

# *The Perl Journal*

## **Building Collaborative Web Applications with *CGI::Wiki***

Kate L. Pugh • 3

## **Managing Your MP3 Library in Perl**

Luis E. Muñoz • 7

## **Enhancing Terminal Output in Perl**

Shay Harding • 11

## **Destructors and Weak References**

Randal Schwartz • 14

## **Five Ways to Find Files**

Andy Lester • 17

## **PLUS**

**Letter from the Editor • 1**

**Perl News Special Report: Perl Whirl '03 • 2**

**Book Review by Russell J.T. Dyer:**

***Perl 6 Essentials* • 20**

**Source Code Appendix • 22**

## LETTER FROM THE EDITOR

### Getting Together

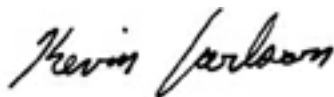
If there's one thing that sets Perl programmers apart, it's a sense of community. Whatever ups and downs the language has experienced, the single constant to date in the history of Perl has been the willingness of folks like you to be part of a team—to collaborate on projects, and just generally help each other out. For the most part, credit for this goes to Larry Wall. He set the tone of openness and inclusion that has pushed Perl in the right direction. But it's more than that.

It's also partly that Perl is not just a working language, but a playing language, too. People use this language for fun, not just for getting the job done. (Of course, if we can have fun getting the job done...) This brings Perl people together in a way that mere necessity doesn't. It's the sort of sentiment that leads to gatherings like the *Perl Whirl* (see our special report on page 2), which embodies this Perl-ish mixing of business and pleasure (not to mention pushing the bounds of avant-garde fashion). And then there are the Perl user groups scattered all over the world, made up of people just as likely to come together for pizza and beer as for solving programming problems.

Interestingly, Perl can be not just a reason for coming together as a community, but can specifically enable it. Take, for instance, the *CGI::Wiki* module that Kate Pugh exposes for you this month; see her article on page 3. It represents a tremendously flexible way to build Wikis and other virtually self-maintaining collaborative communities on the Web. For a somewhat more lightweight way to build Wikis, you should also check out Brian Ingerson's *CGI::Kwiki* module. These modules are two excellent approaches to the same problem. And, true to Perl form, not only do these modules allow a convenient and efficient way to encourage collaboration, but building sites with them is a great deal of fun.

Of course, all collaboration brings conflict. No one is claiming that peace and harmony reign in the kingdom of Perl. The differences of opinion over the directions that Perl 6 should take are evidence enough that we don't all think with one mind. And we definitely all have opinions about each others' code, and some of us aren't shy about expressing them. But it's probably harder in Perl than in any other language to nail down what's "right" and what's "wrong." In Perl, there's only "works" and "doesn't work." And that's by design.

The Perl mantra "There's More Than One Way To Do It" defines more than just a language principle. It says something about coexistence, about including a plurality of ideas in the machinery of a single technology. Our minds don't all work the same way—why should our programming language make us all solve problems the same way? Such inclusiveness is a habit in Perl circles, and it leads to the kind of community that we've all come to depend on. In such a polarized world, it's nice to know we can all be part of this little tribe.



Kevin Carlson  
Executive Editor  
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to [editors@tpj.com](mailto:editors@tpj.com), faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$12.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2003 by CMP Media LLC, unless otherwise noted. All rights reserved.



### The Perl Journal

#### EXECUTIVE EDITOR

Kevin Carlson

#### MANAGING EDITOR

Della Song

#### ART DIRECTOR

Margaret A. Anderson

#### NEWS EDITOR

Shannon Cochran

#### EDITORIAL DIRECTOR

Jonathan Erickson

#### COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, Randal Schwartz

#### CONTRIBUTING EDITORS

Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

#### INTERNET OPERATIONS

##### DIRECTOR

Michael Calderon

##### SENIOR WEB DEVELOPER

Steve Goyette

##### WEB DEVELOPER

Bryan McCormick

##### WEBMASTERS

Sean Coady, Joe Lucca, Rusa Vuong

#### MARKETING / ADVERTISING

##### PUBLISHER

Timothy Trickett

##### MARKETING DIRECTOR

Jessica Hamilton

##### GRAPHIC DESIGNER

Carey Perez

#### THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403

650-513-4300, <http://www.tpj.com/>

#### CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND

MARKETING Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS William Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

PRESIDENT, GROUP PUBLISHER TECHNOLOGY

SOLUTIONS Robert Faletta

PRESIDENT, GROUP PUBLISHER HEALTHCARE MEDIA

Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED

TECHNOLOGIES Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATION

TECHNOLOGY Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK

TECHNOLOGY Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE

DEVELOPMENT MEDIA GROUP Peter Westernman

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

# Perl News



Perl Whirl-ers at the teppan-yaki grill. From left to right: Mark Bilodeau, Larry Fine, Jeff “japhy” Pinyan, Larry Wall, Gloria Wall, Joshua Hoblitt, and Elyse Grasso.

## Geeks at Sea

During Perl’s long and storied history, coders from around the world have attended conferences and seminars, getting to know the luminaries of the language. Speakers and contributors have earned reputations and fan followings. In anticipation of this year’s Perl Whirl, one question dominated the hearts and minds of coders, admins, and fans: Will Larry Wall’s choice of tuxedo stand up to an environment in which people routinely wear “Aloha” shirts? Happily, and to the relief of the attendees, the answer was a resounding “yes.” The Great One graced one evening’s dinner with a canary yellow tuxedo with matching ruffled shirt, and also appeared in a double-breasted burgundy dinner jacket and black bow tie. (This concludes the Perl Whirl Fashion Report.)

Though the conference was scheduled concurrently with “Mac Mania II,” a separate, larger Geek Cruise that took place only a few weeks before YAPC 2003, Perl Whirl ’03 never faded into the shadows. A highly motivated group of about 35 conference attendees enjoyed about three days’ worth of presentations and seminars over the course of the week aboard ship. Conference organizer “Captain” Neil Bauman carefully scheduled the events so as not to conflict with any of the ports of call.

The first portion of the conference presented attendees with a choice of two tracks: Randal Schwartz taught a three-part “Packages, References, Objects, and Modules” course; while Allison

Randal and Damian Conway presented a preview of Perl 6. The Good Doctor (Conway) also delivered a keynote address. The talks later in the week focused more on tips and tricks, culminating in Mark-Jason Dominus’s wildly popular “Red Flags” presentation.

As is often the case, the hallway track proved to be among the most popular, useful, and enjoyable. Every conference attendee had the opportunity to eat dinner with the Walls, and Gloria Wall also gave lessons in shuttle tatting, and even provided the supplies. Two “geeks only” shore excursions and several organized dinners allowed speakers and attendees to get to know each other as well. The manageable size of the conference encouraged this sort of interaction.

Perl Whirl was a great opportunity to learn. When you’ve karaoke’d with Randal Schwartz (a touching rendition of Garth Brooks’s “The Dance”), snorked down shrimp cocktails with Damian Conway, or watched the maitre d’ tell Allison Randal to go put some pants on, you learn that they’re just regular folks. Folks like yourself, only smarter. That, and *foreach* will be eliminated in Perl 6.

—Lawrence D.P. Miller

(We want your news! Send tips to [editors@tpj.com](mailto:editors@tpj.com).)



# Building Collaborative Web Applications with *CGI::Wiki*

Most people now know what a Wiki is—in essence, it is a website that can be edited by anyone who comes across it, using a simple form in a web browser. The original Wiki concept was the work of Ward Cunningham nearly a decade ago. Ward’s original design principles for Wiki can be seen at <http://c2.com/cgi/wiki?WikiDesignPrinciples>.

After having worked on grubstreet, the open community guide to London, for several months, I started to feel that some of these design principles were more important to me than others. Concentrating on a few key principles allows “slippage” of the other principles, and you might end up with something quite different from what you started with. *CGI::Wiki* is an example of this. It was originally planned as a rewrite of *UseModWiki* in more modern Perl, but along the way, it acquired a lot more power and flexibility.

## The Backend

*CGI::Wiki* is designed to be flexible about storage and indexing of its data. The main constructor takes between one and three arguments. The only mandatory argument is a *datastore* object. An *indexer* object can also be supplied, to aid searching, and a *formatter* object can be supplied if you wish to use the Wiki syntax other than the default.

```
my $datastore = CGI::Wiki::Store::SQLite->new(dbname => "/home/wiki/store.db");

my $indexdb = Search::InvertedIndex::DB::DB_File_SplitHash->new(
    -map_name => "/home/wiki/indexes.db",
    -lock_mode => "EX" );

my $indexer = CGI::Wiki::Search::SII->new( indexdb => $indexdb );
my $formatter = CGI::Wiki::Formatter::UseMod->new();

my $wiki = CGI::Wiki->new( store => $datastore,
                          search => $indexer,
                          formatter => $formatter );
```

The datastore types currently available are all stored in a database—your choice of Postgres, MySQL, or SQLite. You

---

*Kate (aka “Kake”) is a Perl programmer living in London. Her current main obsession is writing software that lets people collaborate with each other to write about, map, and annotate their town or city, and then plan pub crawls around it. She can be reached at [kake@earth.li](mailto:kake@earth.li).*

could write a flat-file backend if you liked; nobody has wanted one enough yet, though. The SQLite backend (see *DBD::SQLite* or <http://www.hwaci.com/sw/sqlite/> for details) allows you to store a full RDBMS in a single file, so it will suit most situations where running an actual database server would be inconvenient.

The recommended search indexer is *Search::InvertedIndex*, though support for *DBIx::FullTextSearch* (which has phrase searching built in, but only works with MySQL) is also provided.

The formatters are perhaps the most fun bit. The default formatter uses the *Text::WikiFormat* formatting conventions, but a custom formatter is very easy to write. *CGI::Wiki::Formatter::UseMod*, which provides *usemod*-style syntax, and *CGI::Wiki::Formatter::Pod*, which allows you to write your Wiki entirely in POD, can both be found on CPAN, and should serve as examples if you want to write your own.

## A Toolkit, Not a Wiki

So what are the important differences between *CGI::Wiki* and most other Wiki implementations? For one thing, it’s not actually a Wiki; it’s a toolkit for building Wikis—and things that resemble Wikis. You can’t just install it and have an instant Wiki, though it hardly takes any time at all to create a simple one.

Another major difference is its support for metadata. Most Wiki implementations treat their content as an undifferentiated block of text; indexing and categorization are done by hand, and searching rarely gets any more sophisticated than phrase matching.

Wiki users have developed various conventions to get around these limitations. The most widely used one seems to be the convention of adding WikiWords such as adding “CategoryPattern” and “CategoryPerl” to the bottom of a page about a programming pattern implemented in Perl. The concept of WikiWords is much wider than the concept of faking up categories like this. All pages about Perl can then be found by searching the Wiki for pages that contain the word “CategoryPerl.” A hierarchy of categories can be created by creating a page for CategoryPerl and putting “CategoryProgrammingLanguages” at the bottom; and so on.

