

The Perl Journal

Creating Self-Contained Perl Executables, Part I

Julius C. Duque • 3

Monitoring Network Traffic Revisited

Paul Barry • 6

Coding for Readability

Charles K. Clarkson • 9

Tate: An iPhoto Gallery Exporter in Perl

Simon Cozens • 12

Making Web Images

brian d foy • 15

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Source Code Appendix • 17

LETTER FROM THE EDITOR

BotNet Zombie Nation

People have got to stop connecting their computers to cable or DSL modems. According to a recent study by the Honeynet Project (<http://www.honeynet.org/papers/bots/>), 10 million computers have been hijacked for use in a BotNet—those distributed networks of compromised computers that listen in on IRC channels for nefarious commands from a master.

These networks can consist of thousands of PCs, mostly running Windows 2000 or Windows XP SP1, that have been automatically hacked by other machines that have themselves been previously compromised. Taking advantage of the numerous security holes in these unpatched operating systems, these bots replicate with frightening speed. Some of the test computers that were used in the Honeynet Project's study were infected within seconds of being connected to the network.

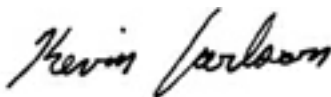
BotNets can be used for all sorts of evil, but the most common uses are for distributed denial-of-service (DDOS) attacks and spamming. But machines compromised in this manner can be involved in much more disturbing illegal activities, including terrorism and child pornography. If mom and pop realized what their computer was doing behind their backs, they might take the whole security issue a bit more seriously. Sadly, the first sign they often get that something is wrong is the sound of the FBI knocking at the door. News of computer security problems rarely reaches the people who most need to hear it. These are folks who don't even look at the technology section of *USA Today*, much less read Slashdot.

But the stakes are rising. As the range and scope of crimes that can be committed on the Internet grows, so does the effort that hackers expend to get control of unsecured machines. Infected computers are valuable commodities—already, hackers in control of BotNets fight each other for control of the networks. Once compromised, a computer is open to any other hackers who can wrest control from the original master. Networks of thousands of computers can change hands in an instant in these wars.

The simple solution to these problems is for everyone to put their computers behind a good firewall. But many people already have enough difficulty just understanding the basic operation of their computer, and they don't want to make changes when it seems to be working fine. So even when they know they should be securing their network, they don't feel confident they can. Sadly, there isn't a knowledgeable geek in every family.

But they do need help. Even when, with the best of intentions, nontechnical people buy a router with a firewall, they often fail to secure the wireless portion of the network. Out of the frying pan, into the fire. Now they are off the BotNet, but they're serving up free, untraceable wireless goodness to any criminal with wardriving skills. A recent article in the *New York Times* detailed the frustration of investigators who traced the activities of criminals trading in stolen credit-card numbers to a residence, only to find an unsecured wireless router and a completely oblivious family.

Perhaps law enforcement should hire some reformed uber-geeks and hook them up with users who want to help fight crime, but don't have a clue about what a firewall is. But then would that be letting the proverbial fox into the high-tech henhouse?



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to 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: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2005 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Wyser

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, Brian d'Joy, Moshe Bar, Andy Lester

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

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Michael Goodman

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

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT AND GENERAL COUNSEL

Sandra Grayson

SENIOR VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

SENIOR VICE PRESIDENT, CORPORATE MARKETING

Kate Spellman

VICE PRESIDENT, GROUP DIRECTOR INTERNET BUSINESS

Mike Azzara

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES

Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK COMPUTING

ENTERPRISE ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT

MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS

Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

Perl News

Perl 6 Gets Puppy Love

After only two months of development, Autrijus Tang's Pugs—an implementation of the Perl 6 language written in Haskell—has hit its 6.0.12 release, representing “full support for source code written in UTF-8, and Perl 5-compatible regular expression matching as `rx:perl5//`.” Several bug fixes have also been made, and tests and documentation added.

Pugs' unique naming scheme requires that the major and minor version numbers converge to 2pi; the initial release was 6.0, and the next milestone will be reached in the 6.2 release. Currently, the planned milestones are:

- 6.0: Initial release.
- 6.2: Basic IO and control flow elements; mutable variables; assignment.
- 6.28: Classes and traits.
- 6.283: Rules and Grammars.
- 6.2831: Role composition and other runtime features.
- 6.28318: Macros.
- 6.283185: Port Pugs to Perl 6, if needed.

Pugs developer Stevan Little commented on perlmonks.org: “Much of 6.2 is already in 6.0.12 actually... We have basic IO and most of the standard control flow statements (*for*, *if*, *else*, *elsif*, *unless*, *while* & *until* are all there and tested). Mutable variables is getting there (I just did some tests for those today). Assignment is mostly in place as well. I believe that the plan is for 6.2 to hopefully be released shortly after the YAPC::Taiwan Pugs Hack-a-thon.”

The Pugs homepage is <http://pugscodex.org/>.

The Parrot Flies at Midnight

Meanwhile, the Parrot virtual machine (<http://www.parrotcode.org/>) has also been updated, to Version 0.1.12. Parrot is the target for the Perl 6 compiler; it's also designed to execute bytecode for other dynamic languages. Parrot 0.1.2 includes Patrick Michaud's Parrot Grammar Engine. According to the release notes, it also features new string-handling code, including charset and encoding for strings; parts of a generational garbage collector; improved support for Python, separated into dynclasses; and better test coverage and documentation.

In the wake of the new release, Dan Sugalski announced that he's yielding the role of Parrot design lead to Chip Salzenburg, a former Perl pumpking. Leo Totsch will continue as Parrot pumpking. As Dan wrote in his blog (<http://www.sidhe.org/~dan/blog/archives/000391.html>): “Chip's a sharp guy, and I have no worries that between Chip and Leo that Parrot's in good hands. I expect the news'll not take too many people by surprise, as it was past due. I've been essentially missing the past few months, with Real Life (pesky thing that—I think I disapprove) getting in the

way... From now on? Mostly I'm dealing with the other things taking up my time. I'm just a regular user of parrot, albeit one with more knowledge of how it works than is good for my sanity.”

Annotating CPAN

The Perl Foundation has awarded Ivan Tubert-Brohman, author of dozens of CPAN modules, a \$1000 development grant to create a web interface for users to comment on CPAN modules. As Ivan elaborated in his grant proposal (http://www.perlfoundation.org/gc/grants/2005_q1.html):

“There are many modules on CPAN that don't have mailing lists or other discussion venues. Scattered discussions may happen in various places, or some users may post general comments on cpanratings.perl.org. However, there is no central place where users can help each other by commenting on specific features, uses, gotchas, tips, and tricks for all Perl modules. A limitation of sites such as CPANRatings (and to a certain degree other sites that host reviews) is that the comments appear out of the context of the module's documentation, so they are necessarily general unless the comment's author decides to write a long review to establish context. AnnoCPAN intends to fill this gap by allowing users to add public annotations on the margin of the documentation of every module on CPAN.”

The grant period is for one or two months, so AnnoCPAN should debut soon.

Upcoming Events

The second Italian Perl Workshop has been announced for June 23–24 at the University of Pisa's Polo Fibonacci. The registration fee of 50 Euros (35 Euros for students) covers attendance at the workshop; printed proceedings from every talk; two lunches, one dinner, and four coffee breaks; as well as a T-shirt. See <http://www.perl.it/workshop/> for details.

Registration is now open for this year's Yet Another Perl Conference, North America, to be held in Toronto, Canada from June 27–29 at a facility within the University of Toronto. The full registration fee is \$85. Hotel discounts for groups will be available until early May. Also, YAPC::NA is still accepting proposals for talks; see <http://yapc.org/America/> for more.

Lastly, OSCON Europe is tentatively scheduled for October 17–20 in Amsterdam. Conference planner Nathan Torkington wrote in his blog (<http://oreillynet.com/pub/wlg/6616>): “Everything about it is still in the planning stage, though I'm getting a feel for the focus. It'll be smaller than the Portland OSCON, though we hope it'll grow over time as attendance grows. As with OSCON, we expect most of the attendees to be from companies using or thinking about using open source, and speakers to be the alpha geeks building the open-source projects. More details, obviously, when the CFP comes out.”

Creating Self-Contained Perl Executables, Part I

Being a Perl programmer, one of my biggest frustrations with using Windows is the absence of a Perl interpreter. On many occasions, I had to install ActivePerl (<http://www.activestate.com/>) just so I could use my Perl scripts. A number of times, I have asked myself, “Wouldn’t it be nice to have a Perl script compiler to convert my scripts into self-contained executables so that I won’t have to worry about a Perl interpreter?” Fortunately for me, I discovered *LibZip*.

This month, I’ll discuss *LibZip*. Next month, I’ll follow up with a discussion of PAR, the Perl Archive Toolkit, which is another way to create Perl executables. Obviously, you’ll need a working C compiler to convert your Perl scripts into self-contained executables, as discussed here.

LibZip

LibZip is the brainchild of Graciliano Monteiro Passos and is available from <http://search.cpan.org/CPAN/authors/id/G/GM/GMPASSOS>. In Graciliano’s own words, the goal of *LibZip* is to “create very low weight executables” out of Perl scripts. To achieve this, *LibZip* bundles all modules needed by a Perl script into one big file, called *lib.zip*, and creates a native code equivalent of the script. When the native code is run, it uncompresses *lib.zip* into a temporary folder and uses the files in this folder, thereby eliminating the need for a Perl interpreter. The downside, of course, is that the native code won’t run without *lib.zip*.

Using LibZip

LibZip needs three other modules before it can be useful: *Pod::Stripper*, *Compress::Zlib*, and *Archive::Zip*, all available from CPAN. Assuming you have successfully installed *LibZip*, using it is a piece of cake. To use *LibZip*, follow these simple steps. Let’s convert *dos2unix.pl* (Listing 1) into a Windows native executable, *dos2unix.exe*:

1. Scan all modules that *dos2unix.pl* needs, including dependencies:

```
perl -MLibZip::Scan dos2unix.pl
```

This will build *libzip.modules*. *LibZip* will use this file to create the EXE file in the next step.

