions = @_ ? @_ : all_functions($caller);
```

- How can we get a list of all the functions in a package?
- We'll examine the stash directly
- It's just a hash
- The stash for package RINGS is available as %RINGS::
 - Keys are names, values are globs

```
sub all_functions {
    my $p = shift;
    my $h = \%{$p . "::*"};
    my @result;
    while (my ($name, $glob) = each %$h) {
        if (defined &$glob) {
            push @result, $name;
        }
    }
    @result;
}
```



Miscellaneous Applications of Globs

Read-Only Constants

```
*PI = \3;
```

- Now \$PI is 3:

```
$circum = 2 * $PI * $r;
```

- But attempts to assign to \$PI fail:

```
$PI = 4;
```

```
Modification of a read-only value attempted at ...
```



Miscellaneous Applications of Globs

Read-Only Constants, Continued

```
sub PI () { 3 }
```

- Now you can use PI and get 3:


```
$circum = 2 * PI * $r;
```
- PI is still read-only:


```
PI = 3;
```

Can't modify constant item in scalar assignment at ...
- () enables special function-call syntax
- This incantation also enables *inlining* optimization



Read-Only Constants Continued

```
use constant PI => 3,
              e  => 2.71828182845904523536,
              emptylist => [];
```

- constant.pm uses a combination of the read-only techniques and exportation:

```
package constant;
sub import {
    my $caller = caller;
    my $package = shift;
    while ($name = shift) {
        my $value = shift;
        *{$caller . '::' . $name} = sub () { $value };
    }
}
```



A Templating System

- This is a pretty big spell.
- You have a hashful of variables, %VARS
- You want to `eval` some code, and you want the environment for the `eval` to be the variables defined by the `hash`.
- For example, many templating modules need to do this

```
my %VARS = ( cust_id => 666,
            items => ['fish', 'dog', 'carrot'],
            amount => 142857.33,
            );

my $template = <<'EOM'; # Or read it from a file
$name = db_lookup('NAME', $cust_id);
$title = db_lookup('TITLE', $cust_id);
$n = @items;
$items = $n == 1 ? "item" : "$n_items items";

return "Dear $title $name,
       You still owe me \$$amount for the following $items:
       @items\n";
EOM
my $result = my_eval($template, \%VARS);

# Result:
# Dear Mr. Gates,
# You still owe me $142857.33 for the following 3 items:
# fish dog carrot
```

- Note: `$cust_id`, `@items`, and `$amount` implicitly defined by the hash
- Note: `$name`, `$title`, `$n`, and `$items_list` don't 'leak out'

A Templating System

Make Up a New Package

- Straightforward:

```
my $fake_pack;
BEGIN { $fake_pack = 'Fake00' }
sub new_package {
    return "HashEval::" . $fake_pack++;
}
```

- `Symbol::gensym` already does something like this
- After we're done with a package, we can destroy it by using the `Symbol::delete_package` function

A Templating System

Three Parts to Our Strategy

1. Make up a new package
2. Install the hash variables into the new package
3. Do the `eval` in the new package

A Templating System

Install Hash Variables Into the New Package

```
sub package_install {
    my ($h, $p) = @_;
    my $n;
    while ($n = each %$h) {
        my $v = $h->{$n};
        *{$p . '::$' . $n} = (ref $v ? $v : \ $v);
    }
}
```

- Scalar context `each` just returns the keys one at a time

A Templating System

Do the `eval` in the New Package

```
sub my_eval {
  my ($program, $hash) = @_;
  my $pack = new_package();
  package_install($hash => $pack);
  my $result = eval "package $pack; $program";
  return $result;
}
```

A Templating System

Caveats

- The `eval`'ed code is not actually *confined* to the new package:

```
$/ = 'e'; # Sucker!
$Security::ENABLED = 0; # Double sucker!
```

- `eval` is still `eval`

```
system("rm -rf /");
```

- To prevent these, you need to use `Safe`.

- The hash-into-new-package strategy is still valuable in conjunction with `Safe`.

```
my $result = Safe->new->reval($program);
```

- `Text::Template` is an extended example of this.



Making Things Appear to Be What They're Not

Part II: Ties



Ties

- A *typed* variable has its accesses mediated by a Perl object.

- For example, if the scalar `$s` is *typed* to the object `$o`, then

```
print $s;           print $o->FETCH();
$s = 119;          $o->STORE(119);
```

Ties: Trivial (Annoying) Example

Make Something Look Strange

- Tied variables are the ultimate in things that appear what they're not:

```
sub STORE {
    my ($self, $val) = @_;
    # Return value is ignored
}

sub FETCH {
    return "You are not cleared for access to that information.";
}
```

- Now what?

```
$cia = "I'm a happy little bunny wabbit";
$cia =~ tr/A-Z/a-z/;
$cia .= "foo";

print $cia;
```

You are not cleared for access to that inform



Tied Scalar Example

```
use Sequence;
tie $IDS => Sequence, 17;          # $IDS is special now

$id = $IDS;                       # $id is now 17
$another = $IDS;                  # $another is now 18
print $IDS, "\n";                 # Prints 19
push @ids, $IDS;                  # Pushes 20

$IDS = 17;                         # Reset to 17
```

How to Tie

- Basic syntax:

```
tie $VAR => PACKAGE, arguments;
tie @VAR => PACKAGE, arguments;
tie %VAR => PACKAGE, arguments;
tie *VAR => PACKAGE, arguments;
```

- Turn into

```
PACKAGE->TIESCALAR(arguments);
PACKAGE->TIEARRAY(arguments);
PACKAGE->TIEHASH(arguments);
PACKAGE->TIEHANDLE(arguments);
```

- The TIEXXX function must construct and return an object to be associated

Tied Scalar Example

```
package Sequence;

sub TIESCALAR {
    my ($package, $start) = @_;
    $start = 1 unless defined $start;
    my $object = {VALUE => $start};
    bless $object => $package;
}

sub FETCH {
    my ($self) = @_;
    $self->{VALUE}++;
}

sub STORE {
    my ($self, $newvalue) = @_;
    $self->{VALUE} = $newvalue;
}
```

Tied Hash Example

A hash with case-insensitive keys

```
use Insensitive;
tie %hash => Insensitive;

$hash{SomeKey} = 'somevalue';
$hash{'John MacDonald'} = 'Author';

print $hash{somekey}, "\n";      # Prints 'somevalue'
print $hash{'John Macdonald'}, "\n"; # Prints 'Author'

$hash{SOMEKEY} = 57;
print $hash{SomeKey}, "\n";      # Prints 57
```



Tied Hash Example

```
package Insensitive;

sub TIEHASH {
    my ($package) = @_;
    my $object = {};
    bless $object => $package;
}

sub STORE {
    my ($self, $key, $value) = @_;
    $self->{lc $key} = $value;
}

sub FETCH {
    my ($self, $key) = @_;
    $self->{lc $key};
}
```



CGI.pm

- CGI.pm provides a `->param()` method for getting the submitted web form data
- For compatibility with older packages, it will also set up an `%in` hash
- `%in` is tied to call `->param()` behind the scenes

```
sub FETCH {
    return $_[0] if $_[1] eq 'CGI';
    return undef unless defined $_[0]->param($_[1]);
    return join("\0", $_[0]->param($_[1]));
}
```

- Other methods similarly



Exporting a Tied Variable

- You can `tie` any scalar, array, or hash variable.
- It could be global or lexical
- You can export it also
- You can use this to write a module that places a magical variable into the package that uses it.
- Normally, `use Package` imports some functions in the program that says it
- But you can make `use Package` mean to import some magical variables instead



Config.pm

- Perl's standard `Config` module supplies a magical `%Config` hash
 - It appears to be full of information about Perl's configuration

```
use Config;

print "osname = $Config{osname}\n";
print "install module manuals into = $Config{installman3dir}\n";

osname = linux
install module manuals into = /usr/local/man/man3
```

- Actually `%Config` is a tied hash

```
package Config;
...
@EXPORT = qw(%Config);
...

sub import {
...

  *{"$callpkg\::Config"} = \%Config;
}

...

tie %Config, 'Config';

1;
```

- `"$callpkg\::Config"` is equivalent to `$callpkg . "::Config"`
 - `"$callpkg::Config"` means something else

Magical Exporter Variable

- This nifty trick was invented by Andrew Pimlott
- Beginners want to say this:


```
$salary = 43_000;
print "After your raise, you will make $salary*1.06.\n";
```
- But it doesn't work:


```
After your raise, you will make 43000*1.06.
```
- Because, of course, expressions aren't evaluated inside of strings.
- ...or are they?

```
@s = (1, 4, 9, 16, 25, 36);
print "$s[(2+7-1*3)/2]\n";      # Prints 16
```

Config.pm

- `Config.pm` contains most of the configuration information as a giant string
- The string is *not* parsed when you load the module
- Instead, the `FETCH` method searches it for the configuration variable you asked for
- Then it caches the result
- `FETCH` also generates some of the configuration information dynamically
- `%Config` is read-only:

```
sub STORE { die "\%Config::Config is read-only\n" }
```

Exporting a Magical Variable

```
package Eval;

sub import {
  my ($package, $name) = @_;
  $name = 'Eval' unless defined $name;
  my %magical_hash;
  tie %magical_hash => Eval;
  my $caller = caller;

  *{"$caller . '::' . $name"} = \%magical_hash;
}
```

- There's that magic glob again.

```
sub TIEHASH {
  my $self = \'dummy';
  bless $self => 'Eval';
}
```

- use `Eval` now calls `Eval::import`
- `import` creates and ties a hash, which it exports back to the caller
- When the caller examines the data in the hash, `Eval::FETCH` is called

Exporting a Magical Variable

- Here's `Eval::FETCH`

```
sub FETCH {
  my ($self, $key) = @_;
  $key;
} # Do NOTHING!
```

- What was *that* all about?