*CGI::Wiki* allows you, as the writer of Wiki software, to attach any kind of metadata you like to a given node. A list of categories is an obvious choice; it’s easy then to create a plug-in to look for everything in *Category Foo* or one of its subcategories. It’s also very easy to provide your users with a macro to inline a category index into any given page. The index can even be a collapsible

hierarchical listing, if you like. Other kinds of metadata that have proven very useful include location data—latitude and longitude. I'll discuss some other useful metadata below.

## OpenGuides

At the moment, the most fully developed *CGI::Wiki* application available is OpenGuides, a complete system for managing a collaboratively written guide to a city or town.

---

*CGI::Wiki was originally planned  
as a rewrite of UseModWiki in  
more modern Perl, but along the  
way it acquired a lot more power  
and flexibility*

---

OpenGuides makes heavy use of *CGI::Wiki*'s metadata support. It uses the ability to store and retrieve location data, so it can easily handle queries such as “find me all pubs within 300m of Holborn Station,” or “find me all Chinese restaurants within 500m of Trafalgar Square.” It also uses a locale field (more on this later) for grouping places into neighborhoods within a city.

*CGI::Wiki* provides simple metadata access directly; for example to find everything in Holborn we use the method:

```
my @nodes = $wiki->list_nodes_by_metadata( metadata_type => "locale",
                                          metadata_value => "Holborn" );
```

More complicated queries are done via plug-ins. Queries such as “pubs within 300m of Holborn Station” are handled by *CGI::Wiki::Plugin::Locator::UK*, for example. Anyone can write a *CGI::Wiki* plugin; more detailed guidelines are available in *perldoc CGI::Wiki::Extending*, but in essence your plug-in can choose between building on the simple access methods provided by *CGI::Wiki*, and accessing the database backend directly with SQL. The UK locator plug-in uses the latter technique for speed.

Many of OpenGuides' metadata fields are motivated by the desire to output RDF—machine-readable versions of the nodes—so that different OpenGuides installs can communicate with each other, and so that other Internet applications can make use of OpenGuides data. For example, an IRC bot interface to the London OpenGuides install might use *LWP::Simple* to request a URI such as:

```
http://un.earth.li/~kake/cgi-bin/wiki.cgi?action=index;
index_type=locale;index_value=Chinatown;format=rdf
```

and receive a response like

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:dc="http://purl.org/dc/1.0/"
  xmlns:gs="http://the.earth.li/~kake/xmlns/gs/0.1/"
>
```

```
<gs:locale rdf:about="http://un.earth.li/~kake/cgi-bin/wiki.cgi?action=index;
  index_type=locale;index_value=Chinatown;format=rdf">
  <gs:name>Locale Chinatown</gs:name>
  <gs:object>
    <rdf:Description rdf:about="http://un.earth.li/~kake/cgi-bin
      /wiki.cgi?De_Hems,_W1D_5BW">
      <dc:title>De Hems, W1D 5BW</dc:title>
      <rdfs:seeAlso rdf:resource="http://un.earth.li/~kake/cgi-bin
        /wiki.cgi?id=De_Hems,_W1D_5BW;format=rdf" />
    </rdf:Description>
  </gs:object>
  <gs:object>
    <rdf:Description rdf:about="http://un.earth.li/~kake/cgi-bin
      /wiki.cgi?Golden_Harvest,_WC2H_7BE">
      <dc:title>Golden Harvest, WC2H 7BE</dc:title>
      <rdfs:seeAlso rdf:resource="http://un.earth.li/~kake/cgi-bin
        /wiki.cgi?id=Golden_Harvest,_WC2H_7BE;format=rdf" />
    </rdf:Description>
  </gs:object>
</gs:locale>
</rdf:RDF>
```

Parsing this with *RDF::Core::Parser* is simple:

```
use LWP::Simple;
use RDF::Core::Parser;
use URI::Escape;

my $locale = "Chinatown";

my $content = get("http://un.earth.li/~kake/cgi-bin/wiki.cgi?action=index;
  format=rdf;index_type=locale;index_value=" . uri_escape($locale));
my $parser = RDF::Core::Parser->new( Assert => \&assert, BaseURI => "foo" );
my ( @finds, $name );
$parser->parse( $content );

my $return = "things in $locale: ";
if ( @finds ) {
    $return .= join(" ", map { $name{$_} } @finds ) . "\n";
} else {
    $return .= "none, sorry.\n";
}

print $return;

sub _assert {
    my $triple = @_;
    if ( $triple[predicate_uri] eq
        "http://the.earth.li/~kake/xmlns/gs/0.1/object" ) {
        push @finds, $triple[object_uri];
    }
    if ( $triple[predicate_uri] eq "http://purl.org/dc/1.0/title" ) {
        $name{$triple[subject_uri]} = $triple[object_literal];
    }
}
```

And now we can talk to our OpenGuides install over IRC:

```
15:12 <Kake> grothbot: things in Chinatown
15:12 <grothbot> OK, working on it
<grothbot> Kake: things in Chinatown: Crispy Duck, W1D 6PR; De Hems, W1D 5BW;
Golden Harvest, WC2H 7BE; HK Diner; Hung's, W1D 6PR; Misato, W1D 6PG; Tai, W1D 4DH;
Tokyo Diner; Zipangu, WC2H 7JJ
```

So we have three ways to look at the same data—as an HTML page in a web browser, as RDF, or via an IRC bot. How about a fourth—a Scalable Vector Graphics plot?

```

use strict;

use SVG::Plot;
use URI::Escape;
use CGI::Wiki;
use CGI::Wiki::Plugin::Locator::UK;

my $wiki = CGI::Wiki->new( ... );
my $locator = CGI::Wiki::Plugin::Locator::UK->new;
$wiki->register_plugin( plugin => $locator );

my @nodes = $wiki->list_nodes_by_metadata(
    metadata_type => "locale",
    metadata_value => "Chinatown" );

my @points;
foreach my $node (@nodes) {
    my ($x, $y) = $locator->coordinates( node => $node );
    if ($x and $y) {
        my $uri = "http://un.earth.li/~kake/cgi-bin/wiki.cgi?" . uri_escape($node);
        push @points, [ $x, $y, $uri ];
    }
}

my $plot = SVG::Plot->new( points => \@points,
    max_width => 800,
    max_height => 600,
    point_size => 3 );

my $svg = $plot->plot;

print "Content-Type: image/svg+xml\n\n";
print $svg;

```

## Collaborative Mapping

Unlike a traditional Wiki, a metadata-enabled Wiki allows you to work with aggregated data. One excellent application of this is collaborative mapping of physical spaces—not just the objective statistics such as latitude and longitude, but subjective measures like psychological distance between two places, or membership in a neighborhood.

Consider neighborhoods in London; places like Holborn, Bloomsbury, Chelsea, Fulham, Islington. Unlike boroughs, which have clear boundaries defined by the extent of local council responsibility, neighborhoods are vague and fuzzy. They overlap. Is my office in Holborn or is it in Bloomsbury? Is it in both? Do I live in Hammersmith or Fulham? Or both? Or neither?

The OpenGuides node edit form includes a field where one or more “locales” can be entered for a page. If a later editor thinks you’ve got the locales wrong, they can add or delete other locales. So each time a given locale is left alone during an edit could be considered to be some kind of “vote” for that place being in that locale. Here’s how you’d write a plug-in to track these votes.

```

package OpenGuides::LocaleVote;
use strict;

use vars qw( $VERSION @ISA $plugin_key );
$VERSION = '0.01';
$plugin_key = "og_localevote";

use Carp "croak";
use CGI::Wiki::Plugin;
@ISA = qw( CGI::Wiki::Plugin );

```

We inherit from *CGI::Wiki::Plugin* for easy access to the Wiki’s backend datastore. We define a *\$plugin\_key* to identify the namespace for the tables we’ll be writing to.

## Synopsis

```

use CGI::Wiki;
use OpenGuides::LocaleVote;

my $wiki = CGI::Wiki->new( ... );
my $ballot = OpenGuides::LocaleVote->new;
$wiki->register_plugin( plugin => $ballot );

$wiki->write_node( "Calthorpe Arms", "nice pub", undef, { locale => [ "Holborn",
    "Bloomsbury" ] } );

my $locales = $ballot->get_locales( node => "Calthorpe Arms" );
print "Votes for Holborn: $locales{Holborn}";

```

The API is nice and simple—create a Wiki object, create a plugin object, register the plugin with the Wiki, write some data, and pull out the votes for a given node.

```

sub new {
    my $class = shift;
    my $self = { table => "p_" . $plugin_key . "_votes" };
    bless $self, $class;
    return $self;
}

```

We follow the convention described in *CGI::Wiki::Extending* for our table name, and store it on instantiation, for convenience.

```

sub on_register {
    my $self = shift;
    my $table = $self->{table};
    my $datastore = $self->datastore;
    my $dbh = $self->datastore->dbh
        or croak "Not implemented for non-database datastores";
    my $store_class = ref $datastore;
    $store_class =~ s/CGI::Wiki::Store:://;

    # Check table is set up.
    if ( $store_class eq "Pg" ) {
        my $sth = $dbh->prepare
            "SELECT count(*) FROM pg_tables WHERE tablename=?";
        $sth->execute($table);
        my ($table_ok) = $sth->fetchrow_array;
        $sth->finish;
        unless ($table_ok) {
            $dbh->do( "CREATE TABLE $table (node varchar(200), locale text,
                votes integer )" );
        }
    } elsif ( $store_class eq "MySQL" ) {
        ...
    } elsif ( $store_class eq "SQLite" ) {
        ...
    } else {
        croak "Store class $store_class unknown";
    }
}

```

The *on\_register* method will be called when the plug-in is registered by calling *register\_plugin* on the Wiki object. *on\_register* is our chance to check that the table we need to use has been set up in the backend database. We are careful to allow for the possibility that our plug-in may be used by Wikis with nondatabase datastores, if anyone ever writes one.

Since we are inheriting from *CGI::Wiki::Plugin*, before *on\_register* is called, *CGI::Wiki* will store its *datastore*, *indexer* and *formatter* in our object, so we can get hold of the

database handle by calling `$self->datastore->dbh`, and then we have full access to the database. Following the conventions in *CGI::Wiki::Extending*, we only write to tables labeled with our *\$plugin\_key*, though we can read from any table.

```
sub post_write {
    my ($self, %args) = @_;
    my $table = $self->{table};
    my $dbh = $self->datastore->dbh
        or croak "Not implemented for non-database datastores";

    my $node = $args{node};
    my $metadata = $args{metadata};
    my $locs = $metadata->{locale} or return 1;
    my @locales = ref $locs ? @$locs : ( $locs );

    foreach my $locale (@locales) {
        my $sth = $dbh->prepare(
            "SELECT votes FROM $table WHERE node=? AND locale=?" );

        $sth->execute( $node, $locale );
        my ($votes) = $sth->fetchrow_array;
        $sth->finish;
        if ($votes) {
            $votes++;
            my $sth = $dbh->prepare(
                "UPDATE $table SET votes=? WHERE node=? AND locale=?" );
            $sth->execute( $votes, $node, $locale );
        } else {
            my $sth = $dbh->prepare(
                "INSERT INTO $table (node, locale, votes) VALUES(?,?,1)" );
            $sth->execute( $node, $locale );
        }
    }
    return 1;
}
```

`post_write` is called after each node is written. This is where we do the actual storing of the votes.

