CMP
United Business Media

November 2002

The Perl Journal

LETTER FROM THE EDITOR

Battling the Three P's

What you are reading right now is an experiment that worked. Just a few scant months ago, *The Perl Journal*'s future was dim, primarily due to the high cost of publishing's three P's—paper, printing, and postage. However, there is one thing that publishing's powers forget to consider at times—people, specifically the people who are committed to magazines like *TPJ*. I'm talking, of course, about you, folks—the people who use Perl day in and day out. You have made it clear that you value a publication that helps you see the simplicity, power, and (some would say) beauty of this language. Consequently, we decided to hitch our wagon not to paper and printing costs, but to what we saw as our best bet—reader enthusiasm. And it's working. The level of support we've seen for this all-electronic, reader-supported incarnation of *TPJ* has been both dramatic and gratifying.

What this all means is that we can get back to the business of writing about Perl. In this, my aspirations are similar to those of Jon Orwant when he launched *TPJ* back in 1996.

> "Perl is a pleasure to write about: It's quirky and useful and fun. We hope the same will be said of *TPJ*…We want *TPJ* to be intricate and interesting, like Perl itself."

Jon is right—Perl is a sometimes odd, but almost always efficient and economical language. We can do a lot worse at *TPJ* than to try to embody those qualities. (Well, some would say we've already got the "odd" part nailed down.)

So we hope you'll like what we've lined up for you in this and future issues: Moshe Bar rolls his own stress-test application that makes use of the Parallel Fork Manager; Robert Kiesling tells us when (and when not) to use perlcc to compile scripts into standalone executables; Sean Burke presents a very useful tool for converting HTML to an RSS feed; Columnists brian d foy and Simon Cozens discuss, respectively, the *Test::Pod* and *Attribute::Persistent* modules, and Jack Woehr reviews *Extending and Embedding Perl* by way of an interview with one of the authors. (By the way, you'll find the listings for these articles both at the end of each article and in the Source Code Appendix at the end of the issue, where the single-column layout makes for easy cut-and-paste.)

And don't forget—this is your magazine. If you've got a clever bit of Perl that you're itching to share with your fellow Perl programmers, how about writing an article about it? If you have any questions, suggestions, or recommendations for or about *TPJ*, just drop me a note at kcarlson@tpj.com.

[illegible]

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. SUBSCRIPTIONS: $12.00 for one year. Payment may be made via Mastercard or Visa; see http://www.tpj.com/. Entire contents (c) 2002 by CMP Media LLC, unless otherwise noted. All rights reserved.

CMP
United Business Media

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

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
NATIONAL SALES MANAGER
Trice Alford
(785) 838-7591
talford@cmp.com
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 COO
Steve Weitzner
EXECUTIVE VICE PRESIDENT AND CFO
John Day
CHIEF INFORMATION OFFICER
Mike Mikos
PRESIDENT, TECHNOLOGY SOLUTIONS GROUP
Robert Faletra
PRESIDENT, HEALTHCARE GROUP
Vicki Masseria
PRESIDENT, ELECTRONICS GROUP
Jeff Patterson
PRESIDENT, SPECIALIZED TECHNOLOGIES GROUP
Regina Starr Ridley
SENIOR VICE PRESIDENT, GLOBAL SALES AND MARKETING
Bill Howard
SENIOR VICE PRESIDENT, HUMAN RESOURCES AND COMMUNICATIONS
Leah Landro
PRESIDENT AND GENERAL COUNSEL
Sandra Grayson

Shannon Cochran

Perl News

Perl Quiz of the Week

Mark-Jason Dominus has launched a mailing list delivering a weekly Perl programming puzzle to subscribers. The quizzes come in two degrees of difficulty—"Normal" or "Expert." An example of an expert-level quiz:

> Write a subroutine, *subst*, which gets a string argument, *$s*. It should search *$s* and replace any occurrences of *$1* with the current value of *$1*, any occurrences of "*$2*" with the current value of *$2*, and so on. For example, if *$1*, *$2*, and *$3* happen to be "dogs," "fish," and "carrots," then *subst("$2, $1, and $3")* should return "fish, dogs, and carrots."

To subscribe to the quiz mailing list, send e-mail to perl-qotw-subscribe@plover.com. Past quizzes are temporarily archived at http://www.urth.org/~metaperl/domains/semantic-elements.com/perl/.