```
use Eval;

$salary = 43_000;
print "After your raise, you will make $Eval{$salary*1.06}.\n";
```

After your raise, you will make 45580.

- If you don't like the syntax, you can change it a little:

```
use Eval => ':';

$salary = 43_000;
print "After your raise, you will make ${:$salary*1.06}.\n";
```

After your raise, you will make 45580.

Magical Exporter Variable

- Magic hash is not limited to evaluation:

```
package Format_Money;

sub FETCH {
  my ($dummy, $amount) = @_;
  my ($dollars, $cents) = split /\./, sprintf("%.2f", $amount);
  1 while $dollars =~ s/^([-+]?\d+)(\d{3})/$1,$2/; # FAQ
  "\$$dollars.$cents";
}
```

- Now:

```
use Format_Money;

$salary = 43_000;
print "After your raise, you will make $Money{$salary*1.06}.\n";
```

After your raise, you will make \$45,580.00

- Also use for automatic URL character escaping (for example)

- Also see Interpolation module

Tied Arrays

- It's easy to make an array that mirrors the contents of a file

```
tie @FILE, 'MirrorFile', $filename or die ...;
```

- Then

```
print $FILE[13]; # Print line 13

for (@FILE) {
  if (/something/) { ... }
}
```

Tied Arrays

```
package MirrorFile;

sub TIEARRAY {
  my ($package, $filename) = @_;
  open my $fh, "<*", $filename or return;
  my $self = { FH => $fh, FILE => $filename, CACHE => [] };
  bless $self => $package;
}

sub FETCH {
  my ($self, $lineno) = @_;
  return $self->{CACHE}[$lineno]
    if defined $self->{CACHE}[$lineno];

  my $fh = $self->{FH};
  while (<$fh>) {
    push @{$self->{CACHE}}, $_;
    return $_ if $#{ $self->{CACHE} } == $lineno;
  }
  return;
}

sub FETCHSIZE {
  my ($self) = @_;
  my $fh = $self->{FH};
  push @{$self->{CACHE}}, <$fh>;
  scalar @{$self->{CACHE}};
}
```

- Supporting `STORE` is quite difficult

○ See `Tie::File` for many details

Tied Filehandles

- To tie a handle, tie the glob in which it resides:

```
tie *FH => 'Package', ...;
```

- Tied handle objects must support several methods:

```
CLOSE
GETC
PRINT
PRINTF
READ      (for 'read')
READLINE  (for '<>')
WRITE     (for 'syswrite')
```



Tied Filehandles

- For example, suppose you'd like to trap all STDOUT output in a file
- But also send it to STDOUT as usual

```
package TeeSTDOUT;

sub import {
    my ($package, @outfiles) = @_;
    open REAL_STDOUT, ">&STDOUT" or die ...;
    my @handles;
    for my $outfile (@outfiles) {
        open my $fh, ">", $outfile or die ...;
        push @handles, $fh;
    }
    tie *STDOUT => 'TeeSTDOUT', \@handles;
}

sub TIEHANDLE {
    my ($package, $fhs) = @_;
    bless $fhs => $package;
}

sub PRINT {
    my ($fhs, $string) = @_;
    for my $outhandle (@$fhs, \*REAL_STDOUT) {
        print $outhandle $string;
    }
}
```



Tied Filehandles

- Suppose you don't like the `opendir/readdir` interface to directories
- Why not a regular filehandle?
- Then you could do:

```
use Dir;
my $dh = Dir->open(".") or die ...;
while (<$dh>) {
    # Do something with the filename in $_
}
close $dh;
```

- We'll do this by tying the handle in `$dh`, which will allow us to overload the `<...>` operator on it



Tied Filehandles

```
package Dir;

sub open {
    my ($package, $dir) = @_;
    opendir my $dh, $dir or return;
    local *FH;
    tie *FH => 'Dir', $dh, $dir;
    return \*FH;
}

sub TIEHANDLE {
    my ($class, $dirhandle, $dirname) = @_;
    my $self = { DH => $dirhandle, NAME => $dirname };
    bless $self => $class;
}

sub READLINE {
    my ($self) = @_;
    readdir($self->{DH});
}
```



Tied Filehandles

```
sub READLINE {
    my ($self) = @_;
    readdir($self->{DH});
}
```

- Or perhaps you would prefer that `<$dh>` returns an object representing the directory entry:

```
sub READLINE {
    my ($self) = @_;
    my $file = readdir($self->{DH});
    my $fullname = "$self->{DIRNAME}/$file";
    my @statinfo = stat($fullname);
    return unless @statinfo;
    return Dir::Statinfo->new(FULLNAME => $fullname,
                             BASENAME => $file,
                             STATINFO => \@statinfo);
}
```

- And then use it like this:

```
while (<$dh> {
    print $->fullname, " is a ", $->filetype;

    print " containing ", $->size, " bytes"
        if $->filetype eq 'plain file';

    print " linking to ", $->readlink
        if $->filetype eq 'symbolic link';

    print "\n";
}
```

The Mother of All Bizarre tie Tricks

- Some badly-designed library subroutine reads from or writes to a certain variable
- You wish it
 - read from a file
 - wrote to a database
 - called a callback function
 - etc. etc. etc.

instead.

- Solution: Tie the variable.
 - Now it *does* call a callback function instead

Missing tie Methods

- If you assign to a tied variable and you don't have a STORE method defined, you'll get a fatal error.
- The standard `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` classes provide reasonable defaults.
- But for simple behavior, an easy thing to do is

```
sub unimplemented { }
```

- or

```
# Load 'Carp' when needed
sub forbidden {
    require Carp;
    Carp::croak("Operation not permitted on tied hash");
}

for $name (qw(STORE DELETE CLEAR FIRSTKEY NEXTKEY)) {
    *{$name} = \&forbidden;
}
```



- There's that magic glob again.

The Mother of All Bizarre tie Tricks

- Best application: Tied filehandle.

```
do_something(...);
```

- And then to your dismay, `do_something` prints a lot of blather on the `STDOUT`
 - And you cannot get it to shut up
 - Moreover, you want the program to examine the error log for diagnostics
- So tie `STDOUT`:

```
{ my $output;
  tie *STDOUT => 'TrapOutput', \$output;
  do_something(); # Blessed silence
  untie *STDOUT;
  # Now examine $output
}

sub TrapOutput::TIEHANDLE {
    my ($class, $var) = @_;
    bless $var, $class;
}

sub TrapOutput::PRINT {
    my ($self, $string) = @_;
    $$self .= $string;
}
```

- Final remark: `ArrayHashMonster` may amaze and delight you

Making Things Appear to Be What They're Not

Part III: Filters



Filter::Simple

- The easy way to do filtering is with `Filter::Simple`
- We'll build a module that understands rot13-scrambled source code
 - Rot13:


```
abcdefghijklmnop noprstuvwxyz ABCDEFGHIJKLM NOPQRSTUVWXYZ
noprstuvwxyz abcdefghijklm NOPQRSTUVWXYZ ABCDEFGHIJKLM
```

- Our test program looks like this:

```
use Rot13;

zl $f = "Uryyb, jbeyq\a";
$| = 1;
sbe (0 .. yratgu($f)) {
    cevag fhoifge($f, $_, 1);
    fyrrc l vs enaq() < .5;
}

no Rot13;

print "All done!\n";
```

- And in fact this works as written, and produces the output:

```
Hello, world
All done!
```

What's a Filter?

- A filter gets the Perl source code before the parser does
- It can transform the code any way it wants to
- Then it hands the result to Perl
- For example:
 - The source code file might be compressed
 - A filter can uncompress it before Perl sees it
 - The source code file might be encrypted
 - A filter can decrypt it before Perl sees it
 - The source code file might contain non-Perl features like macros
 - A filter can translate these to Perl before Perl sees it
- Filtering is described in the beautifully-written `perlfiler` man page
 - Much of this work was done by Paul Marquess

Filter::Simple

- The magic, of course, is in the `Rot13` module:

```
package Rot13;
use Filter::Simple;

FILTER {
    tr/A-Za-z/N-ZA-Mn-za-m/;
};

1;
```

- It really couldn't be any simpler

Filter::Util::Call

- The filter interface is very complicated
- `Filter::Simple` is based on `Filter::Util::Call`
- Which in turn was invented as a simplified interface
- That's why software is great

**Filter::Util::Call**

- Here's a skeleton usage:

```
package Rot13 ;
use Filter::Util::Call ;
sub import
{
    my($type, @arguments) = @_ ;
    my $result = "" ;
    filter_add(
        sub
        {
            my $status = filter_read() ;
            if ($status >= 0) {
                tr/A-Za-z/N-ZA-Mn-za-m/ ;
            }
            return $status ;
        } )
}
1 ;
```

- A filtering module should provide an `import` which calls `filter_add`
- `filter_add` sets up the filter, which calls `filter_read`
- `filter_read` places a line of code into `$_`
- The filter modifies `$_` and returns

**Filter::Util::Call**

- The filter on the previous slide does not honor `no Rot13`
- And how could it?
 - The filter itself gets the `no Rot13` before the compiler does!
- However, it's easy to make it honor `ab Ebg13`; directives:

```
sub unimport {
    warn "Unimport...\n";
    filter_del();
}
```

- Now the filter gets the `ab Ebg13`; line
 - Rot13s it to `no Rot13`;
 - Returns it to the compiler
 - The compiler compiles the line and calls `Rot13::unimport`
 - `unimport` deletes the filter
 - The parsing and compilation process continues as usual

**Filter::Util::Call**

- The previous version supports `ab Ebg13`; but not `no Rot13`;
- For that we have to be a little more devious:

```
...
if ($status >= 0) {
    if (/^\s* no \s+ Rot13 \s* ; # "no Rot13;"
        \s* (?; #.* )? $      # Optional WS or comment
        /x) {
        return $status;
    }
    tr/A-Za-z/N-ZA-Mn-za-m/ ;
}
...
```

- We examine the line for `no Rot13`; *before* we give it to the compiler
- If so, we return the line *without* rot13ing it
- We could also have called `filter_del()` directly
- `Filter::Simple` does this automatically



Filter::Util::Call

```

...
if ($status >= 0) {
    if (/^\s* no \s+ Rot13 \s* ; # "no Rot13;"
        \s* (?:#.*)? $ # Optional WS or comment
        /x) {
        return $status;
    }
    tr/A-Za-z/N-ZA-Mn-za-m/;
}
...

```

- Note that this won't pick up a line like this one:

```
no Rot13; print "I like pie.\n";
```

- But it will pick this up:

```
$z = qq{
no Rot13;
};
```

- You just have to hope that nothing like that comes along
- In general, source filtering is based on hopes like this one
- You can write a filter that works most of the time
- But faced with sufficiently weird code, it will break

"Only perl can parse Perl"

```
$u = blub / 3; # Is this a comment? ;/;
```

- Where did blub come from?