```
sub get_locales {
    my ($self, %args) = @_;
    my $table = $self->{table};
    my $node = $args{node};
    my $dbh = $self->datastore->dbh
        or croak "Not implemented for non-database datastores";
    my $sth = $dbh->prepare( "SELECT locale, votes FROM $table WHERE node=?" );
    $sth->execute($node);
    my @locales;
    while ( my ($locale, $vote) = $sth->fetchrow_array ) {
        $locales{$locale} = $vote;
    }
    return @locales;
}
```

Finally, `get_locales` just does a straight SQL query to find the locales a given node has been placed in, and the number of votes for each locale.

## Where to Go From Here

What else can we do with this? I really don't know the extent of it. It seems that every time I talk with a group of programmers I come up with a new thing I can use *CGI::Wiki* for. As mentioned at the start of this article, I feel some of the original Wiki design principles are more immutable than others. In particular, I feel that almost anything calling itself a Wiki is greatly enhanced by being two things:

- Open. Anyone can edit any page. If something is wrong or just plain confusing, you can change it. *CGI::Wiki*'s plug-in system lets Wiki authors extend the collaborative aspect of openness to valuing the aggregate of opinions as visibly as the opinion of the last person to edit a page.
- Organic. I've watched grubstreet grow from a couple of very diverse pages written by two or three people, to a deeply organized structure. Some aspects of the structure came from pre-existing Wiki conventions such as categories; others, such as the ideas of places being within locales and locales neighboring each other, originated entirely in the nature of grubstreet itself. The metadata support in *CGI::Wiki* is naive yet flexible, precisely because I don't believe I can foresee how people will want to structure the Wikis they use. Simply by creating a couple of specialist nodes for collaborative administration of your Wiki, you can let people add and organize categories, create and edit templates for node display and editing, and any other global aspect that you can think of.

Everything else is just there waiting for you to stretch it in interesting ways.

TPJ

## 101 Perl Articles!

From the pages of *The Perl Journal*, *Dr. Dobbs's Journal*, *Web Techniques*, *Webreview.com*, and *Byte.com*, we've brought together 101 articles written by the world's leading experts on Perl programming. Including everything from programming tricks and techniques, to utilities ranging from web site searching and embedding dynamic images, this unique collection of *101 Perl Articles* has something for every Perl programmer.



Plus, this collection of articles is fully searchable, and includes a cross-platform search engine so you can immediately find answers you're looking for. Delivered as HTML files in a ZIP archive or CD-ROM image, download *101 Perl Articles* and burn your own CD-ROM or store it on hard disk.

For subscribers to  
*The Perl Journal*

**\$9.95**

For nonsubscribers to  
*The Perl Journal*

**\$12.95**

To subscribe to *The Perl Journal*  
and receive *101 Perl Articles*

**\$22.90**

Go to  
**<http://www.tpj.com/>**  
now!



# Managing Your MP3 Library in Perl

These days, the practice of backing up one's music CDs as MP3 (or OGG files) has become widespread. I own a few hundred CDs that I've purchased over the years, and a few of them have become coasters through a complex process that involves the passage of time, rough surfaces, and dirt. However, keeping a few gigabytes worth of MP3 files in my laptop's hard drive is not the best way to back up this material. It is also not cheap. When I decided to write the tool I shall describe in this article, I had about 6 GB of MP3 files. I know people who have much more than that, though.

The solution, of course, was to burn a few CD-ROMs with the MP3 files to make room on my laptop's drive. After all, there's no point in carrying eight days of nonstop music with you unless you're a DJ.

## Overview

I wanted a database to hold information about the ID3 tag of each song as well as its location in my backup library. This would allow me to quickly locate, say, all the songs from a given artist. Sometimes I find errors in the tags, so I would like my fixes to be incorporated in the library automatically.

I also wanted the process to be simple and ideally, integrated with the Mac OS X environment on my PowerBook, which is where I most wanted to use this tool. I wanted to simply insert a blank CD-ROM in my burner, fire up a command, and wait for the CD-ROM with my MP3s to be ejected. As you'll see later, I came really close.

My solution is based on the excellent *File::Find* module, which simplifies the task of traversing a file tree such as the one typically associated with an MP3 library. For managing operations with file names, paths, and the like, I used *File::Spec* and *File::Path*, respectively. This helps me ensure portability for my new tools. I also used *MP3::Tag* for reading the ID3 tags in the MP3 files. I handle the detection of changes in already archived files with *Digest::MD5*. The database is maintained with *DB\_File*, *Storable*, and *MLDBM*, which allow me to conveniently store a Perl data structure in a file that can be accessed very quickly. Thanks to all these wonderful and free modules, easily found through a search in CPAN, my task became much simpler.

I decided to make multiple command-line tools for more or less specific tasks. This helped me to keep the interface simple enough to be easy to remember. At the time of this writing, there are two tools: *mp3cat* for copying files, adding them to the database or

finding out which files are or are not archived; and *mp3dump*, a database reporting and backup utility that I hope will make it easier to find a particular song in the archive. I will visit each of those programs in turn and explain their inner workings. (Both utilities are available in their entirety online at <http://www.tpj.com/source/>.)

Access to the database is encapsulated through the use of a tied hash. Tied hashes provide a very simple model for manipulating data. Normally, all hash operations (especially *keys*, *values*, and *each*) work as expected with a tied hash. *MLDBM* allows for the storage of serialized Perl data structures, which can be easily read back when needed. To do this, *MLDBM* uses the help of a database module such as *DB\_File* and a serialization module such as *Storable*.

## *mp3cat*: Keeping Track of the Database

The first part of the script (Listing 1) handles the specification of the modules to use, loads *warnings* and *strict*, which should always be used as they help trap hard-to-find errors such as misspelled variables. After this comes the declaration of variables, specification of command-line options, some error checking in the command-line options with proper error responses courtesy of *Pod::Usage*, and the specification of the default database name.

Because the database will be accessed through a tied hash, at line 132, I declare the hash and make sure it is empty to begin with. To prevent abrupt interruptions from corrupting the database, lines 134 to 138 show a signal handler that kicks in when the user interrupts the script in the middle of its execution. This handler calls *untie* on the hash that is *tied* to the database at lines 140 and 141, to force it to close gracefully, preventing corruption. To be safe, I assume this might not be enough protection, and I regularly backup the database just in case. In the worst case, rebuilding the database is a matter of reinserting all the library CD-ROMs, but it is so much faster to simply copy a file to another directory.

```
132 my %db = ();
133
134 $SIG{INT} = sub
135 {
136     untie %db;
137     die "User requested interruption\n";
138 };
139
140 tie %db, 'MLDBM', $opt_d, O_CREAT | O_RDWR, 0666
141     or die "Failed to tie database $opt_d: ${!}\n";
```

Next comes the part of the code that traverses the directories specified in the command line, at lines 145 through 156. This code

*Luis is an Open Source and Perl advocate at a nationwide ISP in Venezuela. He can be contacted at [luismunoz@cpan.org](mailto:luismunoz@cpan.org).*



is quite simple, thanks to *File::Find*. Basically, we're requesting a recursive traversal of each path in the command line, stored in *\$dir*. For each filesystem object found, the subroutine *analyze* will be called. Lines 152 and 153 show some customization we've asked for, namely following symbolic links and not changing directories during the directory traversal.

```

145 for my $dir (@ARGV)
146 {
147     find(
148         {
149             wanted => \&analyze,
150             # Follow symlinks and don't chdir() into
151             # each subdir
152             follow => 1,
153             no_chdir => 1,
154         }, $dir
155     );
156 }

```

The subroutine *analyze* (lines 238 to 285), is responsible for extracting the suitable data from each MP3 file. Note how we restrict the file's extension with a simple regexp at line 240, to avoid unnecessary work. However, you could remove this and use this utility to perform incremental backups of your files. The hash reference *\$song* will be used to store all the information we can get from the song we're processing. The first information element we have is its filename, which is passed by *File::Find* via the *\$File::Find::name* scalar. I store this information at line 242.

```

238 sub analyze
239 {
240     return unless $File::Find::name =~ qr/\.(mp3)$/i;
241
242     my $song = { path => $File::Find::name };
243
244     if ($opt_s)
245     {
246         my $mp3 = MP3::Tag->new($File::Find::name);
247
248         ($song->{name},
249          $song->{track},
250          $song->{artist},
251          $song->{album}) = $mp3->autoinfo;
252
253         $mp3 = undef; # Free any resources
254
255         unless ($song->{name} or $song->{artist} or $song->{album})
256         {
257             warn "$File::Find::name contains no understandable tags\n";
258         }
259
260         $song->{_} ||= '?' for qw(name track artist album);
261     }
262 }

```

When the *-s* option is specified, the *if* block on lines 244 through 261 decodes the ID3 tag information that may lie inside the MP3 file. Otherwise, this task is skipped to save time and resources. The information is decoded via a call to the *autoinfo* method that *MP3::Tag* provides, at line 251. This will even try to derive information from the filename if no tags are found. Since no more tag-related operations will be done, I request the destruction of the *MP3::Tag* object at line 253 by assigning *undef* to the object reference. Line 260 stores a placeholder for attributes in case the data is not available.

```

263 $song->{size} = -s $File::Find::name;
264
265 my $fh = new IO::File $File::Find::name, "r";
266
267 unless ($fh)
268 {
269     warn "Failed to open $File::Find::name: $!\n";
270     return;
271 }
272
273 binmode($fh);
274
275 $song->{md5} = Digest::MD5->new->addfile($fh)->hexdigest;
276
277 $fh->close;
278
279 $song->{file}=(File::Spec->splitpath($File::Find::name))[2];

```

In line 263, I store the file length in *\$song->{size}* using the *-s* operator. With the code at lines 265 to 277, I calculate the MD5 signature of the MP3 file. An MD5 signature or "Message Digest" is actually a 128-bit number that is assigned to a sequence of bytes by a series of mathematical operations. I use this to recognize when a file has been changed because of two nice properties of message digests:

- Any change in the file produces a completely different digest.
- Finding two files with the same digest is really difficult.

Note that I read the file in binary mode, as requested at line 273, to prevent differences in the treatment of newlines among different operating systems from causing unnecessary duplication.

Because of the uniqueness of the MD5 signature, this is what we'll use as the key to the database, storing the whole *\$song* hash reference. (If someone ever reports a collision, I promise to expand the key with some other data to avoid it.)

At line 279, I use the services of *File::Spec* to find the filename in the path name that *File::Find* gave us through the *\$File::Find::name* scalar. This helps to ensure the portability of the code to operating systems that use different separators in the path names.