New Parrot Pumpking

Jeff Goff has stepped down as release manager and Keeper of the Keys and Source for Parrot, passing the title on to Steve Fink. In announcing the transfer, Dan Sugalski said, "Steve's been active with Parrot since near the beginning, and I have every confidence in his being able to keep the project moving forward."

Safe Module Patched

A security hole in the Safe module was discovered and reported to the perl.perl5.porters newsgroup by Andreas Jurenda. The problem lies with *Safe::reval()*, which executes given code in a safe compartment. "But," explained Jurenda, "this routine has a one-time safeness. If you call *reval()* a second time (or more) with the same compartment, you are potentially unsafe." Version 2.09 of the Safe module, available at http://search.cpan.org/author/ABERGMAN/Safe/, fixes the problem.

The Perl Foundation Grant Wrap-Up

The 2002 recipients of Perl Foundation funding—Damian Conway, Dan Sugalski, and Larry Wall—have posted accounts of their accomplishments during the grant period (which ended in July) at http://www.perlfoundation.org/index.cgi?page=accom. Conway, who supplemented his grant period with a final speaking tour, posted a final blog entry at the beginning of October calling his grant tenure "the toughest year-and-a-half I've ever experienced, but also the most rewarding." During that time, Conway accepted 56 speaking engagements across four continents. He also wrote 21 new modules, 88 "node" entries on the Perl Monks site, four Exegeses, 192 newsgroup messages, and more than 5000 e-mails in response to Perl questions.

In conclusion, Conway announced that he's on vacation for the rest of 2002—no mailing lists, no module maintenance, no lectures, no Perl-specific e-mail. He will, however, continue to work on the design of Perl 6.

The Perl Foundation solicits contributions to the grant fund at http://www.perlfoundation.org/index.cgi?page=contrib.

"Spoofathon" Fundraiser Opens

Another way to raise money for the Perl Foundation is by entering Nicholas Clark's Perl Advocacy Spoofathon, which offers six donations of 100 guineas each (a total of about $160 US) to be paid to the Perl Foundation in honor of the best spoof essays submitted.

"There have been several badly researched *$foo* is better than Perl articles in the recent past, that have irritated myself and many other people in the perl community," explains Clark on Spoofathon (http://www.perl.org/advocacy/spoofathon/why.html). "I can't stop third parties writing these articles. But I do hope I can start to make people treat them with the seriousness that they deserve."

There is no fixed essay deadline; the judges, Michael G. Schwern and Greg McCarroll, will announce the awards at their discretion. Entries can be submitted to spoofathon@perl.org.

Komodo 2.0 Released

ActiveState has released Version 2.0 of its Komodo IDE, optimized for Perl and other open-source languages. New features include automatic generation of web services proxies; GUI dialogs and a Tk-based dialog builder; and (for team development) source-code control integration and a Project Manager utility. The editor includes tools for code reuse, and the syntax highlighting and code commenting functions have been updated. Komodo 2.0 for Windows is available now, while a Linux version is available in beta release. For more details, see http://www.activestate.com/Komodo/start92.html.

New Perl Books

Several new Perl books have been released in the past month. Tim Jenness and Simon Cozens coauthored *Extending and Embedding Perl* (Manning Publications, 2002), dealing with the use of Perl in C programs. (*Extending and Embedding Perl* is reviewed in this issue.)

Programming Perl in the .NET Environment, by Yevgeny Menaker, Michael Saltzman, and Robert Oberg (Prentice Hall, 2002) is designed for use as a textbook, beginning with a tutorial on the Perl language and ending with enterprise application integration.

Finally, *Embedding Perl in HTML with Mason*, by Dave Rolsky and Ken Williams (O'Reilly & Associates, 2002), covers the *HTML::Mason* framework. Rolsky and Williams describe Mason as "a powerful text templating tool for embedding Perl in text." For more information, see http://www.masonbook.com/.

Moshe Bar

Using Perl to Stress-Test Server Applications

It is one of the ironies of a programmer's life that when we've finished writing a program, the real work has just begun. Once the program compiles cleanly and seems to do what it was supposed to do, good programmers then need to find out how fast (or slow) the program performs its designed purpose. In other words, we need to stress-test the program.

There are commercial stress-test applications in the market, but they are expensive for individuals or small development firms, and often too generalized—capable of most everything, but not really optimized for your particular situation. With its myriad modules (I call them "connectors"), Perl is a great tool to help you stress-test your particular program.