```

package Blub;
use Astro::MoonPhase;

sub import {
    my $caller = caller;
    my ($phase) = phase(time());
    if (0.4 < $phase && $phase < 0.6) {
        *{$caller . "::blub"} = sub () { 1 };
    } else {
        *{$caller . "::blub"} = sub ($) { $_[0] };
    }
}

```

- This program parses differently when the moon is full
 - So to fully parse all Perl programs, you must be able to determine the phase of the moon
- And that's why only perl can parse Perl
- Nevertheless, filters can do reasonably well in practice

**"Only perl can parse Perl"**

- People are fond of saying this
- If you want to know what Perl will think of some program, you must ask perl itself
- No regex or other simple process will always produce the right answer
- This is because to parse Perl, you also have to be able to *interpret* Perl
- Bizarre but typical example:


```

$t = time / 3; # Is this a comment? ;/;
$s = sin / 3; # Is this a comment? ;/;

```
- Now what about this?


```

$u = blub / 3; # Is this a comment? ;/;

```
- You need to know if blub is like time or like sin

Function Tracing Again

- Filter::Simple can help out with the parsing a little
- Suppose we'd like to instrument each function to announce itself when it's called
- It is sufficient to have each function call 'trace':

```

sub trace {
    ($package, $file, $line, $subr) = caller;
    my $depth = 0;
    1 while defined caller(++$depth);
    my $indent = " " x ($depth - 2);
    local $" = ', ';
    print "$package\:::$subr(@_)\n";
}

```

- Our source filter will find this:

```

sub something {
    ...
}

```

- And replace it with this:

```

sub something {
    &Trace::trace;
    ...
}

```

Function Tracing

```
package Trace;
use Filter::Simple;

sub trace { ... }

FILTER_ONLY code => sub {
    s{^\s* sub \s+ [a-zA-Z_]\w* \s* \{}}
    {$! &Trace::trace; }xmg;
};
```

- `FILTER_ONLY` code will *not* modify this:

```
$z = "
sub z {
    Oh no!
}
";
```

- The code that's passed to the filter actually has

```
$z = \034\000\000\000\001\034;
```

- `Filter::Simple` puts this back the way it was afterwards



Internationalization

- Let's convert a program to run in other languages

```
print "Hello there!\n";
print "Should I erase all your files (yes/no)? ";
chomp(my $response = <>);
if ($response eq 'yes') {
    system("rm -rf $ENV{HOME}");
}
```

- The program shouldn't actually say Hello there!
- Instead, it should consult a database of texts
- In Mexico, the database will contain `;Buenos dias!` instead



Internationalization

```
package Translate;
use Filter::Simple;

my %lexicon = (
    'Hello there!\n' => ";Buenos Dias!\n",
    'Should I erase all your files (yes/no)? ' => '¿Debo borrar todos sus archivos (si/no)? ',
    'yes' => 'si',
);

FILTER_ONLY string => sub {
    unless (exists $lexicon{$_}) {
        warn qq{No translation for "$_\n\n"};
        $lexicon{$_} = $_;
    }
    $_ = $lexicon{$_};
};
```

- Or more likely the `%lexicon` will be tied to a disk database



Perl6::Variables

- The Perl 6 variable syntax is a little different
- Beginners always want element 3 of `@array` to be `@array[3]`
- In Perl 6, it is.

Perl 5	Perl 6
-----	-----
<code>\$s</code>	<code>\$s</code>
<code>\$a[\$n]</code>	<code>@a[\$n]</code>
<code>\$h{\$k}</code>	<code>%h{\$k}</code>
<code>\$s->[\$n]</code>	<code>\$s[\$n]</code>
<code>\$s->{\$k}</code>	<code>\$s{\$k}</code>
<code>\$s->(@a)</code>	<code>\$s(@a)</code>

- We'll build a filter that translates Perl 6 syntax to Perl 5's



Simple AUTOLOAD Example

```
@funcs = qw(red yellow blue);

sub red { ... }
sub yellow { ... }
sub blue { ... }

sub AUTOLOAD {
    die "Function $AUTOLOAD unknown; try [@funcs]\n";
}
```

- Now if you do

```
green(...);
```

- You get this:

```
Function main::green unknown; try [red yellow blue]
```

- The name of the would-be function is placed in \$AUTOLOAD

Simple AUTOLOAD Example

```
@funcs = qw(red yellow blue);

sub red { ... } # etc.

sub AUTOLOAD {
    my ($package, $function) = ($AUTOLOAD =~ /(.*)::(.*)/);
    my $correct = approximate_match($function, \@funcs);
    if (defined &$correct) {
        return &$correct(@_);
    } else {
        die "Function $function unknown; try [@funcs]\n";
    }
}
```

- Now if you do

```
blug(...);
```

it just calls blue for you with the same arguments as if nothing was wrong

- Inside of AUTOLOAD, @_ contains the regular function arguments

Simple AUTOLOAD Example

```
blug(...);           # Calls blue() instead
```

- A few years ago I gave this class at YAPC
- Someone in the audience asked "Are you sure this is a good idea?"

○ No, it's a completely terrible idea

- Unfortunately, Dave Cross was also in the audience
- The result was `Symbol::Approx::Sub`
- At least the documentation says:

Why you would ever want to do this is a complete mystery to me.



Magic goto

- These two are *almost* the same:

```
sub AUTOLOAD {
    return &blue;
}

sub AUTOLOAD {
    goto &blue;
}
```

- On the right is *magic goto*.
- Calls blue normally
- But blue returns directly to AUTOLOAD's caller
- Just as if AUTOLOAD had never been called
- Magic goto is perfect for autoloaded functions

Brief Digression: Tracing Again

- In Part I, we saw a trace utility
- It wrapped each function inside a tracing wrapper:

```
my $real_func = \&*$func_name;
*{$caller . "::$func_name"} = sub {
    print "$func_name(@_)\n";
    $real_func->(@_);
};
```

- If `$real_func` depends on `caller`, it could get confused
- It will notice that it was called from the wrapper, not from the real caller
- Solution:

```
{$caller . "::$func_name"} = sub {
    print "$func_name(@_)\n";
    goto &$real_func;
};
```



Case-Insensitive Function Calls

```
sub closethewindow { ... }

sub AUTOLOAD {
    my ($package, $func) = ($AUTOLOAD =~ /(.*)::(.*)/);
    my $true_func = join '::', $package, lc $func;
    goto &$true_func if defined &$true_func;
    croak "Undefined subroutine &$AUTOLOAD";
}
```

- `defined &foo` checks to see if the function exists
- Now you can call `closeTheWindow`
- or `CloseTheWindow`
- or `CLOSETHEWINDOW`
- It doesn't matter.



Function Call Caching

```
sub closethewindow { ... }

sub AUTOLOAD {
    my ($package, $func) = ($AUTOLOAD =~ /(.*)::(.*)/);
    my $true_func = join '::', $package, lc $func;
    if (defined &$true_func) {
        *$AUTOLOAD = \&$true_func;
        goto &$AUTOLOAD;
    }
    croak "Undefined subroutine &$AUTOLOAD";
}
```



- First time we call `closeTheWindow`, alias the two function names
- Second time, we get `closeTheWindow` directly
- There's that magic glob again
- `goto &$AUTOLOAD` and `*$AUTOLOAD = ...` are common idioms



Typical AUTOLOAD Use: Accessor Methods

```
package Object;
my @attrs = qw(color size price ....); # 637 of these
my %is_attr = map {$_ => 1} @attrs;

sub new {
    my $pack= shift;
    my %self;
    @self{@attrs} = @_;
    bless \%self => $pack;
}

...
```



Direct Emulation of Accessors

```
my @attrs = qw(color size price ... ); # 637 of these
my %is_attr = map {$_ => 1} @attrs;

...

sub AUTOLOAD {
    my $self = shift;
    my ($package, $method) = ($AUTOLOAD =~ /(.*):(.*)/);
    unless ($is_attr{$method}) {
        croak "No such attribute: $method; aborting";
    }

    my $val = $self->{$method};
    $self->{$method} = shift if @_;
    $val;
}

```

- What for?