2. Build the library, *lib.zip*, and executable, *dos2unix.exe*:

```
libzip -o dos2unix.pl -lzw -striplib
```

The *-o* option tells the batch file, *libzip*, to create an EXE file. Note also that the name of the Perl script must immediately follow the option *-o*. The *-lzw* option tells *libzip* to use LZW compression. An interesting benefit of compression, albeit a trivial one, is that you can now obfuscate your code with it! *-striplib* tells *libzip* to remove PODs from *lib.zip*.

That’s it! One more thing, though. After the last step, *LibZip* will import *perl56.dll* and *perl58.dll* from your Perl interpreter’s installation folder into the folder where you created *lib.zip* and *dos2unix.exe*. So, go ahead, run your shiny new executable! Rename your Perl installation folder to something else and see if *dos2unix.exe* will run without a Perl interpreter.

A Word Of Warning

I usually use diagnostics in my scripts to make debugging easier. But, for some reason, I can’t get a functioning executable (both in Windows and Linux) if I use diagnostics. Not using diagnostics solves the problem.

Compiling Perl/Tk Scripts

Compiling console Perl scripts are straightforward, but compiling *Perl/Tk* scripts is a bit problematic. If you compile *perl-tk.pl* (Listing 2), for example:

```
perl -MLibZip::Scan perl-tk.pl
```

```
libzip -o perl-tk.pl -lzw -striplib
```

the generated executable won’t run. Apparently, *Tk* performs some magic that *LibZip* fails to see. Specifically, *utf8_heavy.pl* and the whole *unicore* directory were not included in *libzip.modules* during the scan. So when *lib.zip* was finally built, some files were missing. The fix, therefore, is to edit *libzip.modules* manually and add *utf8_heavy.pl* and *unicore*. Recompiling *perl-tk.pl* should now be a breeze.

You may also use the *-gui* option, aside from *-lzw* and *-striplib*, to create a nonconsole executable.

```
libzip -o perl-tk.pl -lzw -striplib -gui
```

Julius is a systems administrator for Ayala Systems Technology. He can be contacted at jcdunque@lycos.com.

You are encouraged to use *-gui* if you're compiling a *Perl/Tk* script.

UPX

LibZip can also use what is known as UPX, short for "Ultimate Packer for executables" to compress *lib.zip* and executables even more. UPX's inventors, Markus F.X.J. Oberhumer and László Molnár, claim that UPX is a "free, portable, extendable, high-performance executable packer for several different executable formats. It achieves an excellent compression ratio and offers very fast decompression." <http://upx.sourceforge.net/> is UPX's official web site where you'll find versions of UPX for different platforms.

Using UPX within *LibZip* is also painless:

```
libzip -o perl-tk.pl -lzw -striplib -gui -upx best
                                -upxlib best
```

Option *-upx* will compress the executable, while *-upxlib* will compress *lib.zip*. The *best* argument to *-upx* and *-upxlib* tells *libzip* to call UPX using the best possible compression.

Because UPX is also a standalone program, you may also use it independently from *LibZip*. For example, you may type Step 2 as:

```
libzip -o perl-tk.pl -lzw -striplib -gui
```

and then follow it up with:

```
upx -best perl-tk.exe perl56.dll perl58.dll
```

perlcc—Perl's Own Compiler

How about Perl's own compiler, *perlcc*? Based from my personal experience, I have yet to produce a working executable that is compiled with *perlcc*. Indeed, the Perl documentation says that *perlcc*'s output is not guaranteed to work. The man page further says "the whole codegen suite (*perlcc* included) should be considered very experimental. Use for production purposes is strongly discouraged." Aside from that, *perlcc* takes several minutes to compile even a simple script.

Compiling Scripts for Linux

You may ask, "Can *LibZip* also create executables for Linux?" The answer is a resounding "Yes." The steps are the same, but without *perl56.dll* and *perl58.dll* being generated (for the obvious reason).

PAR—Another Executable Maker

What if you dislike having to bundle your executables with a separate library and prefer, instead, to have just a single executable? The *LibZip* man page mentions a similar tool, *PAR*, or the Perl Archive Toolkit, from Autrijus Tang. I'll discuss *PAR* next month.

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

Listing 1

```
#!/usr/local/bin/perl

# Julius C. Duque

use strict;
use warnings;
use Getopt::Long;

my ($format, $help) = ();

GetOptions(
    "format=s" => \$format,    # s -> takes mandatory string argument
    "help"     => \$help      # optional switch
);

if (!($format or $help)) {
    if ($^O eq "MSWin32") {
        $0 =~ s/.*\\.\\/g;    # Windows
    } else {
        $0 =~ s/.*\\.\\/g;
    }

    print "Usage: $0 --format=unix file1 [file2] [file3] ...\n";
    print "      $0 -f=unix file1 [file2] [file3] ...\n";
    print "\n";
    print "      $0 --format=dos file1 [file2] [file3] ...\n";
    print "      $0 -f=dos file1 [file2] [file3] ...\n";
    print "\nOriginal file(s) will be overwritten\n";
    exit 1;
}

foreach my $infile (@ARGV) {
    print "Converting $infile to $format format...\n";
    open INFILE, $infile;
    open OUTFILE, ">temp.$infile";
    binmode INFILE;
    binmode OUTFILE;
    while (<INFILE>) {
        $_ =~ s/\015\012/\012/g if ($format =~ /u/);
        $_ =~ s/\012\015\012/g if ($format =~ /d/);
        print OUTFILE;
    }

    close INFILE;
    close OUTFILE;
}
```

```
rename "temp.$infile", "$infile";
}
```

Listing 2

```
#!/usr/local/bin/perl

use strict;
use warnings;
use Tk;
use Tk::Balloon;

my $INDENT_DEF = 0;
my $LWIDTH_DEF = 70;
my $VERSION = "1.4.3";
my $TITLE = "Perl/Tk Example $VERSION";
my $AUTHOR = "Julius C. Duque";
my $indent = $INDENT_DEF;
my $newline = 1;
my $shyphenate = 1;
my $width = $LWIDTH_DEF;
my ($BOTH, $LEFT, $RIGHT, $CENTERED) = (1, 2, 3, 4);
my $format_choice = $BOTH;
my ($infile, $outfile) = ();

my $mw = new MainWindow();
drawButtons();
Tk::MainLoop();

sub processfile
{
    open INFILE, $infile;
    open OUTFILE, "> $outfile";
    while (<INFILE>) {
        print OUTFILE;
    }

    close INFILE;
    close OUTFILE;

    printMessage("info", "OK", "File was successfully saved.");
}

sub drawButtons
{
    $mw->title($TITLE);

    # Status bar widget
    my $status = $mw->Label(-width => 70, -relief => "sunken",
        -anchor => "w")->pack(-side => "bottom", -padx => 1, -pady => 1,
```

```

-fill => "x");

# Create balloon widget
my $b = $mw->Balloon(-statusbar => $status);

# Create menu bar frame
my $menubar = $mw->Frame(-borderwidth => 4, -relief => "ridge")->
    pack(-side => "top", -fill => "x");

# Create Open File button
my $openfilebutton = $menubar->
    Button(-text => "Open File", -relief => "raised", -width => 10,
        -command => [\&fileDialog, $mw, "open"])->pack(-side => "left");

$b->attach($openfilebutton, -msg => "Open a file");

# Create Save File button
my $savefilebutton = $menubar->
    Button(-text => "Save To File", -relief => "raised", -width => 10,
        -command => sub {
            if (defined $infile and $infile ne "") {
                fileDialog($mw, "save");
            } else {
                printMessage("warning", "OK",
                    "You must open a file first");
            }
        })->pack(-side => "left");

$b->attach($savefilebutton,
    -msg => "Proceed with saving a file");

# Create About button
my $aboutbutton = $menubar->Button(-text => "About",
    -relief => "raised", -width => 10,
    -command => [\&printMessage, "info", "OK",
        "A Perl/Tk script created by $AUTHOR"])->pack(-side => "left");

$b->attach($aboutbutton,
    -msg => "$TITLE created by $AUTHOR");

# Create Quit button
my $quitbutton = $menubar->Button(-text => "Dismiss",
    -relief => "raised", -width => 10, -command => sub { exit })->
    pack(-side => "right");

$b->attach($quitbutton, -msg => "Quit this Perl/Tk script");

my $both = $mw->Radiobutton(-variable => \$format_choice,
    -value => $BOTH, -text => "Radio Button 1")->
    pack(-side => "top", -anchor => "w");

$b->attach($both, -msg => "Radio Button 1");

my $left = $mw->Radiobutton(-variable => \$format_choice,
    -value => $LEFT, -text => "Radio Button 2")->pack(-side => "top",
    -anchor => "w");

$b->attach($left, -msg => "Radio Button 2");

my $right = $mw->Radiobutton(-variable => \$format_choice,
    -value => $RIGHT, -text => "Radio Button 3")->pack(-side => "top",
    -anchor => "w");

$b->attach($right, -msg => "Radio Button 3");

my $centered = $mw->Radiobutton(-variable => \$format_choice,
    -value => $CENTERED, -text => "Radio Button 4")->pack(-side => "top",
    -anchor => "w");

$b->attach($centered, -msg => "Radio Button 4");

$both->select; # Set default to $both

my $chknewline = $mw->Checkbutton(-variable => \$newline,
    -text => "Check Box 1")->
    pack(-side => "top", -anchor => "w");

$b->attach($chknewline,
    -msg => "Check Box 2");

$chknewline->select; # Set default to $newline

my $chkhyphen = $mw->Checkbutton(-variable => \$hyphenate,
    -text => "Check Box 2")->pack(-side => "top", -anchor => "w");