As a senior contributing editor at BYTE.com, I've been able to compare operating systems to one another (usually Linux against something else) over the last few years, and the way I do it is by running the exact same stress-test against standard applications like web servers and databases (MySQL). As it turns out, Perl is my tool of choice for this purpose, thanks in part to the CPAN repository of modules.

Parallel Fork Manager

One of the modules that best suits the stress-test purpose is the Parallel Fork Manager (http://aspn.activestate.com/ASPN/CodeDoc/Parallel-ForkManager/ForkManager.html). Parallelizable parts of a Perl program can be forked into separate processes that continue to run concurrently with the parent process in a multiCPU environment. In a massively parallel environment, such as my own openMosix technology (http://www.openMosix.org/), every forked Perl program always finds a CPU to run on. The total run time is therefore drastically reduced. The Perl program in Listing 1 can run from many nodes in parallel in an openMosix cluster, and the server being tested is submitted to a veritable bombardment of client-side requests.

Using the Perl Fork Manager is easy. A simple construct like Example 1 parallelizes very nicely—and by using openMosix you don't even have to care about distribution in the cluster. Several callbacks (or event handlers) are possible, which are called on certain events. Among the callbacks, you typically use *run_on_start $code*, which defines a subroutine called when a freshly forked child begins execution.

Looking at Listing 1, all the script does is run *$processes* in parallel and create tables with *$rows*, then starts *$transactions* (SQL queries) against the database. In the beginning, you define the various fork and database connection managers. Then, you define an array of names that will be used later to populate the database with sample records. If it is important to increase the cardinality of the indices, then you can do so by building an array of *n* names by reading in *n* words from the UNIX dict file (which in Red Hat is in /usr/share/dict/linux.words).

Using the well-known DBI module, you then connect to the database (Oracle in this example) and first drop, then create the tables of this fictitious application. In Part Four of the code (see comments), you tell the openMosix clustering system (through the convenient /proc filesystem of Linux) to make sure all forked Perl programs are free to migrate to other nodes in the cluster to evenly distribute them and keep the cluster load-balanced. Part Seven is where you parallelize the SQL queries to the database using the Fork Manager module.

```
use LWP::Simple;
  use Parallel::ForkManager;
 ...

# Max 30 processes because we have 30 nodes in the cluster
my $pm = new Parallel::ForkManager(30);

foreach my $number_of_nodes {
   $pm->start and next; # do the fork - the child process skips this
           loop
   # execute your server query here
   $pm->finish; # do the exit in the child process
}
$pm->wait_all_children;
```

Example 1: A simple construct for parallelizing.

Moshe is a systems administrator and operating-system researcher and has an M.S. and a Ph.D. in computer science. He can be contacted at moshe@moelabs.com.

That's it. The entire program consists mostly of the SQL handling for connection to the database and housekeeping. The code related to the Fork Manager is just a few lines. The same program structure can be used for stress-testing any server application. Depending on what you need to do with the results, you can use various Perl modules to represent the results graphically or publish them onto a web page.

For my purposes, a simple run-time summary is enough and it keeps the program simple. Everybody is encouraged to use this Perl script. I'd be interested to know how you enhanced it for your particular needs.

TPJ

Listing 1

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

# Part One Modules and parameters
use Parallel::ForkManager;
use DBI;
srand;

############# configruation part #######################
my $processes=30;
my $transactions=5000;
my $rows=100000;  # rows to be added to the database
#######################################################

my $p;

# Part Two - Seed for DB population
@NAMES = qw/Alf Ben Benny Daniel David Foo Bar Moshe
                                          Avivit Jon Linus Larry Safety
First/;

#Part Three - DB environment
my $username;
my $money;
my $id;
my $exists=0;
my $commandlineargs=0;

$SID = $ENV{ "ORACLE_SID" };

if (!$SID) {

printf("No ORACLE_SID environment!\n");
printf("... do not know to which database you want to connect\n");
printf("You have to export the database-name in the environment\n");
printf("variable ORACLE_SID e.g.\n");
printf("export ORACLE_SID [mydb]\n");

exit -1
}

# Part Four - openMosix related. unlocks this process and all its children
sub unlock {

#open (OUTFILE,">/proc/self/lock") ||
#ie "Could not unlock myself!\n";
#print OUTFILE "0";

}

unlock;