```
$object->color('red'); # set object's color
$size = $object->size; # fetch object's size

```

- No need to define 637 separate accessor functions
- All handled by one `AUTOLOAD`
- (Warning: This method also gets called for `DESTROY` and others)

Autoloading From a File

- If there's a lot of autoloading code, it makes more sense to keep it in a file

```
sub AUTOLOAD {
    my $file = $AUTOLOAD;
    $file =~ s{::}{/}g;
    $file = "/src/app/autoloading/$file.al";

    open my $fh, "< $file"
        or croak "Couldn't load code from $file: $!; aborting";

    my $code;
    { local $/; $code = <$fh> }
    *$AUTOLOAD = eval $code;
    goto &$AUTOLOAD;
}

```

- The first time the function is called, the code is loaded from the file
- Code for `Some::Module::foo` is in `.../Some/Module/foo.al`
- Code compiled and installed in symbol table as before
- Second time, the function is called directly - no overhead
- Now you know what `AutoLoader` does - invented for `POSIX`

Caching Accessor Methods

- Calling via `AUTOLOAD` incurs overhead
- Aliasing also incurs some overhead
- We can avoid almost all overhead and win the tradeoff:

```
sub AUTOLOAD {
    # ... as before; set up $method ...

    my $code = q{
        sub {
            my ($self) = @_;
            my $val = $self->{METHODNAME};
            $self->{METHODNAME} = shift if @_;
            $val;
        };
    };

    $code =~ s/METHODNAME/$method/g;

    *$AUTOLOAD = eval $code;
    goto &$AUTOLOAD;
}

```



- The first time, it constructs and compiles the code for the method
- Second time, the method is called directly with no `AUTOLOAD`
- No overhead!
- There's that magic glob again.

Generating Functions Dynamically

- In this example, compiling the code repeatedly is a waste of time
- Only one variable changes in each accessor
- Perl can construct functions that share code without recompiling

```
sub AUTOLOAD {
    # ... as before; set up $method ...

    my $code = sub {
        my ($self) = @_;
        my $val = $self->{$method};
        $self->{$method} = shift if @_;
        $val;
    };

    *$AUTOLOAD = $code;
    goto &$AUTOLOAD;
}

```



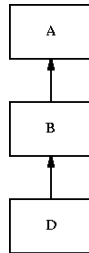
- The new method is a *closure*
- It refers to a private variable, `$method`
- When `AUTOLOAD` returns, only the new method has a reference to `$method`
- The new methods all share code, but each has its own private `$method` variable
- Method code is compiled only once, along with the rest of your program

NEXT.pm

- Consider this class inheritance structure:
- Now consider `D::DESTROY`
 - We would like `D::DESTROY` to call `B::DESTROY` (if there is one)
 - Moreover `B::DESTROY` should call `A::DESTROY` (ditto)
- We can accomplish that this way:

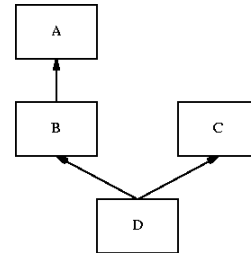
```
# Package D
sub DESTROY {
  my $self = shift;
  # Do various destructions here
  $self->SUPER::DESTROY;
}
```

- `SUPER::DESTROY` means "call DESTROY"
- If `B::DESTROY` also calls `SUPER::DESTROY`, everything works as it should

**NEXT.pm**

```
$self->SUPER::DESTROY;
```

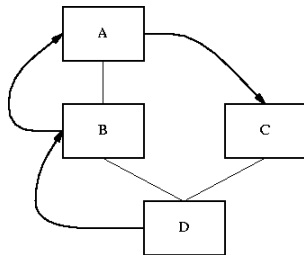
- But what if `$self` has more than one base class?
 - Which `DESTROY` method is called?
 - Only the first one, it turns out.



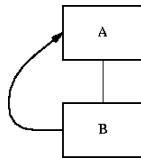
- Suppose each `DESTROY` calls `SUPER::DESTROY`
 - Unfortunately, that's not enough
 - `D::DESTROY` calls `B::DESTROY`
 - `B::DESTROY` calls `A::DESTROY`
 - `C::DESTROY` is never called

NEXT.pm

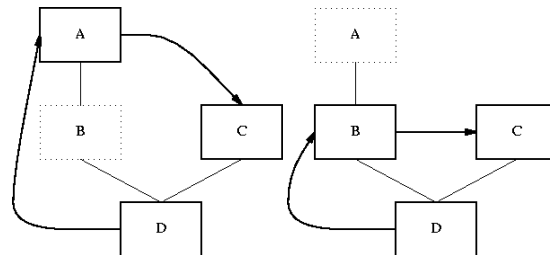
- `NEXT` is the solution to this problem
- Each method calls `->NEXT::method`
- This magically redispaches to the correct 'next' method
- For example:



- Note that when `$self->NEXT::method` in class A depends on the class of `$self`
 - If `$self` is a D, then A dispatches to C
 - If `$self` is a B, then A dispatches nowhere
- Damian Conway invented this

**NEXT.pm**

- If some of the methods are missing, `NEXT` figures that out:



- But how does this all work?
 - `NEXT` is essentially a big fat `AUTOLOAD`
 - `->NEXT::method` wants to call `NEXT::method`
 - But there isn't one, so it calls `NEXT::AUTOLOAD` instead
 - `NEXT::AUTOLOAD` examines the inheritance hierarchy
 - Figures out the correct 'next' method
 - Jumps there with magic `goto`

NEXT.pm

- Here's a simplified version
- First, a utility function:

```
sub class_structure {
    my $start = shift;
    my $prev;

    my @todo = ($start);
    my %next;
    while (@todo) {
        my $cur = shift @todo;
        $next{$prev} = $cur if defined $prev;
        unshift @todo, @{$cur\::ISA};
        $prev = $cur;
    }
    \%next;
}
```

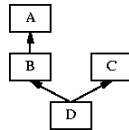
- Given a class name, this returns a hash of 'next' classes

- Given D it returns

```
{ D => B, B => A, A => C, C => undef }
```

- Given B it returns

```
{ B => A, A => undef }
```

**NEXT.pm**

```
sub AUTOLOAD {
    my ($self_class) = ref $_[0] || $_[0];
    my $cs = class_structure($self_class);

    my $caller_class = caller;
    my $next_class = $caller_class;

    my (undef, $method) = ($NEXT::AUTOLOAD =~ /(.*):(.*)/);

    do {
        $next_class = $cs->{$next_class};
    } while defined $next_class
        && not defined &{"$next_class\::$method"} ;

    if (defined $next_class) {
        goto &{"$next_class\::$method"};
    } else {
        return;
    }
}
```

- We find out the class that the target object is in (\$self_class)

- We get the next-class table for that class (\$cs)

- We figure out where we were called from (\$caller_class)

- We scan \$cs until we find a new class that has the method we want

- Then we go to it

NEXT.pm

- That version returns silently if there is no 'next' method
- You might like it to die instead
- For example, an AUTOLOAD might decide it's not prepared to emulate a certain function
- It wants to delegate control to the next AUTOLOAD, which might handle it
- But if there are no more AUTOLOADS, it should croak, since nobody will handle it
- \$self->NEXT::ACTUAL::method(...) will croak if there is no 'next' method
- The code is simple

NEXT.pm

- \$self->NEXT::ACTUAL::method(...) will croak if there is no 'next' method

```
sub AUTOLOAD {
    my ($self_class) = ref $_[0] || $_[0];
    my $cs = class_structure($self_class);

    my $caller_class = caller;
    my $next_class = $caller_class;

    my ($my_class, $method) =
        ($NEXT::AUTOLOAD =~ /(.*):(.*)/);

    do {
        $next_class = $cs->{$next_class};
    } while defined $next_class
        && not defined &{"$next_class\::$method"} ;

    if (defined $next_class) {
        goto &{"$next_class\::$method"};
    } else {
        croak qq{Can't locate object method "$meth"
                via package "$self_class"};
        if $my_class eq 'NEXT::ACTUAL';
        return;
    }
}

@NEXT::ACTUAL::ISA = ('NEXT');
```

Shell.pm

- One final hack:

```
sub AUTOLOAD {
    my ($pack, $func) = ($AUTOLOAD =~ /(.*)::(.*)/);
    qx{$func @_};
}
```

- Now you can write Perl programs that look like shell scripts:

```
$passwd = cat("</etc/passwd");
print $passwd;

sub ps;
print ps -ww;

cp("/etc/passwd", "/tmp/passwd");
```

- This is due to Larry Wall
- I omitted a lot of details here
- See `Shell.pm` for the actual implementation

Cantrips



Returning a False Value

```
sub foo {
    ...
    return undef;          # False
}

if (@result = foo(...)) { ... }
```

- Oops, `undef` is *not* false in a list context!
- `@result` has one element, which is `undef`

```
if (@result) { ... }      # Yes!
```

Returning a False Value

- Solution:

```
sub foo {
    ...
    return;
}
```

- Returns `undef` in scalar context.
- Returns empty list in list context.

The Self-Replacing Stub

- We've already seen

```
...
require Carp;
Carp::croak(...);
...
```

as a way to defer loading of a module until it's needed.

- Alternative: use AUTOLOAD

```
sub AUTOLOAD {
  if ($AUTOLOAD =~ /::croak$/) {
    require Carp;
    goto &Carp::croak;
  }
}
```

The Self-Replacing Stub

- Here's another way:

```
sub croak {
  require Carp;

  *croak = \&Carp::croak;
  goto &croak;
}
```



- There's that magic glob again.

- Also magic goto
- But also see autouse.pm

Schwartzian Transform

- Sort list of items by some *non-apparent* feature
- Example: Sort filenames by last-modified date
- Obvious method is very wasteful:

```
sort { -M $b <=> -M $a } (readdir D);
```

- Calls `-M` over and over on the same files

- Another idea:

1. Construct data structure with both names and dates
2. Sort by date
3. Throw away dates

Schwartzian Transform