```

```

$b->attach($chkhyphen, -msg => "Check Button 2");

$chknewline->select; # Set default to $hyphenate

my $f = $mw->Frame->pack(-side => "left");

my $l = $f->Label(-text => "Indentation: ", -justify => "left");

$b->attach($l, -msg => "Blah blah blah...");

Tk::grid($l, -row => 0, -column => 0);

my $tindent = $f->Entry(-width => 2, -textvariable => \$indent,
    -justify => "right");

$b->attach($tindent,
    -msg => "Number of spaces at the start of every paragraph");

Tk::grid($tindent, -row => 0, -column => 1);

$l = $f->Label(-text => "characters (default: $INDENT_DEF)",
    -justify => "left");

$b->attach($l,
    -msg => "Number of spaces at the beginning of each paragraph");

Tk::grid($l, -row => 0, -column => 2);

$l = $f->Label(-text => "Line width: ", -justify => "left");
Tk::grid($l, -row => 1, -column => 0);
$b->attach($l, -msg => "Maximum length of every line");

my $tlwidth = $f->Entry(-width => 2, -textvariable => \$width,
    -justify => "right");

$b->attach($tlwidth, -msg => "Maximum length of every line");
Tk::grid($tlwidth, -row => 1, -column => 1);

$l = $f->Label(-text => "characters (default: $LWIDTH_DEF)",
    -justify => "left");

$b->attach($l, -msg => "Maximum length of every line");
Tk::grid($l, -row => 1, -column => 2);
}

sub printMessage
{
    my ($icon, $type, $outputmsg) = @_;
    my $msg = $mw->messageBox(-icon => $icon, -type => $type,
        -title => $TITLE, -message => $outputmsg);
}

sub fileDialog {
    my ($w, $operation) = @_;
    my @types = (["Text files", [qw/.txt .doc/]],
        ["Text files", "", "TEXT"],
        ["All files", ""]);

    if ($operation eq "open") {
        $infile = $w->getOpenFile(-filetypes => \@types);
    }

    if ($operation eq "save") {
        $outfile = $w->getSaveFile(-filetypes => \@types,
            -initialfile => "Untitled",
            -defaulttextextension => ".txt");

        processfile() if (defined $outfile and $outfile ne "");
    }
}

```

TPJ

Monitoring Network Traffic Revisited

In the July 2004 issue of *TPJ*, Robert Casey wrote an excellent article on using the *Net::Pcap* module to capture and process network traffic. Robert assumed that the reader was familiar with the inner workings of libpcap, the C library to which *Net::Pcap* provides a Perl interface. The API provided is very C-like, as *Net::Pcap* was purposely designed to match libpcap's C API call-for-call and, as a result, is very complex. Although this gives the Perl programmer total control over what's going on, the requirement to work at the "C level" can often make things harder than they need to be. And trust me, learning the inner workings of libpcap—documented in the *pcap(3)* and *Net::Pcap* man pages—is not for the faint hearted.

Is There Something Easier than *Net::Pcap*?

With this in mind, Tim Potter, the author of *Net::Pcap*, produced the companion module *Net::PcapUtils* to encapsulate most of the *Net::Pcap* functionality, concentrating on providing a set of default initialization values. Only three functions make up the entire *Net::PcapUtils* API, as opposed to the more than 15 provided by *Net::Pcap*. In this article, I'll rewrite Robert's code to use *Net::PcapUtils* instead of *Net::Pcap*. In doing so, I'll use just two of the former module's functions. What's gained is a program that is smaller and, one hopes, easier to understand and maintain. What's lost is the ability to minutely customize the initialization of the packet-capturing environment. By taking advantage of *Net::PcapUtils* default initialization values, we can concentrate on processing the captured packets as opposed to worrying about specific initialization details.

Rewriting Robert's Analyzer with *Net::PcapUtils*

Rather than use the *Net::Pcap* module at the top of the program, my code (in Listing 1) brings in *Net::PcapUtils* instead, together with the other required modules:

```
use Net::PcapUtils;
use NetPacket::Ethernet;
use NetPacket::IP;
use NetPacket::TCP;
```

As with Robert's code, the *NetPacket::** modules—also by Tim Potter—are used to decode chunks of network traffic as they are captured. After setting a Boolean constant, I define two packet-capturing constant values:

Paul lectures at the Institute of Technology, Carlow, in Ireland, and can be reached at paul.barry@itcarlow.ie. His web site is <http://glasnost.itcarlow.ie/~barryp/>.

```
use constant CAPTURE_FILTER =>
    '(dst 127.0.0.1) && (tcp[13] & 2 != 0)';
use constant CAPTURE_DEVICE => 'eth0';
```

The first, *CAPTURE_FILTER*, holds the filter string to use when deciding when to process a captured packet. This string is identical to that used by Robert and it filters on TCP segments that have the SYN flag set. Be sure to change the 127.0.0.1 address to the IP address of the interface you want to capture traffic on. The *CAPTURE_DEVICE* constant identifies the network card to use when capturing and is set to *eth0*. Robert's code was able to work this value out for itself, but as *eth0* is the default network device on most machines, it's quicker and easier to hard-code this value into the program. If you later want to capture on *eth1* instead, simply change this constant. In fact, we need not specify *eth0* at all, as *Net::PcapUtils* uses it by default. However, I find the extra effort required to define the constant to be worthwhile. Changing the filter is neater now, too, as constant values are easily identified at the top of the source code.

The just-defined constant values are used on the very next piece of code, which calls the *Net::PcapUtils* *open* function to prepare the network card to capture traffic:

```
my $pkt_descriptor = Net::PcapUtils::open(
    FILTER => CAPTURE_FILTER,
    DEV    => CAPTURE_DEVICE
);
```

The *open* function does a number of things for us. The most useful is that it automatically puts the identified network card into promiscuous mode, which saves us the trouble. If the call to *open* succeeds, a reference to a valid "packet descriptor" is returned to our code. If the call fails, the reference is undefined. My code checks the status of the packet descriptor and exits with an appropriate message if something goes wrong. Typically, the code in this *if* statement fires if an attempt is made to put the network card in promiscuous mode while running as a regular user on Linux/UNIX systems. Only root can put the network card into promiscuous mode:

```
if ( !ref( $pkt_descriptor ) )
{
    print "Net::PcapUtils::open returned:
        $pkt_descriptor\n";
    exit;
}
```

With a valid packet descriptor created, my code enters an infinite loop, which captures and processes packets:

```
while( TRUE )
{
    my ( $packet, %header )
        = Net::PcapUtils::next( $pkt_descriptor );
    syn_packets( $packet );
}
```

The *next* function from the *Net::PcapUtils* module blocks, waiting for a packet to arrive on the network interface associated with the packet descriptor. When one does, *next* returns the entire packet (as a scalar), as well as its header information (as a hash). The entire packet is sent to the *syn_packets* subroutine for processing, which differs only slightly from that provided by Robert (in how it processes its parameters).

In my view, my program is more Perlish than that written by Robert, even though both programs do roughly the same thing. I hope it is easier to understand, maintain, and extend.

Building Another Analyzer

Writing filter specifications, as defined in the *CAPTURE_FILTER* constant in Listing 1 (and borrowed from Robert's code), requires a little bit of research and learning. Fortunately, the *tcpdump*(8) man page has all the details. To continue to keep things simple, here's a straightforward filter specification that captures all TCP traffic on port 8080:

```
( tcp port 8080 )
```

In Listing 2, this filter is used to create a simple analyzer that captures and displays any and all traffic that matches the specification. Perhaps you've implemented a custom web server or application that communicates on port 8080, and you need to debug (or postprocess) the traffic generated. As this version of the analyzer isn't interested in displaying the *to* and *from* IP address information (unlike Robert's code), we can dispense with all of the decoding when the captured packet is processed. Instead, we simply strip away any header information associated with lower level protocols—IP and Ethernet, in this case—and decode what's left as a chunk as TCP traffic. Here's how to do this:

```
sub process_packet {
    my $packet = shift;
    my $tcp = NetPacket::TCP->decode(
        ip_strip ( eth_strip ( $packet ) )
    );
    print $tcp->{ data };
}
```

Note how, in this program, what was called *syn_packet* has been changed to the more generic *process_packet*. The *eth_strip* and *ip_strip* functions come from their respective *NetPacket* modules.

These are imported into the code using the predefined export tags as follows:

```
use NetPacket::Ethernet qw( :strip );
use NetPacket::IP qw( :strip );
```

When *next* returns a captured packet, it returns only the first 100 bytes captured, which is a *Net::PcapUtils* default. This is enough packet data if all you are interested in is the header data; for instance, IP addresses, protocol ports, or Ethernet types. However, if what you are interested in is the actual data sent—as well as the protocol-generated data—then you need to tell *Net::PcapUtils* to return more of the captured packet. In Listing 2, I define another constant, *CAPTURE_AMOUNT*, and set it to the value of 1500, which is the largest chunk of traffic that an Ethernet network interface can transmit at any one time:

```
use constant CAPTURE_AMOUNT => 1500;
```

This constant is then used as part of the call to *open* and sets the value for the *SNAPLEN* parameter:

```
my $pkt_descriptor = Net::PcapUtils::open(
    FILTER    => CAPTURE_FILTER,
    DEV       => CAPTURE_DEVICE,
    SNAPLEN   => CAPTURE_AMOUNT
);
```

When the program in Listing 2 is executed (as root, you'll recall), all of the output generated by the *print* statement can be piped to a file for later perusal at your leisure. The program in Listing 2 can be made even more generic by adding some simple command-line processing to set the values for the constants using the standard modules *Getopt::Std*, *Getopt::Long*, or good ol' *shift*. Doing so is left as an exercise for the keen reader.

Learning More

For more on *Net::PcapUtils* and packet capturing, refer to Chapter 2 of my first book *Programming the Network with Perl*; and if you want to use this technology to decode traffic for a protocol not covered by the *NetPacket::** modules (such as DNS), see issue 0.6 of *The Perl Review*. And be sure to check out Robert's original article—it's a great read.

References

The *tcpdump*(8) man page.

Casey, Robert. "Monitoring Network Traffic with *Net::Pcap*," *TPJ*, July 2004.

Barry, Paul. "Who's Doing What? Analyzing Ethernet LAN Traffic," *The Perl Review*, Volume 0, Issue 6, November 2002.

Barry, Paul. *Programming the Network with Perl*, Wiley, 2002, ISBN 0471486701.