# Part Five - Here we connect to the DB and populate the tables with
records
sub filldb {

my $dbh = DBI->connect( "dbi:Oracle:$SID",
                         "scott",
                         "tiger",
                         {
                           RaiseError => 1,
                           AutoCommit => 0
                         }
                       ) || die "Database connection not made:
$DBI::errstr";
                     for ($loop=0; $loop<$rows; $loop++) {
                             # prepare the random values
                             srand;
                             $p=rand();
                             $username = $NAMES[rand(@NAMES)];
                             $money=rand()*(rand()*100);
                             $id=rand()*1000;
                             $id=sprintf("%d", $id);
                             # insert
                              my $sql2 = qq{ insert into stress
                                            values
($id,'$username',$money) };
                             my $sth2 = $dbh->prepare( $sql2 );

                             $sth2->execute();
                             $sth2->finish();
            }
                    $dbh->disconnect();
}
# delete old entries from db
sub deldb {
my $dbh = DBI->connect( "dbi:Oracle:$SID",
                         "scott",
                         "tiger",
                         {
                           RaiseError => 1,
                           AutoCommit => 0
                         }
                       ) || die "Database connection not made:
                                $DBI::errstr";

                         # drop
                         my $sql1 = qq{ select TABLE_NAME from user_tables
                                                   where
                                                        TABLE_NAME='STRESS'
                                                        };
                         my $sth1 = $dbh->prepare( $sql1 );
                         $sth1->execute();

             # if the table does not exists
                         while( $sth1->fetch() ) {
              $exists=1;
                           }
             if (!$exists) {

              # create
                          my $sql2 = qq{ create table STRESS ( id number,
                                           name varchar2(255), money float)
                                  };
                          my $sth2 = $dbh->prepare( $sql2 );

                          $sth2->execute();
                          $sth2->finish();

              } else {

                          # cleanup
                          my $sql3 = qq{ delete from STRESS };
                          my $sth3 = $dbh->prepare( $sql3 );

                          $sth3->execute();
                          $sth3->finish();
             }
             $sth1->finish();
                         $dbh->disconnect();
}

# Part Six - Stress-testing the DB with parallel instances

sub stressdb {
my $dbh = DBI->connect( "dbi:Oracle:$SID",
                         "scott",
                         "tiger",
                         {
                           RaiseError => 1,
                           AutoCommit => 0
                         }
                       ) || die "Database connection not made:
$DBI::errstr";
        for ($loop=0; $loop<$transactions; $loop++) {
             # prepare the random values
             srand;
             $p=rand();
             $username = $NAMES[rand(@NAMES)];
             $money=rand()*(rand()*100);

             $id=rand()*1000;
             $id=sprintf("%d", $id);

             # update transactioin
            if ($p<0.3) {

             printf("update\n");

             my $sql = qq{ update stress set name='$username',
                                                money=$money where id=$id
                                                      };
             my $sth = $dbh->prepare( $sql );

             $sth->execute();
             $sth->finish();
            }

            # select
            if (($p>0.3) and ($p<0.7)) {
             printf("select\n");
             $sql = qq{ select id, name, money from stress };
             my $sth = $dbh->prepare( $sql );

             $sth->execute();
             my( $sid, $sname, $smoney );

             $sth->bind_columns( undef, \$sid, \$sname, \$smoney );
             while( $sth->fetch() ) {

               # print "$sid $sname $smoney\n";
             }
            }

            # delete + insert
            if ($p>0.7) {
             printf("delete/insert\n");

             # delete first
                         my $sql1 = qq{ delete from stress where id=$id };
                         my $sth1 = $dbh->prepare( $sql1 );

                         $sth1->execute();
             $sth1->finish();

            # insert
                         my $sql2 = qq{ insert into stress
                                         values ($id,'$username',$money)
};
                         my $sth2 = $dbh->prepare( $sql2 );

                         $sth2->execute();
                         $sth2->finish();

            }
        }
        $dbh->disconnect();
}

############## main ######################

print "\nStarting the orastress test\n\n";

# check for commandline arguments
foreach $parm (@ARGV) {
 $commandlineargs++;
}
$parm=0;
if ($commandlineargs!=3) {

 # get values from the user
 print "How many rows the database-table should have ? : ";

 $rows=<STDIN>;
 chomp($rows);

 while ($rows !~ /\d{1,10}/){
  print "Invalidate input! Please try again.\n";
  print "How many rows the database-table should have ? : ";
  $rows = <STDIN>;
  chomp($rows);
 }

 print "How many clients do you want to simulate ? : ";
 $processes=<STDIN>;
 chomp($processes);
 while ($processes !~ /\d{1,10}/){
  print "Invalidate input! Please try again.\n";
  print "How many clients do you want to simulate ? : ";