```
@names = readdir D;
```

name1	name2	name3	...
-------	-------	-------	-----

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

NAME	name1	NAME	name2	NAME	name3	...
DATE	date7	DATE	date3	DATE	date4	

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

NAME	name6	NAME	name8	NAME	name2	...
DATE	date1	DATE	date2	DATE	date3	

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

name6	name8	name2	...
-------	-------	-------	-----

Schwartzian Transform

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

- Caveat: Do not optimize without benchmarking!

User	System	Total	
5.11	+	6.83	= 11.94
7.37	+	0.82	= 8.19
			Naive sort
			Schwartzian transform

- Donald E. Knuth (famous wizard) says (quoting R. W. Floyd):

Premature optimization is the root of all evil.



Schwartzian Transform

Well-known to Unix shell programmers:

```
# 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}'
```



Debug Printing of Strings

```
if (/carrots$/) { die }
```

- But it didn't die! Why not?
- Try the debugger:

```
DB<119> p $_;
I like carrots
```

- Pull your hair out.
- Or, instead:

```
DB<119> p "<$_>";
<I like carrots >
```

- Oho.
- The *terminal program* should have taken care of this!



Debug Printing of Lists

```
@t = ('x', ' ', '=', ' ', '3.4', '&', 'y', ' ', '=', ' ', '5')
```

- Now `print @t` yields

```
x = 3.4& y=5
```

- Hard to tell what the list elements are!

- `print "@t"` is even worse!

```
x = 3.4 & y = 5
```

- Solution:

```
$" = ')((';
print "@t";

(x) ( ) (=) ( ) (3.4) (& ) (y) ( ) (=) ( ) (5)
```



? : ? : ? :

- Most folks know about the `? :` operator

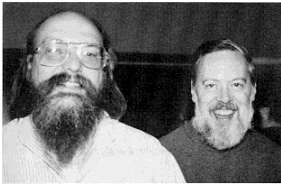
```
*{$p . ' : ' . $n} = (ref $v ? $v : \v);
```

- It's a compact version of an `if-else` block
- What if you want a compact version of an `if-elsif-else` block?

```
sub sign {
  my $x = shift;
  if ($x < 0) { return -1 }
  elsif ($x == 0) { return 0 }
  else { return +1 }
}
```

- No problem:

```
sub sign {
  $_[0] < 0 ? -1 :
  $_[0] == 0 ? 0 :
  1 ;
}
```



- Everything is as it should be
- The precedence is fine, the short-circuiting is fine
- The folks who designed the `? :` operator are very smart
- So chain together as many as you want

Local Effects

- `local` confines a change to a block

- We saw:

```
{ local *F = \&VeryLongName::SomeFunction;
  F(...);
}
{ local $| = 1;
  $rc = print $self "";
}
{ local $/; $code = <$fh> }
```

- Wouldn't it be nice to be able to do this:

```
{ local chdir $DIR;
  ...
}
# Old directory is restored here
```

Boolean numbers

```
sub delete_files {
  my ($dir) = @_;
  opendir my $dh, $dir or return;
  my $deleted = "0e0";
  for (readdir $dh) { ++$deleted if unlink }
  return $deleted;
}

unless (delete_files(...)) { die... }

$num_deleted = delete_files(...);
```

- Function only returns false on an error

- Even when it returns 0, it returns true

- "0e0" is zero, but true

- Also "0 but true" return from `ioctl`

- DBI uses a similar trick

Local Effects

- Here's the idea:

```
{ my $temporary = LocalChdir->chdir_to($DIR);
  ...
}
```

- When control exits the block, `$temporary` will be destroyed

- We can rig up `LocalChdir::DESTROY` to move back to the old directory

```
package LocalChdir;
use Cwd;

sub chdir_to {
  my ($package, $new_dir) = @_;
  my $old_dir = cwd();
  return unless chdir($new_dir);
  bless { DIR => $old_dir } => $package;
}

sub DESTROY {
  my $dir = $_[0]{DIR};
  chdir($dir)
  or croak("Couldn't return to '$dir' on block exit: $!");
}
```

Local Effects

- This trick is widely used:

```
use SelectSaver;
{
  my $saver = SelectSaver->(FH);
  # FH is selected
}
# old handle is selected
```

- Or:

```
use Hook::LexWrap;
{
  my $temporarily = wrap 'myfunction',
    post => sub { print "[post:@_]\n" },
    pre  => sub { print "[pre:@_]\n " };
  # Function is wrapped
}
# Function is no longer wrapped
```

Selecting n Different Things

```
while (keys %h < $n) {
  ${select_thing()}++;
}
@things = keys %h;
```

- In scalar context, `keys %h` is super-efficient.
- No, it does not count the keys one at a time.

Dinner Time!



Thanks very much for attending my class

The evaluation form is at

<http://perl.plover.com/class/eval.cgi>

Or you can send me mail with questions or comments whenever you like

Other Resources

- *Perl Cookbook*, Christiansen and Torkington. O'Reilly and Associates.
- *Perl Paraphernalia* web site. <http://perl.plover.com/>
- *Object-Oriented Perl*, Damian Conway. Manning Publications.
- *Advanced Perl Programming* (2nd Edition), Simon Cozens. O'Reilly and Associates.
- Perl 6 development web site. <http://dev.perl.org/perl6/>

Bonus Slides Not in the Talk Anymore

- Talks evolve over the years
- Things move in, other things move out
- I still have the slides for the stuff that moved out
- You might as well see them if you're interested

Biographical Note

- I first did this class in 1999
- It used to say:

Disclaimer

I am not personally a wizard.



Biographical Note

- But last year at YAPC Larry said he thought I *was* a wizard
- Says Larry:

"One of the benefits of Perl culture is that anyone can become a wizard regardless of age, race, gender, or programming ability."

Making Things Appear to Be What They're Not

Part III: Overloading



(Eliminated summer 2000 in favor of Autoloading)

Overloading Overview

- In *overloading*, you redefine the effect of the standard Perl operators like + and . to have a special meaning for objects in a certain class.
- Operator applications are transformed into method calls.
- Syntax:

```
package MyClass;
use overload '+' => \&myadd,
             '-' => \&mysubtract,
             ...
             ;
```

- Now `$obj1 - $x` turns into

```
$obj1->mysubtract($x);
```



Overload Method Call Summary

- Argument 1 is always an object of the appropriate class, as with any method
- On two objects of the same type, you get the objects in the same order:

<code>\$obj1 - \$obj2</code>	<code>mysubtract(\$obj1, \$obj2);</code>
<code>\$obj2 - \$obj1</code>	<code>mysubtract(\$obj2, \$obj1);</code>
- When operating on an overloaded object and an unoverloaded value, the object is *always* the first argument:

<code>\$obj1 - \$x</code>	<code>mysubtract(\$obj1, \$x);</code>
<code>\$x - \$obj1</code>	<code>mysubtract(\$obj1, \$x, 1);</code>

- On two overloaded objects of different types, the left-hand argument determines whose method will be called:

<code>\$obj1 - \$OBJX</code>	<code>mysubtract(\$obj1, \$OBJX);</code>
<code>\$OBJX - \$obj1</code>	<code>Xcombine(\$OBJX, \$obj1);</code>



Overloading: Normal uses

- `BigInt`, `BigFloat`, `Complex`, etc.
- `Vectors`, `Bit::Vector`, etc.
- I tried to think of more, but actually overloading is overrated.



Overloading: Example

```
package Vector3;
use Carp;

use overload '+' => \&add,
             '*' => \&dotproduct,
             'x' => \&crossproduct,
             ;

sub new {
    my $package = shift;
    $package = ref $package || $package;
    croak "Usage: new(x,y,z)" unless @_ == 3;
    my %self;
    @self{'X','Y','Z'} = @_;
    bless \%self => $package;
}

...
```



Overloading: Example

```
sub add {
    my ($vec1, $vec2) = @_;
    unless (ref $vec1 && $vec1->isa('Vector3')
        && ref $vec2 && $vec2->isa('Vector3')) {
        croak "Invalid vector addition";
    }
    $vec1->new(map {$vec1->{$_} + $vec2->{$_}} qw(X Y Z));
}
```



Overloading: Example

```
sub dotproduct {
    my ($vec1, $vec2, $rev) = @_;

    if (ref $vec2 && $vec2->isa('Vector3')) {
        my $dp = 0;
        for (qw(X Y Z)) {
            $dp += $vec1->{$_} * $vec2->{$_};
        }
        return $dp;
    }
    elsif (! defined ref $vec2) { # It's a scalar
        return $vec1->new(map {$vec2 * $vec1->{$_}} qw(X Y Z));
    }
    else {
        croak "Invalid vector scalar multiplication";
    }
}

sub crossproduct {
    ...
}
```



Overloading: Bizarre Example

- We're going to detect Y2K bugs.
- Perl localtime function is very badly designed.