TPJ

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

Listing 1

```
#!/usr/bin/perl -w
```

```
use strict;
```

```
# Use the correct set of modules.
use Net::PcapUtils;
use NetPacket::Ethernet;
use NetPacket::IP;
use NetPacket::TCP;
```

```
use constant TRUE => 1;
```

```
use constant CAPTURE_FILTER => '(dst 127.0.0.1) && (tcp[13] & 2 != 0)';
use constant CAPTURE_DEVICE => 'eth0';
```

```
# Open the network card for capturing.
```

```
my $pkt_descriptor = Net::PcapUtils::open(
    FILTER => CAPTURE_FILTER,
    DEV    => CAPTURE_DEVICE
);
```

```
# Check that the card "opened" OK.
```

```
if ( !ref( $pkt_descriptor ) )
{
```



```

print "Net::PcapUtils::open returned: $pkt_descriptor\n";
exit;
}

while( TRUE )    # i.e., forever, or until "killed" ...
{
    # Capture a packet from the network card.
    my ( $packet, $header ) = Net::PcapUtils::next( $pkt_descriptor );

    # Process the captured packet.
    syn_packets( $packet );
}

sub syn_packets {
    my $packet = shift;

    # Strip Ethernet encapsulation of captured packet.
    my $ether_data = NetPacket::Ethernet::strip($packet);

    # Decode contents of TCP/IP packet contained within
    # captured Ethernet packet.
    my $ip = NetPacket::IP->decode($ether_data);
    my $tcp = NetPacket::TCP->decode($ip->{'data'});

    # Print all out where its coming from and where its
    # going to!
    print $ip->{'src_ip'}, ":", $tcp->{'src_port'}, " -> ",
        $ip->{'dest_ip'}, ":", $tcp->{'dest_port'}, "\n";
}

```

Listing 2

```
#!/usr/bin/perl -w
```

```

use Net::PcapUtils;
use NetPacket::Ethernet qw( :strip );
use NetPacket::IP qw( :strip );
use NetPacket::TCP;
use strict;

```

```
use constant TRUE => 1;
```

```

use constant CAPTURE_FILTER => '(tcp port 8080)';
use constant CAPTURE_DEVICE => 'eth0';
use constant CAPTURE_AMOUNT => 1500;

my $pkt_descriptor = Net::PcapUtils::open(
    FILTER => CAPTURE_FILTER,
    DEV    => CAPTURE_DEVICE,
    SNAPLEN => CAPTURE_AMOUNT
);

if ( !ref( $pkt_descriptor ) )
{
    print "Net::PcapUtils::open returned:
        $pkt_descriptor\n";
    exit;
}

while( TRUE )
{
    my ( $packet, $header ) =
        Net::PcapUtils::next( $pkt_descriptor );

    process_packet( $packet );
}

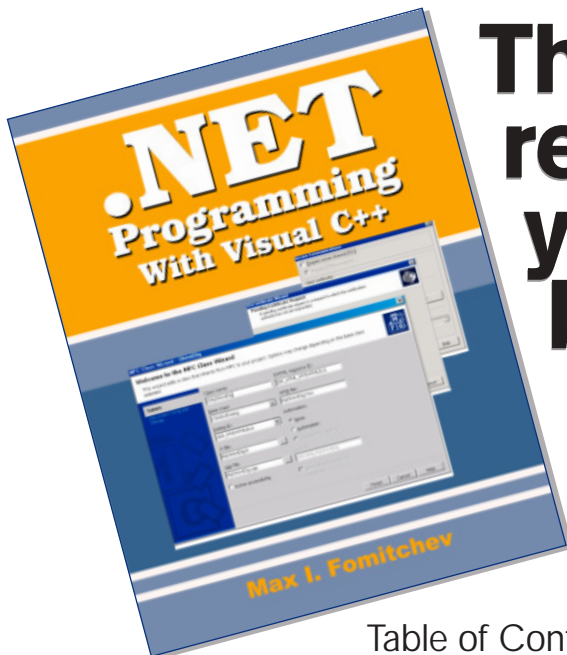
sub process_packet {
    my $packet = shift;

    my $tcp = NetPacket::TCP->decode(
        ip_strip ( eth_strip ( $packet ) )
    );

    print $tcp->{'data'};
}

```

TPJ



The .NET resource you've been waiting for!



- Delivered in PDF format.
- Packed with C++ code examples.
- Thousands of lines of source code.
- A complete reference to the .NET Framework

Table of Contents and sample chapter available at:
<http://www.ddj.com/dotnetbook/>

Get your copy now!

Available via **download** for just **\$19.95**
 or
 on **CD-ROM** for only **\$24.95** (plus s/h).

Coding for Readability

While contributing to several Perl beginner e-mail lists, I noticed that many posts are from people who have little or no formal training in computer programming. As a fellow uneducated programmer, I think poor habits can be unlearned.

I know good code when I see it. It doesn't have to conform to my personal programming style—my style is biased by my history of programming. At 43, I come from a Basic background with little formal training as a computer programmer. When I started using Perl, I had to abandon some of the style I found common, but I had previously switched between enough languages to find that relatively easy.

One important concept when writing programs is to keep the reader in mind. Computer programs are not written for machines. They are written for people, and therefore, style matters. We too often concentrate on the needs of the Perl interpreter, rather than the needs of people. If we write programs for people, we'll become better programmers.

Coding is Writing

Good programs tell a story. They have a setting, a plot, and characters. Good stories are built on writing tools: grammar, language, style, and the like. When programming, we use similar tools: logic, reuse, style, and so on. Better programmers are those better able to use existing tools and create missing tools.

One key to writing good stories is developing a writing style. Authors develop a basic writing style they prefer, and then tend to write that way consistently. Reading a few stories by one author tells us a lot about the author as well as the characters in the book.

Good programmers need the same commitment. Develop your style. Refine it with logical rules. Stick to it consistently. When someone reads your program, they can tell how organized, consistent, and concise you are. Readers can learn a lot about you from your programs.

Some stories are just dialogue in that the author doesn't add description; for example, Shakespeare. You have characters acting out parts, but everything descriptive is said through the actions of the players, the scenery, and their words. Other sto-

ries are written with description mixed in with dialogue. You know what a sunset looks like because the author describes it in detail.

One key to writing good code is mixing just enough code (dialogue) with comments (description). Some code needs comments—we can't tell what is going on by simply reading the code.

*Whatever you do, you should be consistent. If you use a cuddled
} else { in one part of the script,
use it in all parts*

Meanwhile, other code has comments that are unneeded or easily sidetrack the reader.

There are some elements of my personal programming style that are my personal bias (avoiding excessive comments, parentheses, and underscores in variable names) and some things that should be a part of everyone's programming style (well-structured subroutines of limited, clearly defined purpose, and error checking).

Whatever you do, you should be consistent. If you use a cuddled `} else {` in one part of script, use it in all parts. If you prefer `$VariablesLikeThese` to `$variables_like_these`, that's fine—just avoid mixing both styles in one script. I like to maintain the same naming conventions for subroutines as I use for variables.

Whitespace and Comments

Another area to consider is whitespace. Too little whitespace makes scripts look disorganized and confusing. Imagine the difficulty in

Charles is a freelance web programmer and can be contacted at cclarkson@htcomp.net.

reading this article if it had only one paragraph. You would be distracted and have a tough time finding your place. Because I group my words into sentences, and those sentences into paragraphs in (one hopes) meaningful groups, you can more easily digest my meaning.

Readability doesn't always mean lots of comments. Most people use either too many comments or not enough. Beginners who use too many comments are often making up for poorly defined style. Those who use too few comments either don't comment at all or falsely believe they will understand everything when they come back to it later. When writing a comment, I use formal sentences, correct grammar, and punctuation. I place a space between the # and the first word and avoid unexplained jargon. A good comment can make an especially tough piece of code easy to comprehend, and an inappropriate comment can turn good code to the dark side.

Don't use comments to make up for poor naming conventions. If you can give a subroutine a reasonably concise name that adequately describes its function, you might not need to comment it. Nor should your comments repeat the obvious.

Getting Help

If you're a Perl beginner, style becomes especially important when you (inevitably) go to the Internet for help. A readable program will allow others to parse your code for problems more quickly, and will make it more likely that you will get the help you need. Avoid using long lines if you are preparing to ask for help. When I write posts for USENET or e-mail newsgroups, I try to keep my lines shorter than 66 characters across. This aids anyone replying by not having indented lines wrap. In practice, I usually use 72-character-length lines as they print without line wrapping. I mention this because many people use trailing comments like those common to assembler programming. When asking a question on a list, keep your audience in mind (Perl gurus and programmers). Paying attention to their needs gets questions answered.

Dissecting a Subroutine

Let's look at a few common mistakes and their corrections. Take a look at Example 1, which is a subroutine that was pulled off a larger program on USENET. Pay attention to the style more than the errors in the use of Perl.