```

284 _perform $song;

194 $song->{vol} = $opt_V ||
195     (File::Spec->splitdir
196      ((File::Spec->splitpath
197       (File::Spec->canonpath($song->{path}))[1]))[1] || '?');

```

Once the data has been collected, a call is made to *\_perform* at line 284 in order to decide what to do with this particular song before going to the next in turn.

The first part of *\_perform*, on lines 194 to 197, attempts to provide a volume name, the name chosen for each CD-ROM in the library, based on the mount point. This code selects the second component of the pathname given as the destination, which tends to work well when mounting out of */Volumes*, the default place where Mac OS X mounts new volumes. Of course, the volume specified through the *-V* command line option takes precedence.

At lines 199 through 231 (Listing 2), an *if...elsif* block is used to perform slightly different actions depending on the command line options that were specified. These actions might be attempting to copy the file to a given destination through a call to *\_copy*, adding or replacing an entry in the database with a statement such as *\$db{\$song->{md5}} = \$song*, verifying if this song is already archived with a statement such as *exists \$db{\$song->{md5}}* or printing all the song data, such as in line 229. It is vital to use the *exists* in the verifications, to prevent the autovivification from adding empty entries to the database.

Whenever a reference is assigned to the tied hash `%db`, MLDBM will use *Storable* as requested at line 102, to serialize the Perl data structure referenced. The concept of serialization is very important, because it allows for data structures to be stored for later use. This is sometimes referred to as “persistence.” In essence, serializing a data structure means to translate it to a representation that lacks things like pointers and references to a process’s data. This is often called a flat or serial representation, thus the name. By storing the reference to a hash with all the song data, we can later access this information for other purposes.

Our serialized data is then stored in the database by the *DB\_File* module. This process occurs in the opposite order whenever data is read from the tied hash. It is very important to keep in mind that there is no way to track accesses to the nested structures that might live within the reference. For instance, the following code won’t usually work as expected:

```
# wrong
$db{$my_md5}->{title} = "The lost song";
```

It won’t work because the *tie* interface will fetch the entry corresponding to `$my_md5` from the underlying *DB\_File* database and *Storable* will make sure that a hash reference is reconstructed from the stored data. However, that referenced data, which is being modified in your process, is never being stored back in the database. MLDBM has no way to tell that the referenced data has been altered. A correct alternative is shown below. It works because the reference is fetched from stable storage, modified and then explicitly restored.

```
# longer but correct
my $song = $db{$my_md5};
$song->{title} = 'The lost song';
$db{$my_md5} = $song;
```

Note that in the calls to `_copy`, a *die* follows in case of a false return value, as shown at lines 206, 215, and 223. This is useful to stop the process when a CD-R image is full, which helps make the backup process straightforward.

```
161 sub _copy
162 {
163     my $song = shift;
164     return 1 unless defined $opt_c;
165
166     my $dest = File::Spec->canonpath(File::Spec->catfile($opt_c,
167                                                         $song->{path}));
168     my $dp = (File::Spec->splitpath($dest))[1];
```

I begin `_copy` at line 164 by checking whether a file copy destination was specified in the command line with the `-c` option, returning success otherwise. In lines 166 through 168, I obtain a destination path by combining whatever the user supplied in the command line with the source file path name. This is done with *File::Spec* in order to achieve portability to other operating systems with different file naming conventions.

```
170     mkpath([$dp]);
171
172     unless (copy($song->{path}, $dest))
173     {
174         unlink $dest;
175         warn "Copy error: $!\n";
176         return;
177     }
178
179     warn "$song->{path} transferred\n" if $opt_v;
180     return 1;
181 }
```

Once a destination path has been established, the *mkpath* function from *File::Path* is used on line 170 to ensure the existence of all the required directory components. Next, I invoke *copy* from *File::Copy* at line 172 to attempt the copying of the MP3 file from its original location to the CD image. In case of error, I remove the possibly half-copied file, log the error, and return a false value on lines 174 to 177. If the copy is successful, true is returned.

## mp3dump: Simple Reporting

In order to support a simple mechanism for backup, restore, reporting, and general manipulation of the database, I wrote *mp3dump*. I will omit the explanation of the beginning of the script, as it has a lot in common with *mp3cat*.

```
138 if ($opt_r)
139 {
140     my $csv = new Text::CSV_XS;
141     while (my $line = <>)
142     {
143         last unless $csv->parse($line);
144         my @col = $csv->fields;
145         my $song = {};
146         for my $k (@keys)
147         {
148             $song->{$k} = shift @col;
149         }
150         if (! exists $db{$song->{md5}} or $opt_F)
151         {
152             $db{$song->{md5}} = $song;
153         }
154     }
155 }
```

The first interesting bit of code—the restore operation—appears at lines 138 to 155. Here, I read lines from *STDIN* or files specified in the command line, using the diamond operator at line 141 and then use the *parse* method from *Text::CSV\_XS* at line 143 to obtain the columns from a comma-separated file. A *\$song* hash reference is populated with the columns found at lines 146 to 149, which is added if missing or forced through the `-F` option, at lines 150 to 153.

```
156 else
157 {
158     for my $song (values %db)
159     {
160         if ($opt_c)
161         {
162             no warnings;
163             my $csv = new Text::CSV_XS { always_quote => 1 };
164             $csv->combine(map { $song->{$_} } @keys);
165             print $csv->string, "\n";
166         }
167         elsif ($opt_l)
168         {
169             if (@keys)
170             {
171                 no warnings;
172                 print join(' ', map { "$_=$song->{$_}" } @keys), "\n";
173             }
174             else
175             {
176                 print join(' ', map { "$_=$song->{$_}" } keys %$song), "\n";
177             }
178         }
179     }
180 }
```

All the other command-line options specify a report to be generated from the database, so they appear grouped together in an *else* statement between lines 156 and 180.

When *-c* is specified in the command line, the code on lines 162 to 165 generates a comma-separated report. The *no warnings* at line 162 is necessary to prevent warnings when a given song does not contain a column specified in *@keys*. The *-l* option triggers a simpler report, friendlier for *grep*, which is generated at lines 169 to 177. As you see, managing a database tied to a hash is a trivial task in Perl.

## Using the Tools

As an example, here are the commands I used the last time I created an incremental backup of my CD collection:

```
bash-2.05a$ cp mp3db mp3db.backup
bash-2.05a$ ./mp3cat -c /Volumes/MP3_014 -s -V mp3-
cd-014 ~/Music
```

The *cp* statement makes a simple backup of the database. I always do this before starting, just in case the CD-ROM burning fails or something simply goes wrong. This saves me from having to alter the database in a more complex way.

The invocation of *./mp3cat* requests that songs not in the database be copied (*-c*) to a directory below */Volumes/MP3\_014*, new songs have their information stored in the database (*-s*) and that a volume name of *mp3-cd-014* (*-V*) be used. The songs to be processed are in subdirectories of *~/Music*, which is where my MP3 library resides. A few minutes later, the command terminates and I happily burn my

CD-ROM. If this were to fail, I could simply issue the command:

```
bash-2.05a$ cp mp3db.backup mp3db
```

to restore my old version of the database, and start again.

Let's say that I want to load a copy of my database in a spreadsheet program to do some fancy formatting. I could do so with a command such as this:

```
bash-2.05a$ ./mp3dump -c > my_music.txt
```

I could then simply import the CSV file. This file also happens to be a backup of the database, which could be easily restored by a command such as:

```
bash-2.05a$ ./mp3dump -r -d restored_db my_music.txt
```

Another useful trick is restoring songs from my library. The following command shows how would I go about finding out where certain songs I want are backed up. Note that it would be a very good idea to install *agrep* alongside these tools, for tasks such as this.

```
bash-2.05a$ ./mp3dump -C vol,artist -l | egrep -i 'enya' | cut -f1 -d, ' ' |
                                                    sort | uniq -c
21 vol=mp3-cd-008
2 vol=mp3-cd-012
```

As you can easily see, these tools are very simple yet powerful enough to handle the task. I hope this discussion of these tools gives you a valuable start in using these techniques.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

### Listing 1

```
1  #!/usr/bin/perl
2
3  # This is free software restricted by the same terms as Perl itself.
4  # (c) 2003 Luis E. Muñoz, All rights reserved.
5
6
7
87 use Fcntl;
88 use strict;
89 use warnings;
90 use IO::File;
91 use MP3::Tag;
92 use Storable;
93 use File::Find;
94 use File::Spec;
95 use File::Copy;
96 use File::Path;
97 use Pod::Usage;
98 use DB_File;
99 use Getopt::Std;
100 use Digest::MD5;
101
102 use MLDBM qw(DB_File Storable);
103
104 use vars qw($opt_c $opt_d $opt_F $opt_h $opt_l $opt_n $opt_s $opt_v
105             $opt_V);
106
107 our $VERSION = do { my @r = (q$Revision: 1.6 $ =~ /\d+/g);
108                       sprintf "%d.%03d" x $r, @r };
109
110 getopts('c:d:FhlnsvV:');
111
112 if ($opt_h)
113 {
114     pod2usage
115     {
116         -verbose => 2,
117         -exitval => 255,
118         -message => "\n*** This is $0 version $VERSION ***\n\n",
119     };
120 }
121
122 if (defined($opt_s) + defined($opt_n) + defined($opt_l) > 1)
123 {
124     pod2usage
125     {
126         -verbose => 1,
127         -exitval => 255,
128         -message =>
129             "Only one of -s, -l and -n can be specified at the same time",
130     };
131 }
132
133 $opt_d ||= './mp3db';
```

### Listing 2

```
199 if ($opt_s)
200 {
201     # If the song is not in the DB or the
202     # -F option is given, store it
203
204     if (! exists $db{$song->{md5}} or $opt_F)
205     {
206         _copy($song) || die "Terminating due to copy failure\n";
207         print $song->{path}, " stored\n" if $opt_v;
208         $db{$song->{md5}} = $song;
209     }
210 }
211
212 elsif ($opt_n)
213 {
214     unless (exists $db{$song->{md5}})
215     {
216         _copy($song) || die "Terminating due to copy failure\n";
217         print $song->{path}, "\n";
218     }
219 }
220
221 elsif ($opt_l)
222 {
223     if (exists $db{$song->{md5}})
224     {
225         _copy($song) || die "Terminating due to copy failure\n";
226         print $song->{path}, "\n";
227     }
228 }
229
230 else
231 {
232     print join(' ', map { "$_=$song->{$_}" } keys %$song), "\n";
233 }
```

TPJ

# Enhancing Terminal Output in Perl

This article discusses how to make terminal output easier to read and monitor. If you are like me and spend most of the day at a UNIX command prompt, you'll probably benefit from using *Term::Report* and *Term::StatusBar*, two modules I created for making terminal output easier to track. They can be used separately, but are best if used together. These modules have few dependencies, which are loaded as required, so there is no need to install anything new.

Typically, programs send their output to STDOUT or STDERR. If there are too many lines sent to the terminal, they end up scrolling and may be irretrievable depending on your terminal's buffer size. You can send the output to a file, but then how do you monitor the program to make sure everything is going well? You could *tail -f <file>*, but then you run into the same problem of output scrolling off of the screen.

The most important components of useful terminal output are:

1. The ability to see that the program is processing.
2. The ability to see what the program is doing.
3. The ability to track the program's progress.

With these criteria met, there is no doubt about whether the program has hung or what progress it has made in processing the data.

## Standard Terminal Output

A simple example of terminal output seen in many scripts is shown in Listing 1. This first example will not utilize *Term::Report* or *Term::StatusBar*. This will be modified later in order to show how the *Term* modules work and how they can be useful.

This listing prints the output to the terminal. If there were thousands of items to iterate over, the output would end up scrolling off the screen. Another problem is that there is no way to gauge progress and tell how long it might take to finish. The data sent to the terminal ends up being cluttered and not very useful. In processing larger data sets, it would be difficult to determine the program's output by using this method.

## Cleaning up the Output

The previous output can be organized with the help of *Term::Report*. With minor alterations, the amount of output can be reduced, thus improving readability; see Listing 2.

---

*Shay has worked with transaction processing systems at CCBill LLC, for the last five years and can be contacted at [sharding@ccbill.com](mailto:sharding@ccbill.com).*

The use of *Term::Report* usually doesn't get any more complicated than the aforementioned example. But even this simple implementation of *Term::Report* makes the output more organized and easier to follow. It is readily apparent that the program is working and what data it is processing. Two important criteria for useful terminal output have been met.

If there are thousands of items to iterate over, the constructor can be changed as shown below:

```
my $report = Term::Report->new(startRow => 1, numFormat => 1);
```

This would format numbers using *Number::Format* (i.e., 1000 becomes 1,000). This makes it even easier to read the output quickly.

The last criterion, the ability to track the program's progress, is accomplished by using *Term::StatusBar*. Rather than create a separate object, use *Term::Report*'s ability to wrap the *Term::StatusBar* module. (See Listing 3.)

Notice that *Term::Report* has been used to create a status bar. The status bar needs to know how many items there are to process. Then it's just a matter of calling *StatusBar->update()* with each iteration of data processing.

When updating the inventory in the aforementioned example, the status bar is reset rather than creating a new object. To tell the status bar to empty rather than fill, pass *reverse => 1* to the *reset()* method. This is a recent addition to *Term::StatusBar*. Calling the *printBarReport()* method outputs our statistics summary. This prints a horizontal bar chart based on the final values and scale of the status bar.

With these minor changes, all three criteria for useful terminal output are satisfied. Use the *subText* and *subTextAlign* methods of *Term::StatusBar* to enhance the output further. These place information just under the status bar to show what the program is currently processing.

```
my $report = Term::Report->new(
    startRow => 4,
    numFormat => 1,
    statusBar => [
        label => 'Widget Analysis: ',
        subText => 'Locating widgets',
        subTextAlign => 'center'
    ],
);
...
if (!($_%int((rand(10)+rand(10)+1)))){
```



```

    $report->finePrint('discarded', 0, ++$discard);
    $status->subText("Discarding bad widget");
}
else{
    $status->subText("Locating widgets");
}

```

Another new addition to *Term::StatusBar* is the *showTime* parameter. When turned on, an estimated time to completion is placed at the top of the status bar. Notice in the code below, the value of *startRow* has changed. This is to allow space for the estimated completion time. In a future version, this sort of manual adjustment probably will not be necessary.

```

my $report = Term::Report->new(
    startRow => 5,
    numFormat => 1,
    statusBar => [
        label => 'Widget Analysis: ',
        subText => 'Locating widgets',
        subTextAlign => 'center',
        showTime => 1
    ],);

```

If the module is unable to figure out the estimated time, then “00:00:00” will be displayed. When using the *reverse* method, there is no estimated time tracked. This will be possible in future releases of the module.

## Caveats and Possible Enhancements

*Term::StatusBar* has some limitations. It requires knowing up front how many items there are to process. This is so it can properly set its scale and appropriately update progress. Problems might arise in processing an extremely large file. Many computers do not possess enough memory to load an entire file in order to determine how many lines it contains.

Even if memory is not a factor, it would take a while to read these kinds of files twice: once for *Term::StatusBar* and once to process the data. We can determine a file’s size by using the file test operator *-s*. This is only useful if the program reads a byte at a time. Usually, a file is read line-by-line, and each line may not be the same length in bytes. One strategy would be to sample lines in the file some number of times. The *sysopen*, *sysseek*, and *sys-*

*read* functions can be used to avoid using a lot of memory. However, this would only give a good guess and could be completely wrong depending on the unevenness of the line lengths of the file.

Another, and possibly more accurate way would be to use the file’s size as a basis. Then, when calling *Term::StatusBar->update()*, pass it the length of the line just processed. This would allow accurate tracking of progress as the file was processed. This would also allow the processing of many types of files in the same fashion.

Another possible addition in the near future is a way to “serialize” the output to a file and reinstate it to the terminal at a later point in time. This would allow a process to run in the background and be monitored periodically.

While *Term::Report* and *Term::StatusBar* may not be perfect, they can help improve readability of terminal output. These modules give you the ability to monitor a program’s progress and quickly determine its processing status. They may not work in every situation, but the goal is to make them usable in many situations.

TPJ



(Listings are also available online at <http://www.tpj.com/source/>.)

### Listing 1

```

#!/usr/bin/perl

$|++;
use Time::HiRes qw(usleep);

my ($items, $discard) = (100,0);

## Monitor inventory (L=Locating; D=Discarding)
for (1..$items){
    if (!($_%int((rand(10)+rand(10)+1)))){
        $discard++;
        print "D ";
    }
    else {
        print "L";
    }
}

usleep(50000);
print "\n";

## Update inventory
for (1..($items-$discard)){
    print "U";
}

print "\n\n    Summary for widgets: \n\n".

```

```

"    Total:      $items\n".
"    Good Widgets: ".$items-$discard)."\n".
"    Bad Widgets: $discard\n\n";

```

### Listing 2

```

#!/usr/bin/perl

$|++;
use Time::HiRes qw(usleep);
use Term::Report;

my $report = Term::Report->new(startRow => 1);
my ($items, $discard) = (100,0);

$report->savePoint('total', "Total widgets: ", 1);
$report->savePoint('discarded', "\n Widgets discarded: ", 1);

## Monitor inventory

for (1..$items){
    $report->finePrint('total', 0, $_);

    if (!($_%int((rand(10)+rand(10)+1)))){
        $report->finePrint('discarded', 0, ++$discard);
    }

    usleep(50000);
}

## Update inventory

```

```
$report->savePoint('inventory', "\n\nInventorying widgets... ", 1);

for (1..($items-$discard)){
    $report->finePrint('inventory', 0, $_);
}

$report->printLine("\n\n\n\n    Summary for widgets: \n\n");
$report->printLine("        Total:      $items\n");
$report->printLine("        Good Widgets: ".($items-$discard)." \n");
$report->printLine("        Bad Widgets: $discard\n");
```

### Listing 3

```
#!/usr/bin/perl

$|++;
use Time::HiRes qw(usleep);
use Term::Report;

my $report = Term::Report->new(
    startRow => 4,
    numFormat => 1,
    statusBar => [
        label => 'Widget Analysis: ',
    ],
);
my ($items, $discard) = (100,0);

my $status = $report->(statusBar);
$status->setItems($items);
$status->start;

$report->savePoint('total', "Total widgets: ", 1);
$report->savePoint('discarded', "\n Widgets discarded: ", 1);

## Monitor inventory
for (1..$items){
    $report->finePrint('total', 0, $_);

    if (!($_%int((rand(10)+rand(10)+1)))){
        $report->finePrint('discarded', 0, ++$discard);
    }
}
```

```
usleep(50000);
$status->update;
}

## Update inventory
$status->reset({
    reverse=>1,
    setItems=>($items-$discard),
    start=>1
});

$report->savePoint(
    'inventory',
    "\n\nInventorying widgets... ",
    1
);

for (1..($items-$discard)){
    $report->finePrint('inventory', 0, $_);
    $status->update;
}

$report->printBarReport(
    "\n\n\n\n    Summary for widgets: \n\n",
    {
        "        Total:      " => $items,
        "        Good Widgets: " => $items-$discard,
        "        Bad Widgets: " => $discard,
    }
);
```

TPJ

*Subscribe now to*

# Dr. Dobb's E-mail Newsletters

*They're Free!* <http://www.ddj.com/maillists/>

- ✓ **AI Expert Newsletter.** Edited by Dennis Merritt; the AI Expert Newsletter is all about artificial intelligence in practice.
- ✓ **Dr. Dobb's Linux Digest.** Edited by Steven Gibson, a monthly compendium that highlights the most important Linux newsgroup discussions.
- ✓ **Al Stevens C Programming Newsletter.** There's more than one way to spell "C." Al Stevens keeps you up-to-date on C and all its variants.
- ✓ **Dr. Dobb's Software Tools Newsletter.** Having a hard time keeping up with new developer tools and version updates? If so, Dr. Dobb's Software Tools e-mail newsletter is just the deal for you.
- ✓ **Dr. Dobb's Data Compression Newsletter.** Mark Nelson reports on the most recent compression techniques, algorithms, products, tools, and utilities.
- ✓ **Dr. Dobb's Math Power Newsletter.** Join Homer B. Tilton and expand your base of math knowledge.
- ✓ **Dr. Dobb's Active Scripting Newsletter.** Find out the most clever Active Scripting techniques from Mark Baker.

Sign up now at <http://www.ddj.com/maillists/>



# Destructors and Weak References

*Randal Schwartz*

Objects—they come and go. Most of the time, we’re concerned with how they get started, and what they mean while they’re around.

But occasionally, we need to know when an object goes away. For example, the object might be holding a filehandle that needs to be closed cleanly, or be using a temporary file that needs to be removed. Or maybe the object is a database transaction that needs to be committed or aborted.

Perl knows at all times how many references are being held for an object (including the built-in primitive types). This is necessary because Perl needs to free the associated memory for that object when the object is no longer needed. For a user-defined object, we can also ask Perl to notify us when the object is removed, so we can attach behavior to that step of the program.

Let’s look at an example. Suppose we have objects representing colored boxes. A simple class definition might look like this:

```
BEGIN {
    package Box;
    sub colored {
        my $class = shift;
        bless { Color => shift }, $class;
    }
    sub DESTROY {
        my $dead = shift;
        print "$dead->{Color} has been destroyed\n";
    }
}
```

We’ve wrapped this into a *BEGIN* block to simulate the result of *using* a module containing this code, while giving us the flexibility to simply include the text within our main program. The *colored* method is a classic constructor, creating a *hashref* object with a single key named *Color*. This key’s corresponding value holds the color of the box.

The *DESTROY* method defines a behavior when a box is finally removed. As with all instance methods, the first parameter is the instance object, which is about to be destroyed. In this case, we’re merely documenting the object’s demise.

The destructor can be triggered when the last variable holding

the object is assigned a different value, when that variable goes out of scope, or when the program ends. Here’s a short snippet illustrating all three:

```
my $red = Box->colored("red");
my $green = Box->colored("green");
print "green is being undef'ed\n";
$green = undef;
{
    my $blue = Box->colored("blue");
    print "end of blue block\n";
}
print "end of program\n"
```

The red box is created at the beginning of the program, and persists throughout the program. The green box is held in *\$green*, which is then overwritten with a different value (in this case, *undef*). The blue box is being held in a variable local to a block, and disappears when the block exits. The output from this program looks like:

```
green is being undef'ed
green has been destroyed
end of blue block
blue has been destroyed
end of program
red has been destroyed
```

Note that the red box is destroyed at the very end of the program. All remaining objects are destroyed in an unpredictable order at the end of the program, unless the program is killed by an uncaught signal or a very serious Perl error (called “panic” errors), or uses *exec*.

If the object is held by another object, nested object destruction can occur. Again, the object doesn’t go away until its very last reference is removed, so the order of calling destructors for nested objects can be reasonably managed with a little bit of thought and understanding.

Let’s put the boxes on a shelf, and see what happens when we get rid of the shelf. First, we’ll create a shelf object, including methods to add boxes:

```
BEGIN {
    package Shelf;
    sub new {
```

---

*Randal is a coauthor of Programming Perl, Learning Perl, Learning Perl for Win32 Systems, and Effective Perl Programming, as well as a founding board member of the Perl Mongers (perl.org). Randal can be reached at merlyn@stonehenge.com.*

```

    bless [], shift;
}
sub DESTROY {
    my $dead = shift;
    print "a shelf has been destroyed\n";
}
sub add {
    my $self = shift;
    push @$self, @_;
}
}

```

The object is a simple blessed array reference, rather than a hash reference, created with a simple *new* method. The array is initially empty, reflecting an empty shelf. (I suppose I could have had the constructor also take the remaining arguments as the initial contents, but let's keep things simple.) The destructor method for a shelf is very brief: merely announcing that the shelf is going away. We'll be updating that method shortly. The last method deals with the contents. Adding an object involves pushing it onto the end of the (infinite) shelf.

Let's start by creating a shelf, putting a red and blue box onto it, and then destroying the shelf:

```

my $shelf = Shelf->new;
my $red = Box->colored("red");
my $blue = Box->colored("blue");
$shelf->add($red, $blue);
print "destroying the shelf\n";
$shelf = undef;
print "end of program\n"

```

and this generates the output of:

```

destroying the shelf
a shelf has been destroyed
end of program
blue has been destroyed
red has been destroyed

```

Just before "destroying the shelf," each box has two references: the lexical variables of the program (*\$red* and *\$blue*), and the array contents of the shelf itself. When the last reference to the shelf is destroyed, the destructor is called, and then all of the contents are eliminated. Because the two boxes still have a remaining reference, they are not eliminated just yet.

However, if we remove the scalar variables, we get a different behavior:

```

my $shelf = Shelf->new;
{
    my $red = Box->colored("red");
    my $blue = Box->colored("blue");
    $shelf->add($red, $blue);
    print "exiting the block\n";
}
print "destroying the shelf\n";
$shelf = undef;
print "end of program\n"

```

Here, we're using *\$red* and *\$blue* only within the block, so when the block exits, those references are removed, but the shelf still contains a single reference to the boxes. Thus, destroying the shelf also destroys the boxes, and we get the following output:

```

exiting the block
destroying the shelf
a shelf has been destroyed

```

```

blue has been destroyed
red has been destroyed
end of program

```

Note that the blue and red boxes are destroyed immediately after the shelf goes away. With no remaining references, they must be removed. Also note that the destruction is rather automatic. In the absence of additional code, the container class is destroyed before the things it contains, which makes sense when you think about the reference counting that is happening behind the scenes.

What if we wanted the boxes gone before the shelf goes away? We could smarten up the shelf destruction a bit to push the boxes off the shelf before the shelf is gone:

```

## in package Shelf:
sub DESTROY {
    my $dead = shift;
    while (@$dead) {
        print "shelf has ".$dead." items\n";
        shift @$dead;
    }
    print "a shelf has been destroyed\n";
}

```

Now, with the same code from before, we get:

```

exiting the block
destroying the shelf
shelf has 2 items
red has been destroyed
shelf has 1 items
blue has been destroyed
a shelf has been destroyed
end of program

```

For every box on the shelf, we shift the contents array, causing the box to fall off the shelf. Because these are the last references to the boxes, the box also suffers a fatal blow, and we get a different behavior.

Thus, by default, we get the container destroyed before any contents, but with a simple bit of coding to walk through the contents and remove them explicitly, we can get the contents to be destroyed first.

Destructors can be handy, but remember that they are invoked only when the *last* reference has been eliminated. As an example, let's alter the box class so that it keeps track of every box created so that we can iterate over them, and also returns the same identical box if asked for the same color twice (there can be only one red box, that is). We'll do that with a *registry* of boxes:

```

BEGIN {
    package Box;
    my %REGISTRY;
    sub colored {
        my $class = shift;
        my $color = shift;
        $REGISTRY{$color} ||= bless { Color => $color }, $class;
    }
    sub colors {
        sort keys %REGISTRY;
    }
    sub DESTROY {
        my $dead = shift;
        print "$dead->{Color} has been destroyed\n";
        delete $REGISTRY{$dead->{Color}};
    }
}

```



The `%REGISTRY` hash is keyed by the color name, with the value being the box object of that color. As each new box is requested, if the box already exists, it is returned. Otherwise, a new box is created and stored into the hash before being returned. The `colors` method lets us get at all the current box colors, and we've added a step to the destructor to ensure that the box is removed from our registry.

At first glance, it looks like we've got decent code. Let's add a few lines to our sample program and run it:

```
my $shelf = Shelf->new;
my $orange = Box->colored("orange");
print "before, we have boxes colored: ", join(" ", Box->colors), "\n";
{
    my $red = Box->colored("red");
    my $blue = Box->colored("blue");
    my $green = Box->colored("green");
    $shelf->add($red, $blue);
    print "inside, we have boxes colored: ", join(" ", Box->colors), "\n";
    print "exiting the block\n";
}
print "outside, we have boxes colored: ", join(" ", Box->colors), "\n";
print "destroying the shelf\n";
$shelf = undef;
print "finally, we have boxes colored: ", join(" ", Box->colors), "\n";
print "end of program\n";
```

Note that we've got an orange box at the beginning and a green box inside the block, and we're calling the `colors` method from time to time to see what all the boxes are like. But let's study the perhaps unexpected output for a moment:

```
before, we have boxes colored: orange
inside, we have boxes colored: blue, green, orange, red
exiting the block
outside, we have boxes colored: blue, green, orange, red
destroying the shelf
shelf has 2 items
shelf has 1 items
a shelf has been destroyed
finally, we have boxes colored: blue, green, orange, red
end of program
red has been destroyed
blue has been destroyed
green has been destroyed
orange has been destroyed
```

Everything looks fine up to exiting the block. Why didn't the green box go away? And when the red and blue boxes were pushed off the shelf, why didn't they go away? In fact, nothing goes away until the end of the program!

The problem is in the registry. Because the `registry` hash holds a reference to every box created until it is destroyed, there's at least one reference to every box as long as the registry is intact. And even though we tried to eliminate that reference in the destructor, the destructor is never called because there's still that last reference!

The solution is to have some way to tell Perl that the links within the registry really aren't as important as the references throughout the rest of the program, and should not count for reference counting. In Perl terminology, this is called a *weak reference*.

We weaken a reference using `Scalar::Util`'s `weaken` routine, which is in the core in Perl 5.8, but can be installed on Perl 5.6 or later. The argument to `weaken` is marked as insignificant to the reference count. If all nonweak references are deleted, the object destructor is called. The remaining weak references are then set to `undef` automatically.

If we fix the constructor to use weak references, we get:

```
## in package Box:
use Scalar::Util qw(weaken);

sub colored {
    my $class = shift;
    my $color = shift;

    return $REGISTRY{$color} if $REGISTRY{$color};
    my $self = bless { Color => $color }, $class;
    weaken ($REGISTRY{$color} = $self);
    $self;
}
```

Again, if there's already an existing object of the right color, we reuse it. If not, we create a new object, and copy that object into the registry. The copied reference is then weakened, and we return the object back as the final value. Note that a weak reference makes a normal reference when copied: only the copies specifically marked with *weaken* are weak. And when we run this, we get output a little more like we expected:

```
before, we have boxes colored: orange
inside, we have boxes colored: blue, green, orange, red
exiting the block
green has been destroyed
outside, we have boxes colored: blue, orange, red
destroying the shelf
shelf has 2 items
red has been destroyed
shelf has 1 items
blue has been destroyed
a shelf has been destroyed
finally, we have boxes colored: orange
end of program
orange has been destroyed
```

I hope you've enjoyed this little walk through destructors and weak references. For more, see my latest book, *Learning Perl's Objects, References, and Modules*, where I cover this topic in more depth, along with everything else needed to write larger programs. Until next time, enjoy!

TPJ





# Five Ways to Find Files

*Andy Lester*

For most command-line utilities you'll write in Perl, chances are you'll need to operate on multiple files in the filesystem. As you'd expect, there's more than one way to do it, and you should choose your method based on the specifics of the task at hand. Do you want a list of files, or do you want to iterate over them? Do you need to operate on directories as well as files? Do you need to find files in subdirectories, too? In this article, I'll show you five different ways your Perl program can find files.

No matter which method you choose, remember that searches will be relative to your current directory. Make sure that any solution you choose gives you filenames with path information. It doesn't help to get a filename of "foo.pl" if the file is three directory levels down.

## UNIX *find*

UNIX is filled with tools designed to do one thing well, and the *find* utility fits this description. If you're comfortable with *find*'s options, then it may make sense to use *find* from inside your Perl program and parse its output. This approach also has the advantage of making it easy for you to debug the rules you're using in the shell, and then wrap it up in your Perl code.

*find* can be a bit cumbersome to use from the command line, but for simple searches, it does just fine. The following works from the shell for our example:

```
$ find . -name '*.pl' -o -name '*.pm' -o -name '*.t' -print
./lib/WWW/Mechanize.pm
./t/00.load.t
./t/99.pod.t
... etc ...
```

*find* takes a starting directory, and then a series of options that tell which files you're interested in. The *-name* *\*.pl* tells *find* to find files that match the pattern *\*.pl*. The *-o* is the "or" option. Chained together, *find* finds anything that matches *\*.pl*, *\*.pm*, or *\*.t*, and then *-print* tells *find* to print the found file to standard output. If you're using GNU *find*, you can leave off the *-print*, as it's assumed to be the action to take.

Note that you must single quote the pattern, or else the shell will expand the wildcard. For example, if you use:

---

*Andy manages programmers for Follett Library Resources in McHenry, IL. In his spare time, he works on his CPAN modules and does technical writing and editing. He can be contacted at [andy@petdance.com](mailto:andy@petdance.com).*

```
find . -name *.pl -print
```

and there is one file in the current directory, and it is named "foo.pl," the shell will expand into

```
find . -name foo.pl -print
```

which tells *find* to only find files named "foo.pl," which is certainly not what you wanted.

Now that I have a working *find* command, I copy it into my Perl program and surround it with the backtick operators. To get the output from *find* into a list, use the backtick operator and read the lines into an array.