```
(..., $year, ...) = localtime(...);

$q = "$mon/$day/$year"; # wrong
$q = "$mon/$day/" . sprintf('%02d', $year % 100); # RIGHT

print "The year is 19$year.\n"; # wrong
print "The year is 19" . $year . ".\n"; # wrong
print "The year is ", 1900+$year, ".\n"; # RIGHT
```



Overloading: Y2K Detection Example

- Strategy:
 - Override localtime to call our fake localtime function
 - Our function will return the usual values, except...
 - The year item will be a special object...
 - Which will be overloaded to call carp if it is concatenated with "19"



Overloading: Y2K Detection Example

```
package y2k;
use Carp;
use overload '.,' => \&concat,
             '0+' => \&to_num,
             ;

sub import {
    my $caller = caller;
    *{$caller . '::localtime'} = \&fake_localtime;
    *{$caller . '::gmtime'} = \&fake_gmtime;
}
```

- There's that magic glob again.



Overloading: Y2K Detection Example

```
package y2k;
...
sub fake_localtime {
    unless (wantarray) {
        return @_ ? localtime(@_) : localtime();
    }
    my @lt = @_ ? localtime(@_) : localtime();
    $lt[5] = { YEAR => $lt[5] };
    bless $lt[5] => 'y2k';
    @lt;
}
```

This product is
Y2K
compliant

Overloading: Y2K Detection Example

```
package y2k;
...
sub to_num {
    my ($year) = @_;
    return $year->{YEAR};
}

sub concat {
    my ($y2k, $s, $rev) = @_;
    carp("Detected possible Y2K problem");
    my $year = sprintf("%02d", $y2k->{YEAR} % 100);
    $rev ? $s . $year : $year . $s;
}
```

- Or use Syslog instead of carp.

Overloading: Y2K Detection Example

- Program now croaks on \$year % 100, \$year + 1900, etc.
- One solution: Just add modulus, addition, etc. methods.
- Another solution:

```
package y2k;
...
use overload 'nomethod' => \&default;
...
sub default {
    my ($y2k, $arg, $rev, $op) = @_;
    my $y = $y2k->{YEAR};
    my $expr = $rev ? "$arg $op $y" : "$y $op $arg";
    eval $expr;
}
```

Overloading: Y2K Detection Example

- Another solution uses a *dispatch table*:

```
{ my ($year, $arg);
  %methods = (
    '+' => sub { $year + $arg },
    '%' => sub { $year % $arg },
    'r%' => sub { $arg % $year },
    ...
  );
  sub default {
    my ($y2k, $a, $rev, $op) = @_;
    my $code = $rev
      ? ($methods{"r$op"} || $methods{$op})
      : $methods{$op};
    croak "No method defined for y2k object for operation '$op'
    unless $code;
    $arg = $a;
    $year = $y2k->{YEAR};
    &$code;
  }
}
```



Big Techniques



(Eliminated in favor of autoloading)

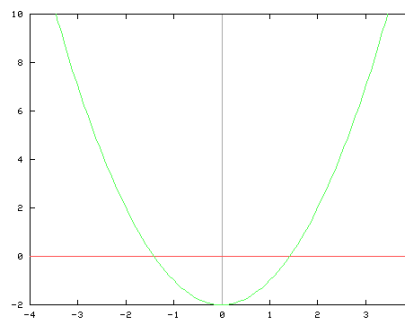


Big Technique #1: Newton-Raphson Method

- Almost everyone has to work with numbers
- Numerical computation techniques are an entire field
- Most techniques are special-purpose
- This one is an exception



`sqrt()`



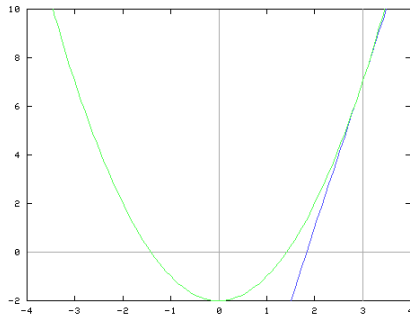
- How does the Perl `sqrt()` function work?
- Probably uses something like the *Newton-Raphson Method*
- To compute `sqrt(2)`, we need to solve the equation

$$x^2 - 2 = 0$$

- That's where this parabola crosses the x axis



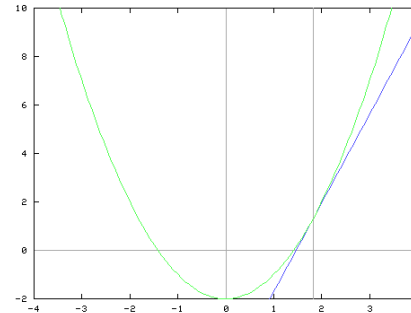
Newton-Raphson Method



- Guess a solution (yes, just guess!)
- Tangent line through that point will locate a better guess
- Repeat as desired

- Repeat as desired

Newton-Raphson Method



- Math and code turn out to be very simple in general
- Even simpler for this particular case

Square Roots with Newton-Raphson Method

```
sub square_root {
  my ($n, $e) = @_;
  $e ||= 0.00001;
  my $guess = $n; # Yes, just guess!

  while (abs(($guess*$guess - $n)/$n) > $e) {
    $guess = ($guess + $n/$guess)/2;
  }
  return $guess;
}
```

- That's all!

Solve Any Equation With Newton-Raphson

- Suppose you want to solve $f(x) = N$
- Step 1: Compute the derivative $d(x)$
 - Or get the math expert down the hall to do it
- Then the answer is:

```
sub solve {
  my ($N) = @_;
  my $g = 1; # Substitute a reasonable guess here
  until (the guess is good enough) {
    $g -= (f($g) - $N) / d($g);
  }
  return $g;
}
```

- Warnings:
 - Initial guess must be reasonable
 - Method doesn't *always* work

Solve Any Equation With Newton-Raphson

- Example: Financial computations
- A principal P invested for time N at rate of return i grows to:

$$F = P * (1+i)^{N}$$

- Question: How long before I have a million dollars?
- (Given F , P , and i , compute N)
- The math expert down the hall says that the derivative is

$$P * (1+i)^{N} * \log(1+i)$$

- Looks nasty, but that's OK, just plug it in

Newton-Raphson: Financial Computations

```
sub how_long {
  my ($P, $i, $F) = @_;
  my $g = 1; # Initial guess
  my $d = 1;
  until ($d/$g < 0.000001) {
    $d = ($P * (1+$i)**$g - $F)
        / ($P * (1+$i)**$g * log(1+$i));
    $g -= $d;
  }
  return $g;
}
```

- Example: `how_long(10000, 0.065, 1000000)` is 73.1271913100701
- \$10,000 invested at 6.5% interest becomes \$1,000,000 after 73.12 years

Big Technique #2: Caching

- Makes programs faster
- Exchanges space for time
- When a cached function is called, its return value is saved
- When called again with same arguments, the saved value is returned

Caching

- Very commonly used
- Example: Your local DNS server caches the responses it gets from other DNS servers
- Another example: Converting RGB values to CMYK values:

```
sub cmyk {
  my ($r, $g, $b) = @_;
  my ($c, $m, $y) = (1-$r, 1-$g, 1-$b);
  my $k = $c < $m ? ($c < $y ? $c : $y)
                : ($m < $y ? $m : $y); # Minimum
  for ($c, $m, $y) { $_ -= $k }
  [$c, $m, $y, $k];
}
```

- Many image formats (including GIF) have many pixels that are the same color.
- This recomputes the same CMYK values over and over.

Caching

- Faster version:

```
{
  my %cmyk;

  sub cmyk {
    my $key = join ' ', @_ ;
    return $cmyk{$key} if exists $cmyk{$key};
    $cmyk{$key} = real_cmyk(@_);
  }

  sub real_cmyk {
    # as before ...
  }
}
```

Memoizing

- *Memoizing* is the process of converting a function to use caching.
- It can be done *automatically*
- Here's how you do it:


```
use Memoize;
memoize 'cmyk';

sub cmyk { ... as before ... }
```
- That's all!
- I'd love to tell you all about the internals, but we don't have time
- You can read my TPJ article about it on my web site.

<http://perl.plover.com/Memoize/>

Memoizing

- Memoizing is a really useful tool to have in your toolbox
- Program too slow? Try sprinkling in a little memoization. It's cheap and easy.
- Need to profile? Try memoizing. If it works, rewrite the function you memoized; if not, try another function.
- Worried about recursion inefficiencies? Memoization is often a cheap and effective alternative to rewriting in iterative style.
- Continued...

Memoizing

- Memoize slow functions like `gethostbyname`.
- Memoize to a permanent database and speed up your function *forever*.
- Same technique can be adapted to make a simple profiler
 - Or call counter
 - Or call-graph generator
 - See Philippe Verdret's `Hook::PrePostCall` module
- Memoization is *impossible* in C

Big Technique #3: Iterators

- An iterator is an object interface to a list
 - Supports a 'next' operation to generate the next item when it is needed
- Why?
 - The list might be enormous
 - Might take a long time to come up with list elements
 - You don't know in advance how many you will want
 - You can pass the object around so it can be used by anyone who needs it



Iterators

- This is not new: A filehandle is an iterator!
 - It encapsulates a list of strings (the lines)
 - The <...> operation requests the next string
 - Other examples: each, readdir(), glob



Iterator Example

- Suppose you want to generate strings of a certain form
- I got this example from a biologist
- He wanted "AT(GC)A(TA)" to become ATGAT, ATGAA, ATCAT, ATCAA.
- He had built a recursive subroutine to generate all the strings of a given form
- It took a long time to run and generated a uselessly large list
- Iterators are a better solution
- As an example, will expand "foo-#-bar#" instead:
 - foo-0-bar0
 - foo-0-bar1
 - ...
 - foo-9-bar8
 - foo-9-bar9



Iterator Example

```
sub make_iterator {
    my @tokens = split /(\\#)/, shift();
    my $n_digits = grep {$_ eq '#'} @tokens;
    my $digits = '0' x $n_digits;

    return sub {
        my $result;
        my $d = 0;
        for my $t (@tokens) {
            if ($t eq '#') {
                $result .= substr($digits, $d++, 1);
            } else {
                $result .= $t;
            }
        }

        $digits++;
        if (length $digits > $n_digits) { # Overflow?
            $digits = '0' x $n_digits; # Reset
        }

        return $result;
    };
}
```

- Anonymous subroutine is a *closure*
- my variables are *captured* by the closure
- Each call to make_iterator constructs a new closure with new private state variables



Iterator Example