Do we really need a subroutine description? More importantly, do we need *that* subroutine description? Look at the name of the subroutine; *checkLogin* is pretty descriptive. Most readers will immediately know what the script is doing. Let's peek ahead at the other comments. The next one is a trailing comment (I've removed the code part for brevity—see the full line in Example 1):

```
# split $line, using : as delimiter
```

Note that the comment was written for a different revision of the script. These types of comments are a maintenance problem. We can safely assume that others reading the script will know that *split()* splits something. We can also assume that one of the arguments to the function may be the character(s) to split on. We can safely drop this comment and be assured the program will be as easy to read. We can also safely drop the last comment three lines down. Also, using better names for the fields could lead to more descriptive code with no comment:

```
my( $user, $password ) = ( split /\t/, $_ )[1, 2];
```

Notice the author's style in determining variable names. It looks sort of like this. The subroutine name uses a *lowercaseFirst-MixedCase* style. One key to good style is consistency. Although I dislike mixed-case style, let's stick to it. *\$loginname* becomes *\$loginName* and *\$matchokay* becomes *\$matchOkay*.

```
# function that will check to ensure the username are
# password match and are ok
sub checkLogin {
    my ($loginname) = @_ ;

    open USERFILE, " $SETTINGS{'userfile'}"
        || die "Failed to open file: $!";
    if ($^O ne "MSWin32") { flock(USERFILE, LOCK_EX); }
    while (<USERFILE>) {
        my @fields = split(/\t/, $_); # split $line, using : as delimiter
        if (@fields[1] =~ m/^$loginname$/) {
            if (@fields[2] =~ m/^$password$/) {
                $matchokay='true'; # login name matches password
            }
        }
    }
    close(USERFILE);

    if ($matchokay) { return $matchokay; }
    else { return ''; }
}
```

Example 1: Hard-to-read code.

```
(a)
if ($^O ne "MSWin32") { flock(USERFILE, LOCK_EX); }

    if (@fields[1] =~ m/^$loginname$/) {
        if (@fields[2] =~ m/^$password$/) {
            $matchokay='true'; # login name matches password
        }
    }

    if ($matchokay) { return $matchokay; }
    else { return ''; }

(b)
flock USERFILE, LOCK_EX if $^O ne 'MSWin32';

    if ( $user =~ m/^$loginName$/) {
        if ( $password =~ m/^$password$/) {
            $matchOkay = 'true';
        }
    }

    if ( $matchokay ) {
        return $matchokay;
    } else {
        return '';
    }
```

Example 2: (a) Poorly indented code; (b) Improved indenting.

```
sub checkLogin {

    my $loginName = shift;

    # Do not clobber open FH file handles
    local *FH;

    my $file = $SETTINGS{userfile};
    open FH, $file or die qq(Cannot open "$file": $!);
    flock FH, LOCK_EX if $^O ne 'MSWin32';

    while ( <FH> ) {
        my( $user, $password ) = ( split /\t/, $_ )[1, 2];
        if ( $user =~ m/^$loginName$/) {
            if ( $password =~ m/^$password$/) {
                return 'true';
            }
        }
    }
    close FH;
    return '';
}
```

Example 3: Eliminating a flag variable.

```

#
# This sub depends on the Fcntl.pm :flock constants to
# function properly.
#

sub isValidUser {

    my( $givenUser, $givenPassword, $file ) = @_;

    # Do not clobber open FH file handles
    local *FH;

    open FH, $file or die qq(Cannot open "$file": $!);
    flock FH, LOCK_EX if $^O ne 'MSWin32';

    while ( <FH> ) {
        my( $user, $password ) = ( split /\t/, $_ )[1, 2];

        return 1 if $user == m/^\$givenUser$/
            && $password =~ m/^\$givenPassword$/;
    }
    close FH;
    return;
}

```

Example 4: Renamed subroutine with better parameter-passing behavior.

The author uses a two-space indent inconsistently. We'll stick with the two-space indent, but we'll need to fix the inconsistent *if* statement structure. Let's separate those statements out with the original formatting left intact; see Example 2(a)

We have two distinct styles here. Notice that the inner *if* in the example is not indented two spaces. In fact, it is out-dented two spaces. That's inconsistent with indenting in the rest of the subroutine. Notice also that it is sometimes okay to cuddle the result statement while other times it is not. Perhaps that's because of the comment. Let's rewrite these, using a consistent style; see Example 2(b).

My style includes opening all files the same way. Unless I'm using the filehandle somewhere far from its opening, or the name is already taken, *FH* is just fine as the filehandle name. I have an editor macro just for opening files in the same way each time. To use it in this subroutine, we would define *\$file* as *SETTINGS{userfile}*:

```

my $file = '';
open FH, $file or die qq(Cannot open "$file": $!);

close FH;

```

I have a consistent variable use policy in my style. I attempt to use the least number of variables that will allow my script to run and remain human readable. One variable type I can often avoid is the flag. A flag is a convenience variable that marks an expression and allows us to later perform an action based on the value of that flag.

Many times, flags are unavoidable. In this subroutine, however, the *\$matchOkay* flag is easily avoided. We set its value in a loop and later act upon that value. In this case, it is needed probably because the author does not understand how to localize a file handle to a subroutine. Without correcting the other errors in this sub, let's look at a localized file handle without the *\$matchOkay* flag (Example 3). *FH* will close when the *sub* returns a True value.

Another beginner style mistake involves return values. For functions that return truth, return a result that Perl interprets as True. Typically, this is the number 1. For a false result, return an undefined value. The *return* function returns undefined by default. So *return 1*; is preferable to *return 'true'*; and *return*; is preferable to *return ''*;

```

if ( $user == m/^\$loginName$/ ) {
    if ( $password =~ m/^\$password$/ ) {
        return 1;
    }
}

```

Becomes this:

```

if ( $user == m/^\$loginName$/
    && $password =~ m/^\$password$/ ) {
    return 1;
}

```

Becomes this:

```

return 1 if $user == m/^\$loginName$/
    && $password =~ m/^\$password$/;

```

Then goes back in as this:

```

while ( <FH> ) {
    my( $user, $password ) = ( split /\t/, $_ )[1, 2];
    return 1 if $user == m/^\$loginName$/
        && $password =~ m/^\$password$/;
}

```

Figure 1: Transforming *if* statements for readability.

When you have two *if* statements stacked inside each other the way they are in our example, it's generally best to combine them. Combining them into a one-statement *if* block can further aid readability with the use of a statement modifier. I also prefer to vertically align operators if they do not detract from reading horizontally; see Figure 1.

Design Issues

The final rules I would like to apply to this subroutine are not just style rules—they also add to readability, but they touch on larger rules for good overall program design. A subroutine is really just one tiny step down from an object. It should be as self contained as possible. It should be passed the values it processes, and should pass new values on return, except under special circumstances. A well-written subroutine should not be affected by changes in the external environment, and should only affect the external environment in well-defined, predetermined ways.

checkLogin() uses values that were never passed in (*\$matchokay*, *SETTINGS{userfile}*, and *\$password*). The original version seemed to affect a variable (*\$matchokay*) which was not in the subroutine scope. It also opened a filehandle (*USERFILE*) without any checks to find if it was being used somewhere else. Finally, the name seems poorly chosen. Use might dictate something else, but I would prefer something more descriptive such as *isValidUser()*; see Example 4.

There are still two things that bother me. Those regexes could fail if the password or the user name have special characters in them. Frankly, I would have gone with the *eq* operator instead. But that's definitely outside the realm of style, so I'll leave that alone.

Basic coding style is about more than just making your work look pretty. It will make your code easier for other programmers to follow, will allow you to get the help you need more quickly, and can even help you discover logical flaws in your own code. If you follow some of these style rules, you'll be well on your way to becoming a better Perl programmer.



Tate: An iPhoto Gallery Exporter in Perl

Simon Cozens

Recently I took a photography course, and as a result I've gotten into taking a lot of photographs. I put some of them on my web site for public view, but this has always been a tedious exercise: I choose the album in iPhoto, then have it generate the scaled images, thumbnails, and HTML for the album, and then upload the whole gallery to my server via scp. Then I need to remember to update the page that contains the index of all the galleries—which is where I usually fail. I looked at a few other gallery programs, but none of them quite did what I wanted; that is, produce nice pages while minimizing the steps required to go from iPhoto to a working web site.

A recent server crash and upload gave me the opportunity to think about how I would do things in an ideal world. I would mark in iPhoto the albums that I wanted to upload, and a program would generate the appropriate HTML and images, use the iPhoto album comments as a description for each album and put together a front page, and then upload everything that needed to be uploaded to the server.

I soon found myself writing the piece of pseudocode in Figure 1. After that, writing the software wasn't very challenging at all. From my initial design, which was really only useful for my own site, I've developed a handy little iPhoto-to-web utility with a slightly more general application. Rather than take you through exactly how I built the new tool, which I've called "Tate," I'll look at some of the things that were nonobvious, and then at what I plan to do with Tate in the future.

Interacting with iPhoto

The first problem, and in fact the hardest part of writing Tate, was working out which albums to select, extracting the comments from them, and finding out which photos are in the album. I started by grabbing brian d foy's "iphoto" program from CPAN: This uses *Mac::Glue* to talk to iPhoto's scripting interface. I hacked around with the bits of it that I needed, to produce:

```
my $iPhoto = Mac::Glue->new( "iPhoto" );
my $albums = $iPhoto->prop( "albums" );
my @albums = map {
    $albums->obj(item => $_)
} 1..$albums->count;
```

Simon is a freelance programmer and author, whose titles include Beginning Perl (Wrox Press, 2000) and Extending and Embedding Perl (Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumping. Simon can be reached at simon-cozens.org.

```
for my $album (@albums) {
    my $name = $album->prop( "name" )->get;
    my $photos = $album->prop( "photos" );
    my $count = $photos->count;
    my @photos = map {
        $photos->obj(item => $_)
    } 1..$photos->count;
    # ...
}
```

This gives us an array of photo albums and allows us to get at the object representing each photo in the album. There are only two problems with this: The first is that we don't necessarily want to publish all the albums in the library; the second is that we also need to use the album's comment as a description and, annoyingly, that isn't accessible using the scripting interface.

In fact, after closely examining file modification times while changing album comments in iPhoto, I determined that the file that contains the comments is `Pictures/iPhoto Library/Library.iPhoto`. This is a binary file (an Objective C "typedstream" file) that I couldn't work out how to read. In desperation, I first found that each album's data contained the string `\223\231\223\204\232` at most once, and I found that the album's name was the first string of printable characters in the record.

But the position of the album's description was not obvious. I spent quite a while on it, but couldn't work out how to extract it reliably. So in the end, I used the "canary" technique. A canary is

```
Identify albums to be exported
For each album:
    Get comment and number of photos
    For each photo, grouped in three columns:
        Get paths to thumbnail and image
        Copy thumbnail and image to staging area
        Get size of image
        Scale image to max size
        Get image comment
        Write individual page for photo
    Write page for whole album
    Store information to be used on title page
        Name of album / Number of photos / Comment
        Thumbnail of first
    Write title page
    Upload the whole lot
```

Figure 1: Steps in the gallery upload process.

a known piece of data in an unknown location. So, as the user, I make sure that I add the string “Export:” to the comments of albums I wish to publish. This means that when I see “Export:” in the binary file, I can grab everything printable after that string and use that as the album comment; see Example 1.

Now we have a hash linking album names to descriptions; when we’re iterating through the albums using *Mac::Glue*, we now only need to look at albums that have an entry in this hash.