```
my @files = `find -name '*.pl' -o -name '*.pm' -o -name '*.t'`;
```

Each line of the output from the *find* will be put into the *@files* array. They're not ready to be used yet, because each line has a "\n" at the end, so we just have to chomp it off.

```
chomp @files;
```

## *opendir/readdir/closedir*

For a purely Perl solution, you have three options: *readdir*, *glob*, or *modules*. I'll start with *readdir*.

Perl has the concept of *dirhandles* that act like filehandles, but let you iterate through entries in a directory in much the same way as iterating through lines of a text file. *dirhandles* are opened with *opendir*, read from with *readdir*, and closed with *closedir*.

In scalar context, *readdir* returns the next directory entry, or *undef* if there are no more. In list context, it returns all the directory entries. What makes *readdir* a challenge is that "directory entry" could be anything: a plain file, a directory, a symlink, special directories such as ".", and so on.

For single-directory searching, *readdir* is pretty handy. Since it can return a list, it's easy to filter your files through *grep*, as in:

```
opendir( DIR, $dir ) or die "Can't open $dir";
my @files = grep -f "$dir/$_" && /\.(pl|pm|t)$/, readdir DIR;
closedir DIR;
```

Each entry from *readdir* is checked to see if it's a plain file and ends in *.pl*, *.pm* or *.t*. Note that we have to prepend the *\$dir* before we check the *-f* operator. Because *readdir* only returns the basename, not the full path, if *\$dir* is anything other than ".", the test results will be inaccurate. This bit me during testing, so keep it in mind.

This approach with *readdir* works nicely for single directories, but for entire trees you need to create a recursive function.

```
my @files = get_files( "." );
sub get_files {
    my $dir = shift;
    opendir( DIR, $dir ) or die "Can't open $dir";
    my @entries = readdir(DIR);
    closedir DIR;
    my @files;
    for my $entry ( @entries ) {
        # Skip current & parent directories, lest we loop
        next if $entry eq "." || $entry eq "..";
        my $fullpath = "$dir/$entry";
        if ( -f $fullpath ) {

            if ( $entry =~ /\.(\pm|pl|t)$/ ) {
                push( @files, $fullpath );
            }
        } elsif ( -d $fullpath ) {
            push( @files, get_files( $fullpath ) );
        }
    }
    return @files;
}
```

The *get\_files* subroutine is basically the same as grepping through the list of *readdir* entries, but it recursively calls itself to get the contents of subdirectories. The full path name for each file must be built in each call to *get\_files*, since *readdir* only returns filenames, not paths. Also, I need to *readdir* the entire directory at once, or else reading the global *DIR* in the recursed call to *get\_files* will mess up the parent.

## globbing

The *glob* operator `<>` and its spelled-out cousin *glob* can be an improvement over *readdir*, both for brevity and for providing shell filename-matching semantics. For example, *glob* does not return the `"."` and `".."` directory entries, or any other file that starts with `"."`. It understands character classes like *\*p[lm]*, alternation such as *\*{pl,pm,t}*, and will expand the tilde to a home directory.

The first *readdir* example would be written with *glob* as:

```
my @files = grep -f, <*.p[l,pm,t]>;
```

I prefer the *glob* keyword in most cases. It looks less like line noise, and can't get confused with the diamond operator in file-reading context.

Updating the sample *get\_files* from above to use *glob* is left as an exercise to the reader, but only if the reader wants to do more work than necessary. *glob* is meant only for single directories, so to handle directory structures, use *readdir*. Or, you can use our next file-finding method, *File::Find*.

## File::Find

The *File::Find* module takes care of all the mess with directory descending that I've shown you in the previous examples, plus it adds a number of handy features. *File::Find* has been included with Perl since Perl 5.00307, and is available on CPAN if you have an older version.

*File::Find* only has two functions, *find* and *finddepth*. They're identical except for *finddepth* performing a depth-first search. *find* takes a reference to a callback subroutine, often called "*wanted*" and a list of starting directories to start searching through. For each file or directory *find* finds, *wanted* is called, allowing you to perform actions and check information about the file.

When *wanted* is called, the following global variables will be set:

- *\$\_* Set to the current filename in the current directory *\$File::Find::dir*.
- *\$File::Find::dir* Set to the name of the current directory. *File::Find* automatically *chdir*s you to this directory when *wanted* is called.
- *\$File::Find::* Name the full path name, relative to the starting directory, of the current file. This is simply "*\$File::Find::dir/\$\_*", and it is mostly a convenience.

One of the simplest ways to use *File::Find* is to accumulate a list of wanted files, like I showed in previous examples, but without having to deal with the recursion into subdirectories. The following prints out all of the files from the current directory downward:

---

*If you spend a lot of time  
rewriting the same wanted  
functions over and over again,  
you may want to turn to Richard  
Clamp's File::Find::Rule*

---

```
use File::Find;
my @files;
sub wanted {
    print $File::Find::name, "\n" if -f;
}
find( \&wanted, "." );
```

Note that I need to print *\$File::Find::name* and not just *\$\_*, or I would only get the filename without the path.

Since *wanted* can do anything, and you can specify multiple starting directories, you have a lot of flexibility. This program figures out how many bytes are used in the files in three different module distributions.

```
use File::Find;
my @modules = qw(
    www-mechanize
    html-lint
    marc-record
);
my @dirs = map { "/home/andy/$_" } @modules;
my ($size, $n);
find( \&wanted, @dirs );
print "$size bytes in $n files\n";
sub wanted {
    if ( -d ) {
        $File::Find::prune = 1 if $_ eq "CVS";
    } elsif ( -f ) {
        $size += -s;
        ++$n;
    }
}
```

Note the use of *\$File::Find::prune*. Setting it to a true value

tells *find* not to descend into the directory. In this case, we want to omit files in the CVS directory from our totals, since they're not part of the actual distribution.

The subroutine reference need not refer to a named subroutine. An anonymous subroutine will work just as well, and often makes for easier-to-follow logic for simple *wanted* functions. For example, to count the number of files in a given directory:

```
use File::Find;
find( sub { $n++ if -f }, ".");
print "$n files found\n";
```

or even from the command line, as I discussed in the May issue:

```
perl -MFile::Find -le'find( sub { $n++ if -f }, ".");' \
-e'END {print "$n files found"}
```

## File::Find::Rule

If you frequently use *File::Find* to return lists of files, and you spend a lot of time rewriting the same *wanted* functions over and over again, you may want to turn to Richard Clamp's *File::Find::Rule*. Instead of creating *wanted* functions, you'll create sets of rules, not unlike the *find* command, in order to retrieve lists of files.

In true Perl style, *File::Find::Rule* makes it easy to do the common tasks, such as finding all the \*.pl files below the current directory:

```
use File::Find::Rule;
my @files = File::Find::Rule->file->name('*.pl')->in( "." );
```

Each function in the chain (*file* and *name*) acts as both a con-

structor and a link in a method chain, and then the *in* method evaluates all the rules for a given directory. You can also use individual method calls on a *File::Find::Rule* object, like so:

```
my $rule = File::Find::Rule->new;
$rule->file;
$rule->name( '*.pl' );
my @files = $rule->in( "." );
```

The various methods are extremely flexible. For example, to get \*.pl, \*.pm, and \*.t files, call *name* with an anonymous array of specifications:

```
$rule->name( ['*.pl','*.pm','*.t'] );
```

This ability to add rules one at a time makes it easy to build rulesets on the fly. For example, you might have command-line options in your program that tell which files you want, and the “*\_big*” option shows that you only want files larger than 20 MB:

```
$rule->size( '>20M' ) if $opt_big;
```

If the object style of specification doesn't suit you, *File::Find::Rule* also supports a functional interface. To find all the files in my home directory beginning or ending with a tilde, which are good candidates for deletion:

```
my @files = find( name => ['*~','~*'], in => '/home/andy' );
```

The rules you use can be combined using Boolean logic with the *and* and *or* functions. If I wanted to find files beginning or ending with tildes, or files with 0 bytes, I can use:

```
my $rule = File::Find::Rule->new;
$rule->file;
$rule->or(
    find( name => ['*~','~*'] ),
    find( size => 0 ),
);
```

Sometimes, you want to iterate over the list of files, like *File::Find*, rather than gathering a whole list of them. *File::Find::Rule* supports this with the *start* and *match* methods:

```
$rule->start( "/home/andy" );
while ( my $file = $rule->match ) {
    # do something with $file
}
```

For more on *File::Find::Rule*'s options and help on how to create your own extensions, see <http://search.cpan.org/author/RCLAMP/File-Find-Rule/>.

## Conclusion

I've taken you on a tour of five different ways to gather lists of, and operate on, files in the filesystem based on your specific criteria. Which method is best? As with any tool, it depends on your context.

For simple lists of files with rudimentary filename matching, globbing is the way to go. If you need to include directory names in your results as well as files, use *readdir*. As soon as you want to start finding files in subdirectories, one of the two modules will make life easier. Both *File::Find* and *File::Find::Rule* handle directories nicely, although *File::Find::Rule* may be better for complex logic that doesn't need to be wrapped up in a *wanted* function.

# Subscribe Now To The Perl Journal

For just \$1/month, you get the latest in Perl programming techniques in e-zine format from the world's best Perl programmers.

<http://www.tpj.com/>

TPJ





# Perl 6 Essentials

*Russell J.T. Dyer*

O'Reilly & Associates has announced a new book on Perl 6 which should be available by the time you read this. *Perl 6 Essentials* is unlike any of O'Reilly's other Perl books; it's not a normal tutorial, per se. Instead, it's a detailed review of the current state of the Perl 6 project, as well as a fairly thorough introduction to Perl 6 and the Parrot environment. If you've been wanting to know what to expect from Version 6, you won't find a better book on the subject. It's especially useful if you're considering getting involved in the development of Perl 6. Additionally, Python and Ruby programmers may find the chapters on Parrot (the programming-language-neutral interpreter) of interest.

## Style

Not an overly lengthy book (about 200 pages), *Perl 6 Essentials* comprises a mere seven chapters. One could read it in a week or two, but you shouldn't rush through it. There's plenty for the reader to absorb. It's a comfortable read—although three authors worked on the book, there is a unified voice. This is a book to be digested: It's not a practical book with exercises at the end of each chapter requiring you to read it by your computer.

You may not want or need to read all of *Perl 6 Essentials*. It probably contains more material than you want to know, even though it's relatively short for a computer book. Instead, selective reading may be called for here. The book's preface gives some suggestions on which chapters to read for different purposes, for different types of readers or programmers. However, to make your own choices, I'll dedicate the remainder of this article to reviewing the contents of each chapter.

## Content

The first chapter contains an interesting short history of the beginning of Perl 6. It names some of the key people who started the project and those who are now in charge of each major component and stage. It's quite a nonjudgmental overview of the Perl community and the state of Perl 6 affairs.

Chapter Two, as brief as it is (only eight pages), is broken up into two parts: The first part is on how to get involved in Perl 6 language development. This is the essentially linguistic aspect of Perl and the level that most of us think of when we discuss or

---

*Russell is a Perl programmer, a MySQL developer, and a web designer living and working on a consulting basis in New Orleans. He is also an adjunct instructor at a local college where he teaches Linux and other open-source software. He can be reached at [russell@dyerhouse.com](mailto:russell@dyerhouse.com).*

## *Perl 6 Essentials*

*Allison Randal, Dan Sugalski,*

*and Leopold Totsch*

O'Reilly & Associates

200 pp., \$24.95

ISBN: 0-596-00499-0

work with Perl. The second part of Chapter Two details how to get involved in Parrot development. The Parrot project is partially under the Perl 6 umbrella. Parrot encompasses the development of Perl 6 internals. However, Parrot goes further in that its design goal is to accommodate Python, Ruby, and some other similar languages, as well. It's to be a language-neutral run-time environment, allowing for flexibility and compatibility in the general programming community.

A linguistic tutorial on Perl is provided in Chapter Three. It encompasses the theoretical language goals of the project. It's not an overly technical or long-winded chapter. The authors work through a series of linguistic principles with regard to Perl. They provide the reader with simple examples, using English language phrases and sentences as illustrations and close analogies in Perl. This chapter has good instructions for potential and existing contributors to Perl 6. It attempts to help would-be contributors to understand the linguistic policies to which Larry Wall and other Perl language leaders have been adhering. One of the goals is to prevent Perl from becoming rigid and forced like many other programming languages, languages in which there is only one way to do things.

Perl syntax is reviewed in Chapter Four. It's a very brief run-through of variables, arrays, hashes, references, and other components of Perl. One wouldn't use this as a substitute for O'Reilly's *Learning Perl* book. Instead, it's useful in getting the reader in sync for the chapters that follow and for the reader's potential involvement in the project. I found it helped me to remove myself from the details of Perl, and pull back to the conceptual level. Chapter Four is not a lightweight chapter, though. It teaches

# Sign Up For BYTE.com!

**DON'T GET LEFT BEHIND!**  
**Register for BYTE.com today**  
**and have access to...**

- Jerry Pournelle's "Chaos Manor"
- Moshe Bar's "Serving with Linux"
- Martin Heller's "Mr. Computer Language Person"
- David Em's "Media Lab"
- and much more!...

**BYTE.com will keep you up-to-date on emerging trends and technologies with even more rich technical articles and opinions than ever before! Expert opinions, in-depth analysis, trusted information...find all this and more on BYTE.com!**



**As a special thank you for signing up, receive the BYTE CD-ROM and a year of access for the low price of \$26.95!**

**Registering is easy...go to [www.byte.com](http://www.byte.com) and sign up today! Don't delay!**

The logo for BYTE.com. It consists of the word "BYTE" in a bold, white, sans-serif font, set against a red rectangular background.

Perl as one would a spoken, purely human language. It is more detailed than the strictly theoretical approach of Chapter Three, but it doesn't get lost in the minutia, either.

Chapter Five covers Parrot: its essential purpose and the goals of the designers, as well as its architectural structure. Chapter Five describes the function and role of each component of the Parrot architecture: the parser, the compiler, the optimizer, the interpreter, and the bytecode loader. The authors explain the interpreter in great detail, because it's where most of the action resides—actually this part composes 19 of the 25 pages of Chapter Five. They explain how the interpreter handles strings and variables, and how it reacts to and controls processes and threads. Chapter Five also explains how Parrot is intended to deal with objects. (This could be particularly tricky, because the Parrot project has the ambitious goal of being compatible with Perl, Python, and Ruby, and their manners of implementing objects are slightly different.)

The Parrot assembly language (PASM) is covered in Chapter Six. It provides a tutorial on working with PASM, which includes Parrot Magic Cookies (PMC: low-level objects). Apparently there are whole other worlds below the surface of Perl that many of us take for granted. The end of the chapter provides a brief reference manual (24 pages in a format similar to *Perl in a Nutshell*) on PASM commands (or rather, *opcodes*).

The final chapter, Chapter Seven, covers the Intermediate Code Compiler (IMCC). The IMCC is an alternative compiler for Parrot bytecode which can embed the Parrot run-time engine to shorten the compile time. The IMCC uses the language called "Parrot Intermediate Language" (PIR). PIR overlays PASM and as such, it is a step above PASM. Like the previous chapter, this chapter ends with a short reference manual (about eight pages) on PIR directives and instructions.

## Conclusion

Although *Perl 6 Essentials* has a practical purpose in that it calls the Perl community to action and provides them with information to help them choose an area of participation, this is primarily a theoretical work. If you're looking to improve your Perl skills in immediate and practical ways, this may not be the book for you. However, if you're looking to deepen your understanding of Perl (Perl 5 included), then a careful reading of this book will help. It will also help prepare you for Perl 6; it will allow you to write better Perl 5 code now with an eye toward the not-so-distant future. In short, *Perl 6 Essentials* may not be essential for all Perl programmers, but all Perl programmers can probably benefit from reading it.

TPJ



---

# Source Code Appendix

---

Luis E. Muñoz “Managing Your MP3 Library in Perl”

## Listing 1

```
1  #!/usr/bin/perl
2
3  # This is free software restricted by the same terms as Perl itself.
4  # (c) 2003 Luis E. Muñoz, All rights reserved.
5
6
7  use Fcntl;
8  use strict;
9  use warnings;
10 use IO::File;
11 use MP3::Tag;
12 use Storable;
13 use File::Find;
14 use File::Spec;
15 use File::Copy;
16 use File::Path;
17 use Pod::Usage;
18 use DB_File;
19 use Getopt::Std;
20 use Digest::MD5;
21
22 use MLDBM qw(DB_File Storable);
23
24 use vars qw($opt_c $opt_d $opt_F $opt_h $opt_l $opt_n $opt_s $opt_v
25             $opt_V);
26
27 our $VERSION = do { my @r = (q$Revision: 1.6 $ =~ /\d+/g);
28                       sprintf "%d.%03d" x $r, @r };
29
30 getopts('c:d:FhlnsvV:');
31
32 if ($opt_h)
33 {
34     pod2usage
35     {
36         -verbose => 2,
37         -exitval => 255,
38         -message => "\n*** This is $0 version $VERSION ***\n\n",
39     };
40 }
41
42 if (defined($opt_s) + defined($opt_n) + defined($opt_l) > 1)
43 {
44     pod2usage
45     {
46         -verbose => 1,
47         -exitval => 255,
48         -message =>
49             "Only one of -s, -l and -n can be specified at the same time",
50     }
51 }
52
53 $opt_d ||= './mp3db';
```

## Listing 2

```
199 if ($opt_s)
200 {
201     # If the song is not in the DB or the
202     # -F option is given, store it
203
204     if (! exists $db{$song->{md5}} or $opt_F)
205     {
206         _copy($song) || die "Terminating due to copy failure\n";
207         print $song->{path}, " stored\n" if $opt_v;
208         $db{$song->{md5}} = $song;
209     }
210 }
211 elsif ($opt_n)
212 {
213     unless (exists $db{$song->{md5}})
214     {
215         _copy($song) || die "Terminating due to copy failure\n";
216         print $song->{path}, "\n";
217     }
218 }
219 elsif ($opt_l)
220 {
221     if (exists $db{$song->{md5}})
222     {
223         _copy($song) || die "Terminating due to copy failure\n";
```

```

224         print $song->{path}, "\n";
225     }
226 }
227 elsif ($opt_v)
228 {
229     print join(' ', map { "$_=$song->{$_}" } keys %$song), "\n";
230 }
231 }

```

## Shay Harding “Enhancing Terminal Output in Perl”

### Listing 1

```

#!/usr/bin/perl

$|++;
use Time::HiRes qw(usleep);

my ($items, $discard) = (100,0);

## Monitor inventory (L=Locating; D=Discarding)
for (1..$items){
    if (!($_%int((rand(10)+rand(10)+1)))){
        $discard++;
        print " D ";
    }
    else {
        print "L";
    }

    usleep(50000);
}
print "\n";

## Update inventory
for (1..($items-$discard)){
    print "U";
}

print "\n\n\n    Summary for widgets: \n\n".
    "        Total:          $items\n".
    "        Good Widgets: ".($items-$discard)."\n".
    "        Bad Widgets:  $discard\n\n";

```

### Listing 2

```

#!/usr/bin/perl

$|++;
use Time::HiRes qw(usleep);
use Term::Report;

my $report = Term::Report->new(startRow => 1);
my ($items, $discard) = (100,0);

$report->savePoint('total', "Total widgets: ", 1);
$report->savePoint('discarded', "\n Widgets discarded: ", 1);

## Monitor inventory
for (1..$items){
    $report->finePrint('total', 0, $_);

    if (!($_%int((rand(10)+rand(10)+1)))){
        $report->finePrint('discarded', 0, ++$discard);
    }

    usleep(50000);
}

## Update inventory
$report->savePoint('inventory', "\n\nInventorying widgets... ", 1);

for (1..($items-$discard)){
    $report->finePrint('inventory', 0, $_);
}

$report->printLine("\n\n\n\n    Summary for widgets: \n\n");
$report->printLine("        Total:          $items\n");
$report->printLine("        Good Widgets: ".($items-$discard)."\n");
$report->printLine("        Bad Widgets:  $discard\n\n");

```

### Listing 3

```

#!/usr/bin/perl

```



```

$|++;
use Time::HiRes qw(usleep);
use Term::Report;

my $report = Term::Report->new(
    startRow => 4,
    numFormat => 1,
    statusBar => [
        label => 'Widget Analysis: ',
    ],
);
my ($items, $discard) = (100,0);

my $status = $report->{statusBar};
$status->setItems($items);
$status->start;

$report->savePoint('total', "Total widgets: ", 1);
$report->savePoint('discarded', "\n Widgets discarded: ", 1);

## Monitor inventory
for (1..$items){
    $report->finePrint('total', 0, $_);

    if (!($_%int((rand(10)+rand(10)+1)))){
        $report->finePrint('discarded', 0, ++$discard);
    }

    usleep(50000);
    $status->update;
}

## Update inventory
$status->reset({
    reverse=>1,
    setItems=>($items-$discard),
    start=>1
});

$report->savePoint(
    'inventory',
    "\n\nInventorying widgets... ",
    1
);

for (1..($items-$discard)){
    $report->finePrint('inventory', 0, $_);
    $status->update;
}

$report->printBarReport(
    "\n\n\n\n Summary for widgets: \n\n",
    {
        " Total: " => $items,
        " Good Widgets: " => $items-$discard,
        " Bad Widgets: " => $discard,
    }
);

```