```
my $it = make_iterator('foo-#-bar#');

for (1..105) {
    my $s = $it->();
    print "$s\n";
}

# Prints foo-0-bar0, foo-0-bar1,
#      ...,
#      foo-9-bar8, foo-9-bar9, ...
```

- Easy change to make it stop and return undef instead of starting over.
- Construct many iterators that all operate independently.
- Pass iterators to functions, store in data structures.

Iterator Operations

- An iterator is just as good as a list:

```
while (defined ($item = $iterator->())) {
    # Do something with this $item
}
```

Iterator Operations

- If the iterator returns the list items in some canonical order, you can do this:

```
sub both {
    my ($it1, $it2) = @_;
    my ($a, $b) = ($it1->(), $it2->());

    sub {
        return undef unless defined $a || defined $b;
        my $rv;
        if ($a lt $b || ! defined $b) {
            $rv = $a;
            $a = $it1->();
        } if ($b lt $a || ! defined $a) {
            $rv = $b;
            $b = $it2->();
        } else {
            # $a eq $b
            $rv = $a;
            ($a, $b) = ($it1->(), $it2->());
        }
        return $rv;
    }
}
```

- This function works for *any* iterators that return items in alphabetical order
- If an iterator represents a database query, this is the *OR* operation

More Applications of Iterators

- Database lookups can return an iterator that generates solutions on demand
- Tree searches can return an iterator that generates solutions on demand
- Search functions of any sort can ..

- **Important note:**

- This is just a technique for saving the state of a partially-completed function...
- ...and restarting it later
- Not usually considered an easy thing to do!

Big Technique #4: State Machines

- In a *state machine*, the program tracks a ‘current state’
- It has a table that says, for each possible state and each possible input type:
 - What state to be in next
 - An action to perform
- For example, suppose we’re writing an NNTP server:
 - In state START:
 - call `&say_hello, goto MAIN`
 - In state MAIN:
 - GROUP command: call `&cmd_group, goto MAIN`
 - QUIT command: call `&cmd_quit` and `goto ...`
 - POST command: call `&cmd_post, goto HEADER`
 - ...
 - In state HEADER:
 - Blank line: call `&article_save_header, goto BODY`
 - `..`: goto `ARTICLE_FINISH`
 - Other: store line, `goto HEADER`
 - In state BODY:
 - `..`: goto `ARTICLE_FINISH`
 - Other: store line, `goto BODY`
 - In state ARTICLE_FINISH:
 - call `&article_check_and_post, goto MAIN`

State Machines For NNTP

```
%machine = (
  START => { DEFAULT=> [ \&say_hello, MAIN, ],
  MAIN => { group => [ \&cmd_group, MAIN],
            quit => [ \&cmd_quit, MAIN],
            post => [ \&cmd_post, HEADER],
            ...
  },
  HEADER => { BLANK => [ \&article_save_header, BODY],
             DOT => [ undef, ARTICLE_FINISH
             DEFAULT=> [ \&store_line, HEADER ],
             ...
  },
);
```

- Associated with each state is a *transition table*
- Keys in transition table represent *input conditions*

Implementing State Machines in Perl

- The easiest way is with a hash table.
- The keys are the state names.
- The values have:
 - The action to perform (a coderef)
 - The name of the next state
 - Other information if appropriate

State Machines For NNTP

```
%machine = (
  START => { DEFAULT=> [ \&say_hello, MAIN, ], },
  MAIN => { group => [ \&cmd_group, MAIN],
            ... } ... );

BEGIN { $STATE = 'START' }

sub run_machine {
  for (;;) {
    my $t_table = $machine{$STATE};
    my ($input, @args) = get_input();
    my ($action, $next_state) =
      @{$t_table->{$input} || $t_table->{DEFAULT}};
    unless defined($next_state) {
      die "No transition defined for state $STATE, input $input";
    }
    $action->(@args) if $action;
    $STATE = $next_state;
  }
}
```

State Machines Are Very Easy to Read!

```
sub run_machine {
  for (;;) {
    my $t_table = $machine{$STATE};
    my ($input, @args) = get_input();
    my ($action, $next_state) =
      @{$t_table->{$input} || $t_table->{DEFAULT}};
    unless defined($next_state) {
      die "No transition defined for state $STATE, input $input";
    }
    $action->(@args) if $action;
    $STATE = $next_state;
  }
}
```

- For something as complicated as NNTP, this is very simple code!
- All the details are in the table, which is tidy and compact
- Brian Kernighan (noted wizard) says:

Capture regularity with code, irregularity with data.

Big Technique #5: Building a Replacement Debugger

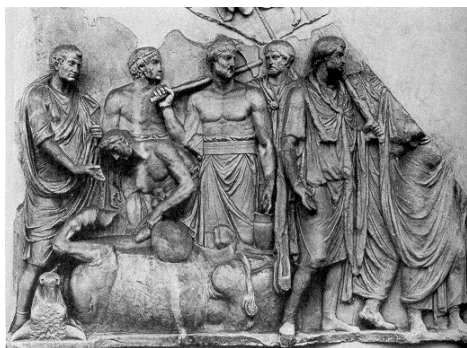
- There's nothing special about the perl debugger
- It's just another module
- When you run `perl -d ...` it loads `perl5db.pl`
- Code in `perl5db.pl` is enlightening

Why Build a Replacement Debugger?

- Obvious tactic: Copy `perl5db.pl`, modify slightly, use.
- But there are some non-obvious tactics
- The debugger isn't *just* an ordinary module
- In debug mode, Perl enables special features
- To use: Name the module `Devel::Something`
- Run with `perl -d:Something` to automatically load

Debugger Features

- Lots of functions for haruspication
- See `perldebguts` (or `perldebug`) for fullest details



- `@{::_<foo.pl}` contains the source code of `foo.pl`
- `%{::_<foo.pl}` contains breakpoints and actions
- `%DB::sub` contains subroutine start-end information
- `DB::DB()` is called before each executed line
- `caller()` returns current package, filename, line as usual, also sets `@DB::args`

Trivial Debugger

```
package Devel::Count;
sub DB::DB { ++$count }
END { print "Total statements: $count\n" }
```

- Now `perl -d:Count program.pl` prints out:

```
Total statements: 286
```



Trace Execution

- Occasionally-asked question:
- "How can I emulate the behavior of the Bourne shell `-x` option?"
- Here's one way:

```
package Devel::Trace;
sub DB::DB {
  my ($p, $f, $l) = caller;
  my $code = \@{":_<$f"};
  print STDERR ">> $f($l) $code->[$l]";
}
```

- Now `perl -d:Trace sample.pl` prints out:

```
>> sample.pl(1) for (1 .. ($ARGV[0] || 12)) {
>> sample.pl(2)  next unless $_ % 12;
>> sample.pl(3)  print "";
>> sample.pl(1) for (1 .. ($ARGV[0] || 12)) {
(etc.)
```



Examine Source Code

```
package Devel::Dumpcode;
sub DB::DB { } # Do nothing special
sub main::source_of_function {
  my $package = caller;
  $function = $package . '::_' . shift();
  my ($file, $start, $end) =
    $DB::sub{$function} =~ /(.*):(\d+)-(\d+)/;
  @{":_<$file"}[$start..$end];
}
```

- Now the program can do

```
print source_of_function('foo')
```

to print out the source of function `foo`

- Print code to file, invoke editor, reload, eval



Simple Profiler

- `Devel::DProf` is complicated and hard to use
- But building a simple profiler is *easy*

```
package Devel::Profile;
sub DB::DB {
  my ($package, $file) = caller();
  my ($subroutine) = (caller(1))[3];
  return if $subroutine eq '(eval)';
  $subroutine = "<$file>" unless defined $subroutine;
  ++$count{$subroutine};
}
END {
  for $subr (sort { $count{$b} <=> $count{$a} } (keys %count)) {
    printf STDERR "%8d %s\n", $count{$subr}, $subr;
  }
}
```

- Output:

```
798 main::page
66 </usr/local/bin/perldoc>
57 Exporter::import
49 main::check_file
39 main::minusf_nocase
(etc.)
```



Simple Coverage Analyzer

```
package Devel::Coverage;

sub DB::DB {
    my ($package, $file, $line) = caller();
    $files{$file} = 1;
    $covered{$file}{$line} = 1;
}

END {
    for my $file (keys %files) {
        my $array = \@{"::<$file"};
        my ($executable, $covered) = (0, 0);
        for my $line (1 .. $#$array) {
            next if $array->[$line] == 0;
            $executable += 1;
            $covered += $covered{$file}{$line};
        }
        printf STDERR "%4d/%4d (%3.0f%%) covered in %s.\n",
            $covered, $executable, 100*$covered/$executable, $file
            unless $executable == 0;
    }
}
```

- In numeric context, @{"::<foo"} elements are special
- They are equal to zero only when the line is not executable

```
10/ 31 ( 32%) covered in /usr/local/lib/perl5/5.6.0/Exporter.
10/ 70 ( 14%) covered in /usr/local/lib/perl5/5.6.0/Getopt/Std.
 8/ 12 ( 67%) covered in /tmp/Devel/Coverage.pm.
60/ 74 ( 81%) covered in ./MAKE_SLIDES.
```

Tokenizing

- In C, you use programs like `lex` to convert a description of the legal tokens into a tokenizer program.
- Or you write a program to read the input character-by-character and run a state machine
- That is not very Perl-like.
- It is also not very efficient.



Big Technique #6: Tokenizing

- *Tokens* are the basic syntactically meaningful portions of an input.
- For example, in