When we’re looking at individual photos, we can ask iPhoto for various properties, as in Example 2.

This is all the information we need from iPhoto; now that we know where the images and thumbnails live, we can work on them directly.

Resizing and Filtering

And what work on them we need to do! Generally, the original photos in iPhoto are extremely large, so we need to resize them for web use. The *Imager* module is what you need for this. *Imager* is an XS module that binds a consistent interface to all the appropriate graphics libraries on the system. So, if we have libjpeg installed, we can use it to read, scale, and write JPEG images—exactly what we need for our gallery.

Imager is very easy to use, with a nice, simple OO interface:

```
my $maxwidth = 800;

Imager->new
    ->open(file => $image, type => "jpeg")
    ->scale(xpixels => $maxwidth)
    ->write(file => $image, type=>"jpeg");
```

We can even add another step in there for thumbnails: When you’re scaling down an image dramatically, you lose some of the sharpness. *Imager* has a Photoshop-like “unsharp mask” filter (and many other filters besides) that we can apply to make those images look crisper:

```
Imager->new
    ->open(file => $image, type => "jpeg")
    ->scale(xpixels => $maxwidth)
    ->filter(type=>'unsharpmask',
           scale=>0.2, stddev=>1.5)
    ->write(file => $image, type=>"jpeg");
```

Imager is a handy Swiss Army Knife for throwing images around programmatically and it gives us just what we need for resizing and sharpening our gallery images.

Incidentally, we expect to be running Tate every time we’ve got some new photographs to go up on the Web, which means we don’t want to be resizing and filtering every single image every time. Instead, we keep the output photos around and just test to see if the photo is newer than the one we created last time:

```
if (!-e $album_dir."/".$image_name
    or (stat($image_path))[9] >
    (stat($album_dir."/".$image_name))[9]){
    copy($thumb_path, $thumb_dir."/".$thumb_name);
    copy($image_path, $album_dir."/".$image_name);
    # Do the Imager stuff
}
```

Templating and Providing Information

So now we know where to find our images and we have them in the form we want them, the next stage is to produce the HTML to go around them. There are three types of pages we want to generate: the individual pages for each picture, the page full of thumbnails for each gallery, and the title page, which is the list of all the galleries.

```
open LIBRARY, $lib or die $!;
{
    local $/ = "\231\231\223\204\232";
    while (my $data = <LIBRARY>) {
        while ($data =~ /([[:print:]]{3,}).*Export: ([[:print:]]+)/gms) {
            $comment{$1} = $2 ;
        }
    }
}
```

Example 1: Finding the iPhoto album description.

```
my $title = $photo->prop("title")->get;
my $comment = $photo->prop("comment")->get;
my $thumb_path = $photo->prop("thumbnail path")->get;
my $thumb_name = $photo->prop("thumbnail filename")->get;
my $image_name = $photo->prop("image filename")->get;
my $image_path = $photo->prop("image path")->get;
```

Example 2: Getting properties from iPhoto.

When I first wrote Tate, I had it create the pages by printing individual snippets of HTML. As so often happens, I ended up quickly becoming confused, fighting through a file written in two different languages, and trying to remember which bits of HTML belonged where. It was more out of frustration than a drive for customizability that lead me to template out the HTML generation to separate template files. But when I’d done this, I realized it made Tate much more useful; it means the end-user doesn’t have to put up with my crazy ideas of what a photo gallery should look like and can completely redesign it.

So, for instance, with the main index page, we build up an array containing useful data for each album:

```
for my $album (@albums) {
    # ...
    push @album_data, {
        name => $name,
        photos => $count,
        comment => $comment{$name},
        thumb_url => process_photos($name,
                                    $photos)
    };
}
```

And then send the whole thing off to be templated:

```
my $template = Template->new(
    {INCLUDE_PATH => TEMPLATES}
);
$template->process("album_index.tt",
    { albums => \@album_data },
    STAGING_AREA."/index.html")
or die $template->error;
```

The page for an individual gallery is similar, but for individual photos, we can do something a bit special.

Satisfying Geek Photographers

I’m trying to learn to take better pictures, and much like any other discipline, the only way to improve is to practice, to look at what went right and what went wrong, and to try to learn how to do it better next time. So when I’m browsing through some of my pictures, it’s helpful to know how the camera was set up when the photo was taken: the exposure settings, aperture setting, whether the camera was in shutter or aperture priority, the zoom, the flash, and so on. Thankfully, modern cameras embed all this information

```

use constant STAGING_AREA => "/Users/simon/Library Support/Tate/Staging/";
use constant DESTINATION => "/web/default/htdocs/photographer/";
use constant SERVER => "simon-cozens.org";

my $rs = File::RsyncP->new({
    rsyncCmd => "ssh ".SERVER." /usr/bin/rsync",
    rsyncArgs => [ "--perms", "--recursive" ]
});
$rs->remoteStart(0, DESTINATION);
$rs->go(STAGING_AREA);
$rs->serverClose;

```

Example 3: Incorporating File::RsyncP.

in the form of “EXIF tags” into the pictures that they take. *Image::Info*, a generic interface to all kinds of information stored in graphic file headers, can read these tags.

I was planning to use *Image::Info* anyway because it gives you the size of the image, from which you can create the appropriate height and width attributes for the IMG tags. To do this, we create an *Image::Info* object for the photo, and pass it in to the template:

```

my $info = image_info($image_path);

my $page_data = {
    album => $name,
    title => $title,
    comment => $comment,
    file => $image_name,
    info => $info
};

```

But then, I also read in the *Image::Info* documentation that it lets me get at the EXIF tags. So if I put something like this in the template:

```

Taken: [% info.DateTime %] <br>
Camera: [% info.Make %] [% info.Model %] <br>
Exposure: [%info.ExposureMode%],
          [% info.ExposureProgram %],
          f[% info.FNumber %],
          [%info.ExposureTime%]",
          ISO [%info.ISOSpeedRatings%],
          [%info.MeteringMode%] metering<br>
Flash: [% info.Flash%]<br>

```

I can get a handy source of information about how the shot was taken.

Distributing Templates

The version of Tate that I’ve released uses a little trick to ensure that you have the templates without making it complicated to install. I just wanted Tate to be a single file, which you download and run; having to actually install it the MakeMaker way and then install the templates somewhere globally so that they could be accessed by all users who wanted to use Tate just seemed like overkill.

So, after my frustration made me rip the HTML code out of the program, a different frustration made me finally put them back in, but this time, each template in its own Perl subroutine:

```

sub photo_page_template{ <<'EOF'
<html>
<head>
    ...
EOF
}

```

(I used this as a cheap-and-cheerful alternative to using *Inline::Files* to store multiple data files in the same program.)

When Tate starts, it checks to see if you have the templates installed. If you don’t, it creates `~/Application Support/Tate/Templates` and writes the three templates into it:

```

open OUT, ">".TEMPLATES."image.tt";
print OUT photo_page_template();
close OUT
# ...

```

The templates are still separate blocks of data from the program, so we still maintain separation of code and data. They still turn out as separate files in the filesystem so that you can customize them, but now we only have a single file for the user to download and run.

Uploading with rsync

Now we have a set of HTML files, photographs, and thumbnails arranged in a “staging area.” Each album is in its own directory, and the thumbnails in subdirectories off them. The nice thing about this arrangement is that it’s precisely the directory layout we’re going to want on the web server. All we need to do is get the whole file tree up to the server, and we’re done.

Of course, there’s a dumb way to do it: I can just transfer every file up to the server. But as I mentioned earlier, I expect Tate to be run reasonably often, every time I have new photography to go up on the Web. Most of the old galleries won’t change, so there’s no sense sending all the files up again—I only want to upload files that have changed.

The rsync protocol, by Andrew Tridgell and Paul Mackerras, is a good solution to this problem. It exchanges lists of file information to determine which files have been changed between a server and a client, and then sends the necessary files across. If two files differ, the server chunks the file up into pieces, and only sends across the pieces that differ. This means that it’s an efficient way of mirroring large sets of files that don’t change very much. In fact, it’s often used for mirroring CPAN, since using rsync means only the new modules get pushed out to the mirror sites, rather than everything, every night.

Best of all, there’s a Perl implementation of the rsync protocol (*File::RsyncP*), we can use to transfer files to the web server. The idea is that you use your normal way of connecting to the server—I’m going to log in via ssh—and then execute the rsync command. Then *File::RsyncP* will talk with the remote rsync and exchange files.

After all that explanation, making it work is actually stunningly simple; see Example 3.

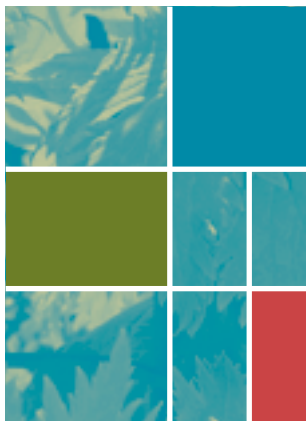
The “staging area” is where we’ve built our static tree of galleries; the destination is where we want to put it all on the remote filesystem.

Then we create a *File::RsyncP* object, telling it how to connect to the server and what to run, tell the object that it is sending to a particular destination, point it at the staging area, and tell it to go. Our beautiful photographs have made their way from the camera, to iPhoto, to a staging area on the local drive, and now finally onto the web server, for all the world to see!

Future Plans

Tate is available for download from <http://simon-cozens.org/programmer/tate.html>. At some point in the future, I plan to use it to learn to program the CamelBones toolkit; CamelBones is a binding to the OS X Cocoa libraries to create a native, GUI OS X application written in Perl. When I do, I shall write about it here!

TPJ



Making Web Images

brian d foy

I don't often create images for web sites, but when I do, I do the same things—put a black border around the image and make a smaller, thumbnail image. I can do all of this stuff in just about any photo-editing software, but that's annoying. Fortunately, Perl can easily do both of these for me. This is really handy when I want to add many images at the same time.

As with most things in Perl, there is more than one way to do this. One popular way, which I discuss here, uses the *Image::Magick* module (also known as *PerlMagick*), although that requires me to install ImageMagick (<http://www.imagemagick.org/>), which sometimes isn't the easiest thing to install. Aside from that and some left over ugliness from the original C interface, the *Image::Magick* module gets my job done.

To add a border to an image, I could just use the *convert* utility that comes with ImageMagick:

```
convert a.gif -bordercolor black -border 1x1 b.gif
```

This command opens *a.gif*, sets the border color to black, draws a one-pixel-high and one-pixel-wide border, and saves the result in *b.gif*. The order of the options on the command line matter: Start with the original image name, then the operations, and finally, the output image name.

I want my own program, though. Although I don't show it in this article, the actual program I use does a lot of other things to find the images and store them in the right place. That's just a simple matter of programming, so I skip it so I can focus on the image manipulations.

For all of the images I specify on the command line, I want to add a one-pixel-wide black border. I start off my program in the usual order by turning on strictures, then I pull in the *Image::Magick* module; see Listing 1.

In the *foreach()* block, I put each filename from the command line into *\$file* in turn. Inside the block, I create a new *Image::Magick* object. I get a file with the *Read()* method, and *Image::Magick* automatically figures out its type. It can handle any type that the underlying libMagick supports, which depends on which image libraries I configured when I compiled and installed it. Oddly, *Image::Magick* returns a false value when things succeed and true otherwise.

Once I read the image, I call my *add_border* subroutine. It's not much of a subroutine because *Image::Magick* makes things so easy. I set the border color to black with the *Set()* method. I

could make this configurable, but I've never wanted a color other than black; you can have any color you like. I can also set just about any parameter in the same way as long as I know the parameter name. After I've set the border color, I call the *Mogrify()* method, which applies a particular routine to the image. In this case, I add the border with a width of one pixel on each side. If I wanted different widths on the top/bottom and left/right dimensions, I just adjust my "1x1" value in the *Mogrify()* call.

That's only part of the problem, though. I also want to make thumbnails. I can do this directly with the *convert* command as I used before:

```
convert fullsized.jpg -resize '100x90' resized.jpg
```

I can also do this programmatically by stealing from my previous script. Instead of adding a border, I call *Mogrify()* to resize the image. In this example, I've hardcoded the final image size based on the files I get off of my digital camera. It's a simple matter of programming to choose a scaling factor and adjust the size, and I leave that up to you.

I add another twist to this program, though. At the start, I create a directory named "Thumbnails" (see Listing 2). I'll use that to store the smaller versions of the image and I'll use the same filename for the full-size and thumbnail versions. They'll just be in separate directories. Later on, at the end of the *foreach()*, I need to save the result to the right directory and filename. I use the *File::Spec* module's *catfile()* method to join the directory and filename according to the convention on the current operating system. Once I have a good path name, I use it as the destination filename.

Image::Magick has many other functions to programmatically process the image, too. Sadly, the *Image::Magick* documentation points to a web page (<http://www.imagemagick.org/www/perl.html>), so you are out of luck unless you have Internet access at the same time you want to use *Image::Magick*. Do what I did: Save the web page source locally.

I can still do better than my two previous programs, though. I usually do those two tasks at the same time, and I also want to put a border around my thumbnail images. I want to run one program and get everything done at the same time. After I add the border to the image, I make the thumbnail. This time, instead of assuming that I have certain dimensions for the image, I do a simple calculation in *scaled_dimensions()* to set the longest dimension to 80 pixels and adjust the other one appropriately (see Listing 3).

And, finally, I leave you with one more *ImageMagick* trick, even though it doesn't use Perl. You may have noticed that *Image::Magick* doesn't need me to tell it about image formats. The *Read()* method figures it out, and the *Write()* method figures it out. This makes for a cheap image format conversion utility simply by choosing the file extension:

brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com. Contact brian at comdog@panix.com.

```
convert foo.gif foo.png
```

In the Perl script, you do the same thing: Read in the image in any format that libMagick supports, then write it to another for-

mat by changing the file extension. I'll leave the Perl version of that last convert utility to you, though. Good luck.

TPJ

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

Listing 1

```
#!/usr/bin/perl
use strict;

use Image::Magick;

foreach my $file ( @ARGV )
{
    my $image = Image::Magick->new();

    if( $image->Read( $file ) )
    {
        warn "Could not read $file, skipping\n";
        next;
    }

    add_border( $image );

    if( $image->Write( $file ) )
    {
        warn "Could not write $file, skipping\n";
        next;
    }
}

#####
sub black_border
{
    my $image = shift;

    $image->Set( bordercolor => 'black' );
    $image->Mogrify( border => '1x1' );
}
```

Listing 2

```
#!/usr/bin/perl
use strict;

use File::Spec;
use Image::Magick;

my $dir = "Thumbnails";

unless( -d $dir or mkdir $dir, 0755 )
{
    die "$dir directory does not exist\n".
        "and I could not create it\n!\n";
}

foreach my $file ( @ARGV )
{
    my $image = Image::Magick->new();

    if( $image->Read( $file ) )
    {
        warn "Could not read $file, skipping\n";
        next;
    }

    $image->Mogrify( resize => '400x300' );

    my $output = File::Spec->catfile( $dir, $file );

    if( $image->Write( $output ) )
    {
        warn "Could not write $output, skipping\n";
        next;
    }
}
```

Listing 3

```
#!/usr/bin/perl
```

```
use strict;

use File::Spec;
use Image::Magick;

my $dir = "Thumbnails";

unless( -d $dir or mkdir $dir, 0755 )
{
    die "$dir directory does not exist\n".
        "and I could not create it\n!\n";
}

foreach my $file ( @ARGV )
{
    my $image = Image::Magick->new();

    if( $image->Read( $file ) )
    {
        warn "Could not read $file, skipping\n";
        next;
    }

    add_border( $image );

    if( $image->Write( $file ) )
    {
        warn "Could not write $file, skipping\n";
        next;
    }
}

thumbnail(
    $image, File::Spec->catfile( $dir, $file )
);

sub thumbnail
{
    my $image = shift;
    my $dest = shift;

    my $width = $image->Get( 'width' );
    my $height = $image->Get( 'height' );

    my $longer = $width > $height ? $width : $height;

    my $new_size = scaled_dimensions( $image );

    $image->Mogrify( resize => $new_size );

    return $image->Write( $dest ) ? 0 : 1;
}

sub scaled_dimensions
{
    my $image = shift;

    my $width = $image->Get( 'width' );
    my $height = $image->Get( 'height' );

    my $longer = $width > $height ? $width : $height;

    my $factor = $longer / 80;

    join "x", map { int( $_ / $factor ) }
        ( $width, $height );
}

sub add_border
{
    my $image = shift;

    $image->Set( bordercolor => 'black' );
    $image->Mogrify( border => '1x1' );
}
```

TPJ

Source Code Appendix

Julius C. Duque “Creating Self-Contained Perl Executables, Part I”

Listing 1

```
#!/usr/local/bin/perl

# Julius C. Duque

use strict;
use warnings;
use Getopt::Long;

my ($format, $help) = ();

GetOptions(
    "format=s" => \$format,    # =s -> takes mandatory string argument
    "help"     => \$help,      # optional switch
);

if (!$format or $help) {
    if ($^O eq "MSWin32") {
        $0 =~ s/.*\\//g;      # Windows
    } else {
        $0 =~ s/.*///g;
    }

    print "Usage: $0 --format=unix file1 [file2] [file3] ...\n";
    print "      $0 -f=unix file1 [file2] [file3] ...\n";
    print "\n";
    print "      $0 --format=dos file1 [file2] [file3] ...\n";
    print "      $0 -f=dos file1 [file2] [file3] ...\n";
    print "\nOriginal file(s) will be overwritten\n";
    exit 1;
}

foreach my $infile (@ARGV) {
    print "Converting $infile to $format format...\n";
    open INFILE, $infile;
    open OUTFILE, ">temp.$infile";
    binmode INFILE;
    binmode OUTFILE;
    while (<INFILE>) {
        $_ =~ s/\015\012/\012/g if ($format =~ /u/);
        $_ =~ s/\012/\015\012/g if ($format =~ /d/);
        print OUTFILE;
    }

    close INFILE;
    close OUTFILE;
    rename "temp.$infile", "$infile";
}
```

Listing 2

```
#!/usr/local/bin/perl

use strict;
use warnings;
use Tk;
use Tk::Balloon;

my $INDENT_DEF = 0;
my $LWIDTH_DEF = 70;
my $VERSION = "1.4.3";
my $TITLE = "Perl/Tk Example $VERSION";
my $AUTHOR = "Julius C. Duque";
my $indent = $INDENT_DEF;
my $newline = 1;
my $hyphenate = 1;
my $width = $LWIDTH_DEF;
my ($BOTH, $LEFT, $RIGHT, $CENTERED) = (1, 2, 3, 4);
my $format_choice = $BOTH;
my ($infile, $outfile) = ();

my $mw = new MainWindow();
drawButtons();
Tk::MainLoop();

sub processfile
{
```

```

open INFILE, $infile;
open OUTFILE, "> $outfile";
while (<INFILE>) {
    print OUTFILE;
}

close INFILE;
close OUTFILE;

printMessage("info", "OK", "File was successfully saved.");
}

sub drawButtons
{
    $mw->title($TITLE);

    # Status bar widget
    my $status = $mw->Label(-width => 70, -relief => "sunken",
        -anchor => "w")->pack(-side => "bottom", -padx => 1, -pady => 1,
        -fill => "x");

    # Create balloon widget
    my $b = $mw->Balloon(-statusbar => $status);

    # Create menu bar frame
    my $menubar = $mw->Frame(-borderwidth => 4, -relief => "ridge")->
        pack(-side => "top", -fill => "x");

    # Create Open File button
    my $openfilebutton = $menubar->
        Button(-text => "Open File", -relief => "raised", -width => 10,
            -command => [\&fileDialog, $mw, "open"])->pack(-side => "left");

    $b->attach($openfilebutton, -msg => "Open a file");

    # Create Save File button
    my $savefilebutton = $menubar->
        Button(-text => "Save To File", -relief => "raised", -width => 10,
            -command => sub {
                if (defined $infile and $infile ne "") {
                    fileDialog($mw, "save");
                } else {
                    printMessage("warning", "OK",
                        "You must open a file first");
                }
            })->pack(-side => "left");

    $b->attach($savefilebutton,
        -msg => "Proceed with saving a file");

    # Create About button
    my $aboutbutton = $menubar->Button(-text => "About",
        -relief => "raised", -width => 10,
        -command => [\&printMessage, "info", "OK",
            "A Perl/Tk script created by $AUTHOR"])->pack(-side => "left");

    $b->attach($aboutbutton,
        -msg => "$TITLE created by $AUTHOR");

    # Create Quit button
    my $quitbutton = $menubar->Button(-text => "Dismiss",
        -relief => "raised", -width => 10, -command => sub { exit })->
        pack(-side => "right");

    $b->attach($quitbutton, -msg => "Quit this Perl/Tk script");

    my $both = $mw->Radiobutton(-variable => \$format_choice,
        -value => $BOTH, -text => "Radio Button 1")->
        pack(-side => "top", -anchor => "w");

    $b->attach($both, -msg => "Radio Button 1");

    my $left = $mw->Radiobutton(-variable => \$format_choice,
        -value => $LEFT, -text => "Radio Button 2")->pack(-side => "top",
        -anchor => "w");

    $b->attach($left, -msg => "Radio Button 2");

    my $right = $mw->Radiobutton(-variable => \$format_choice,
        -value => $RIGHT, -text => "Radio Button 3")->pack(-side => "top",
        -anchor => "w");

    $b->attach($right, -msg => "Radio Button 3");