```
print 12+$var;
```

- The tokens are `print`, `12`, `+`, `$`, `var`, and `;`
- Individual characters are not generally meaningful.
- *Tokenizing* is the act of converting a character stream into a token stream.
- Also called *lexing*

Tokenizing

- A regex is *already* a program for reading data character-by-character and running a state machine
- Let's write a lexer for a calculator. It has the following tokens:
 - `+`, `-`, `*`, `/`, `^`, `**`, `(`, `)`, `=`
 - `:=`
 - Variable names: `Value2`, for example
 - Numbers with optional decimal points and scientific notation
 - Whitespace will be ignored except where it separates tokens

Tokenizing

- Our trick:

```
split /(a+)/, $string
```

- This breaks `$string` into pieces which alternate between
 - Strings of a's
 - The other stuff that was between the a's
- Note special `split` meaning of (capturing parentheses).



Tokenizing

- The tokenizer:

```
sub tokens {
  my @tokens =
    split m{
      \*\* | := # ** or := operator
      | [-+*/^()=] # some other operator
      | [A-Za-z]\w+ # Identifier
      | \d*\.\d+(?:[Ee]\d+)? # Decimal number
      | \d+ # Integer
    }x, shift();
  grep /\S/, @tokens;
}
```

- Easy to understand and to change, efficient, predictable.
- Behaves very much like similar `lex`-generated parsers



Tokenizing

- We can get rid of that `grep`:

```
sub tokens {
  split m{
    \*\* | := # ** or := operator
    | [-+*/^()=] # some other operator
    | [A-Za-z]\w+ # Identifier
    | \d*\.\d+(?:[Ee]\d+)? # Decimal number
    | \d+ # Integer
  }x, shift();
}
```

- (Thanks to Andy Wardley.)



Exportation (Inheritable Method)

- This exporter can be inherited by subclasses of `Rings`:

```
package Rings;
use Carp;

%exports = map {$_ => 1} qw(Narya Nenia Vilya);

sub import {
  my $caller = caller;
  my $package = shift;
  my %exported = \%{$package . '::exports'};
  for my $name (@_) {
    unless (%exported->{$name}) {
      croak("Module $package does not export &$name; aborting")
    }
    *{$caller . '::' . $name} = \%{$package . '::' . $name};
  }
}
```



Aliasing

```
my $exported = \%{$me . '::exports'};
... $exported->{$name} ...
```

- That worked well enough, but here's a better trick

```
local *exported = \%{$me . '::exports'};
```

- Now `$exported` is the hash.

```
... $exported{$name} ...
```

- You want the `local` so the change is confined to `import`
- You can't my a glob.

Aliasing

This is how Sarathy's clever `Alias` module works.

A typical object:

```
{ SALARY => 45_000, Children => ['Ishmael', 'Isaac'] }
```

A typical method:

```
sub method {
    my $self = attr shift;          # Alias::attr
    $$SALARY *= 1.06;               # Raise salary 6%
    print "You have lovely children, named @Children.\n";
    pop @Children;                  # Pay the price for that 6%
}
```

Another Tied Hash: %!

- Perl magic `%! variable` reflects the operating system error status

- Example of use:

```
unless (open FH, $filename) {
    if ($! == EACCESS) {
        # Permission denied...
    } elsif ($! == ENOENT) {
        # No such file...
    } elsif ($! == ENOTDIR) {
        # Some part of the path is not a directory...
    } elsif ...
}
```

- This doesn't work--where did `EACCESS` etc. come from?
- Solution 1: Import lots and lots of compile-time constants. (Bleccch.)
- Solution 2: Use `%! instead`: (5.005 and later.)

```
unless (open FH, $filename) {
    if (!$!{EACCESS}) {
        # Permission denied...
    } elsif (!$!{ENOENT}) {
        # No such file...
    } elsif (!$!{ENOTDIR}) {
        # Some part of the path is not a directory...
    } elsif ...
}
```

- When Perl saw you use `%! instead`, it loaded the `Errno` module and tied `%! instead` into it.
- `FETCH` method checks the value of `%! instead`.

%! Implementation

```
package Errno;

sub ENOENT () { 2 }
sub EACCESS () { 13 }
sub ENOTDIR () { 20 }
# ... many more ...

sub TIEHASH { bless [] }          # Dummy object

sub FETCH {
    my ($self, $errname) = @_;
    return $! == &$errname;
}

sub STORE {
    croak("ERRNO hash is read only!");
}
```

- This was invented by Tom Christiansen and implemented by Graham Barr.

Bizarre Tricks

- A fruitful source of ideas is to ask:

“What can I tie today?”

- Then if you get an answer, you learn something new and interesting.
- For example: “I know! Let’s tie `$_!`”



Bizarre tie Tricks: no underscore

- Theory: People find implicit use of `$_` confusing
- Sometimes, it’s a genuine error, as with

```
$z = s/x/y/g;           # Should be =~
```

- So let’s forbid it.

```
no underscore;         # Forbids use of $_
                        #
                        # Forbidden
$z = s/x/y/g;         # Forbidden
print HANDLE;         # Forbidden
chop;                 # Forbidden
-x;                   # Forbidden
```

- This was invented by Tom Christiansen



no underscore

```
package underscore;
use Carp;

sub TIESCALAR {
    my $class = shift;
    my $dummy;
    return bless \$dummy => $class;
}

sub FETCH { croak "Read access to \$_ forbidden" }
sub STORE { croak "Write access to \$_ forbidden" }

sub unimport { tie $_ => __PACKAGE__ }
sub import { untie $_ }

1;
```



Import.pm Module

- Idea: Method inheritance via `@ISA` is nice
- Wouldn’t it be nice to inherit regular functions also?
- We will emulate it with `AUTOLOAD`

```
sub AUTOLOAD {
    my $code = get_code($AUTOLOAD);
    goto &$code if $code;
    die "Undefined subroutine $AUTOLOAD called";
}

sub get_code {
    my ($fullname) = @_;
    return \&$fullname if defined &$fullname;
    my($pkg, $sub) = ($fullname =~ /^(.*)::(.*)/);
    for my $parent (@{$pkg . '::ISA'}) {
        my $code = get_code(join '::', $parent, $sub);
        return $code if defined $code;
    }
    return;
}
```

- This was invented by Philip Gwyn



Build Your Own map

- map and grep are great.
- Wouldn't it be nice to make some new, similar operators?
- Example:

```
$n = reduce { $a + $b } 1, 4, 2, 8, 5, 7
```

(Yields the sum, 27)

```
$n = reduce { $a * $b } 1, 4, 2, 8, 5, 7
```

(Yields the product, 2240)

```
$n = reduce { $a > $b ? $a : $b } 1, 4, 2, 8, 5, 7
```

(Yields the max, 8)

```
$n = reduce { [@$a, $b] } [], (1, 4, 2, 8, 5, 7)
```

(Yields a list, [1,4,2,8,5,7])



reduce

```
sub reduce (&$@) {
    my $code = shift;
    local $a = shift;

    for (@_) {
        local $b = $_;
        $a = &$code;
    }

    $a;
}
```

- (&\$@)?!
- local?!
- Why \$a and \$b?



reduce

- Here's a fine, fine trick.
- Let's write a reduce call to ask if a list contains all positive numbers.
- If you apply this to the list (0 .. 1000000), it goes all way to the end
- Solution:

```
reduce { $a && $b > 0 } "yes", @list;
```

```
reduce { $a && $b > 0 || ($a=undef, last) } "yes", @list;
```

- last?!
- Yes! last is dynamically scoped!



combine

```
@list1 = (1,2,3,4,5);
@list2 = (2,3,5,7,11);
@result = combine { $a + $b } @list1, @list2;
```

@result is (3,5,8,11,16)

```
sub combine (&\@\@) {
    my ($code, $ar1, $ar2) = @_;
    my @result;

    while (@$ar1 && @$ar2) {
        local $a = shift @$ar1;
        local $b = shift @$ar2;
        push @result, &$code;
    }

    @result;
}
```



Matching Many Patterns at Once

```
@state_abbr = qw(AK AL AR AZ CA ... WV WY);

@state_pat = (
  'Alaska',
  'Alabama',
  'Arkansas',
  'Ariz(?:\.|ona)?',
  'Cal(?:\.|if(?:\.|ornia)?)',
  ...
  'W(?:est|\.)?s*V(?:irginia|\.)?',
  'Wyo(?:\.|ming)?',
);
```

- Given \$input, does it match a state? Which one?

Matching Many Patterns at Once

- The wrong way:

```
for ($i=0; $i < @state_pat; $i++) {
  return $state_abbr[$i]
  if $input =~ /$state_pat[$i]/;
}
return;
```

Matching Many Patterns at Once

- A better way:

```
$pat = join '|', map "($_)", @state_pat;
```

- \$pat now looks like:

```
(Alaska)|(Alabama)|...|(Wyo(?:\.|ming)?)
```

- Now use:

```
my @matchlist;
if (@matchlist = ($input =~ /$pat/o))
  my $i = 0;
  ++$i until defined $matchlist[$i];
  return $state_abbr[$i];
} else {
  return;
}
```

OFFICIAL
DISCORDIAN SOCIETY
HAIL ERIS

- Caution: Important to use (?:...) instead of (...) in subpatterns.

Quick Return with Warning

```
unless (open LOG, ">> $LOGFILE") {
  warn "Couldn't append to $LOGFILE: $!";
  return;
}
```

- This is a very common locution.

- Perhaps you might prefer this:

```
return warn "Couldn't append to $LOGFILE: $!"
unless open LOG, ">> $LOGFILE";
```

- That returns 1 on an open failure---perhaps not what you want.

```
return !warn "Couldn't append to $LOGFILE: $!"
unless open LOG, ">> $LOGFILE";
```