```

```

my $centered = $mw->Radiobutton(-variable => \$format_choice,
    -value => $CENTERED, -text => "Radio Button 4")->pack(-side => "top",
    -anchor => "w");

$b->attach($centered, -msg => "Radio Button 4");

$both->select;    # Set default to $both

my $chknewline = $mw->Checkbutton(-variable => \$newline,
    -text => "Check Box 1")->
    pack(-side => "top", -anchor => "w");

$b->attach($chknewline,
    -msg => "Check Box 2");

$chknewline->select;    # Set default to $newline

my $chkhyphen = $mw->Checkbutton(-variable => \$hyphenate,
    -text => "Check Box 2")->pack(-side => "top", -anchor => "w");

$b->attach($chkhyphen, -msg => "Check Button 2");

$chknewline->select;    # Set default to $hyphenate

my $f = $mw->Frame->pack(-side => "left");

my $l = $f->Label(-text => "Indentation: ", -justify => "left");

$b->attach($l, -msg => "Blah blah blah...");

Tk::grid($l, -row => 0, -column => 0);

my $tindent = $f->Entry(-width => 2, -textvariable => \$indent,
    -justify => "right");

$b->attach($tindent,
    -msg => "Number of spaces at the start of every paragraph");

Tk::grid($tindent, -row => 0, -column => 1);

$l = $f->Label(-text => "characters (default: $INDENT_DEF) ",
    -justify => "left");

$b->attach($l,
    -msg => "Number of spaces at the beginning of each paragraph");

Tk::grid($l, -row => 0, -column => 2);

$l = $f->Label(-text => "Line width: ", -justify => "left");
Tk::grid($l, -row => 1, -column => 0);
$b->attach($l, -msg => "Maximum length of every line");

my $tlwidth = $f->Entry(-width => 2, -textvariable => \$width,
    -justify => "right");

$b->attach($tlwidth, -msg => "Maximum length of every line");
Tk::grid($tlwidth, -row => 1, -column => 1);

$l = $f->Label(-text => "characters (default: $LWIDTH_DEF)",
    -justify => "left");

$b->attach($l, -msg => "Maximum length of every line");
Tk::grid($l, -row => 1, -column => 2);
}

sub printMessage
{
    my ($icon, $type, $outputmsg) = @_;
    my $msg = $mw->messageBox(-icon => $icon, -type => $type,
        -title => $TITLE, -message => $outputmsg);
}

sub fileDialog {
    my ($w, $operation) = @_;
    my @types = ([ "Text files", [qw/.txt .doc/]],
        [ "Text files", "", "TEXT"],
        [ "All files", "" ]);
};

if ($operation eq "open") {
    $infile = $w->getOpenFile(-filetypes => \@types);
}

if ($operation eq "save") {

```



```

$outfile = $w->getSaveFile(-filetypes => \@types,
    -initialfile => "Untitled",
    -defaultextension => ".txt");

processfile() if (defined $outfile and $outfile ne "");
}
}

```

Paul Barry “Monitoring Network Traffic Revisted”

Listing 1

```

#!/usr/bin/perl -w

use strict;

# Use the correct set of modules.
use Net::PcapUtils;
use NetPacket::Ethernet;
use NetPacket::IP;
use NetPacket::TCP;

use constant TRUE => 1;

use constant CAPTURE_FILTER => '(dst 127.0.0.1) && (tcp[13] & 2 != 0)';
use constant CAPTURE_DEVICE => 'eth0';

# Open the network card for capturing.
my $pkt_descriptor = Net::PcapUtils::open(
    FILTER => CAPTURE_FILTER,
    DEV    => CAPTURE_DEVICE
);

# Check that the card "opened" OK.
if ( !ref( $pkt_descriptor ) )
{
    print "Net::PcapUtils::open returned: $pkt_descriptor\n";
    exit;
}

while( TRUE )    # i.e., forever, or until "killed" ...
{
    # Capture a packet from the network card.
    my ( $packet, %header ) = Net::PcapUtils::next( $pkt_descriptor );

    # Process the captured packet.
    syn_packets( $packet );
}

sub syn_packets {
    my $packet = shift;

    # Strip Ethernet encapsulation of captured packet.
    my $ether_data = NetPacket::Ethernet::strip($packet);

    # Decode contents of TCP/IP packet contained within
    # captured Ethernet packet.
    my $ip = NetPacket::IP->decode($ether_data);
    my $tcp = NetPacket::TCP->decode($ip->{'data'});

    # Print all out where its coming from and where its
    # going to!
    print $ip->{'src_ip'}, ":", $tcp->{'src_port'}, " -> ",
        $ip->{'dest_ip'}, ":", $tcp->{'dest_port'}, "\n";
}

```

Listing 2

```

#!/usr/bin/perl -w

use Net::PcapUtils;
use NetPacket::Ethernet qw( :strip );
use NetPacket::IP qw( :strip );
use NetPacket::TCP;
use strict;

use constant TRUE => 1;

use constant CAPTURE_FILTER => '(tcp port 8080)';
use constant CAPTURE_DEVICE => 'eth0';
use constant CAPTURE_AMOUNT => 1500;

my $pkt_descriptor = Net::PcapUtils::open(
    FILTER => CAPTURE_FILTER,

```

```

        DEV      => CAPTURE_DEVICE,
        SNAPLEN => CAPTURE_AMOUNT
    );

if ( !ref( $pkt_descriptor ) )
{
    print "Net::PcapUtils::open returned:
          $pkt_descriptor\n";
    exit;
}

while( TRUE )
{
    my ( $packet, %header ) =
        Net::PcapUtils::next( $pkt_descriptor );

    process_packet( $packet );
}

sub process_packet {
    my $packet = shift;

    my $tcp = NetPacket::TCP->decode(
        ip_strip ( eth_strip ( $packet ) )
    );

    print $tcp->{ data };
}

```

brian d foy “Making Web Images”

Listing 1

```

#!/usr/bin/perl
use strict;

use Image::Magick;

foreach my $file ( @ARGV )
{
    my $image = Image::Magick->new();

    if( $image->Read( $file ) )
    {
        warn "Could not read $file, skipping\n";
        next;
    }

    add_border( $image );

    if( $image->Write( $file ) )
    {
        warn "Could not write $file, skipping\n";
        next;
    }
}

#####
sub black_border
{
    my $image = shift;

    $image->Set( bordercolor => 'black' );
    $image->Mogrify( border => '1x1' );
}

```

Listing 2

```

#!/usr/bin/perl
use strict;

use File::Spec;
use Image::Magick;

my $dir = "Thumbnails";

unless( -d $dir or mkdir $dir, 0755 )
{
    die "$dir directory does not exist\n".
        "and I could not create it\n$!\n";
}

foreach my $file ( @ARGV )

```

```

{
my $image = Image::Magick->new();

if( $image->Read( $file ) )
{
    warn "Could not read $file, skipping\n";
    next;
}

$image->Mogrify( resize => '400x300' );

my $output = File::Spec->catfile( $dir, $file );

if( $image->Write( $output ) )
{
    warn "Could not write $output, skipping\n";
    next;
}
}

```

Listing 3

```

#!/usr/bin/perl
use strict;

use File::Spec;
use Image::Magick;

my $dir = "Thumbnails";

unless( -d $dir or mkdir $dir, 0755 )
{
    die "$dir directory does not exist\n".
        "and I could not create it!\n!\n";
}

foreach my $file ( @ARGV )
{
    my $image = Image::Magick->new();

    if( $image->Read( $file ) )
    {
        warn "Could not read $file, skipping\n";
        next;
    }

    add_border( $image );

    if( $image->Write( $file ) )
    {
        warn "Could not write $file, skipping\n";
        next;
    }

    thumbnail(
        $image, File::Spec->catfile( $dir, $file )
    );
}

sub thumbnail
{
    my $image = shift;
    my $dest = shift;

    my $width = $image->Get( 'width' );
    my $height = $image->Get( 'height' );

    my $longer = $width > $height ? $width : $height;

    my $new_size = scaled_dimensions( $image );

    $image->Mogrify( resize => $new_size );

    return $image->Write( $dest ) ? 0 : 1;
}

sub scaled_dimensions
{
    my $image = shift;

    my $width = $image->Get( 'width' );
    my $height = $image->Get( 'height' );

    my $longer = $width > $height ? $width : $height;

```

```
my $factor = $longer / 80;

join "x", map { int( $_ / $factor ) }
  ( $width, $height );
}

sub add_border
{
  my $image = shift;

  $image->Set( bordercolor => 'black' );
  $image->Mogrify( border => '1x1' );
}
```

TPJ