  $processes = <STDIN>;
  chomp($processes);
 }
 print "How many transactions per client do you want to simulate ? : ";
 $transactions=<STDIN>;
 chomp($transactions);

 while ($transactions !~ /\d{1,10}/){
  print "Invalidate input! Please try again.\n";
  print "How many transactions per client do you want to simulate ? : ";

  $transactions = <STDIN>;
  chomp($transactions);
 }

 } else {
 # parse the values from the command line

 $rows = $ARGV[0];
 $processes = $ARGV[1];
 $transactions = $ARGV[2];

 if (($rows !~ /\d{1,10}/) or ($processes !~ /\d{1,10}/) or
                                             ($transactions !~
/\d{1,10}/)) {
  print ("Invalidate input! Please try again.\n");
  print ("e.g.   stress-oracle.pl 10 10 10\n");

  exit -1;

  } else {
  print("got the values for the stress test from the command line\n");
  print("rows = $rows\n");

  print("processes = $processes\n");
  print("transactions = $transactions\n");

 }
}

# cleaning up old db
print("delete old entries from db\n");

deldb();

print("fill db with $rows rows\n");

filldb();

# Part Seven
print("starting $processes processes\n");
my $pm = new Parallel::ForkManager($processes);

$pm->run_on_start(
  sub { my ($pid,$ident)=@_;
  print "started, pid: $pid\n";
  }
);

for ($forks=0; $forks<$processes; $forks++){
  $pm->start and next;
  stressdb;

  $pm->finish;
}

$pm->wait_all_children;
$pm->finish;

print "Simulation finished\n";
```

TPJ

perlcc & Compiling Perl Script

Robert Kiesling

Perl has always had a compiler, as the perlcompile manual page provided with Version 5.8.0 points out, but its normal use is only to generate the internal bytecode that is run by the interpreter.

All recent versions of Perl, however, have had the ability to generate C source code or run a compiler to produce a standalone executable program. The library modules B.pm, C.pm, and CC.pm provide an interface to Perl's internal bytecode compiler. The perlcc script processes command-line options and runs all of the components of the compilation process. A listing of command-line options that perlcc accepts is given in Figure 1.

In earlier versions of Perl, compiler support for extension modules was inconsistent, reducing the usefulness of the compiler. There are still some language features that don't compile correctly, like unsigned math operations. But scripts that use complex extensions like Perl/Tk are much easier to turn into standalone programs using Perl 5.8.0.

There has been much reworking of the Perl internals—improvements in the IO abstraction layer, new interpreter threads, and code for a new version of MacPerl. This redesign effort has also made Perl scripts easier to compile.

The compiler is still experimental, though, and some of its modules, especially those that optimize code, are not yet fully integrated with the rest of the Perl libraries. It's likely that you'll need to experiment to find the best way to turn scripts into standalone programs.

Why Not to Compile Perl

Perl remains a primarily interpreted language. Some of the language's greatest advantages, such as its ability to bind variables very late in the interpreter process and its lack of strong data typing, make the language difficult to compile and add a lot of extra code to executables. Standalone programs don't run noticeably faster, and Perl gurus say that the executables compiled from scripts aren't any more secure than the interpreted versions.

However, standalone programs compiled from Perl do not need the interpreter or libraries to run. Also, compiling Perl scripts with statically linked extensions built into the interpreter might be a better option for producing specific applications. This could be the best reason to provide compiled programs, especially those that rely on libraries that users must install themselves.

Robert is the former maintainer of the Linux Frequently Asked Questions with Answers Usenet FAQ. He can be reached at rkiesling@ mainmatter.com.

The compiler prefers scripts to have the extension .p, which can be confusing if you also write Pascal programs. In fact, the compiler's design and operation is reminiscent of a Pascal p-code compiler, without the necessity of an extra run-time module.

Executables compiled from intermediate C code are huge as well, when statically linked against the Perl libraries. A minimal "Hello, world!" script like the hello.p script shown in Listing 1 results in a binary that is over 1 MB in size. Scripts that use the Perl/Tk GUI result in binaries of at least 5 MB. When generating C code, the compiler does not translate a Perl script into corresponding C source code. Instead, it generates a C representation of the internal bytecode, including the code for all the library modules that the script uses.

However, generating a C language source file and then using the compiler's optimization can reduce executable size by about 10 percent. Producing stripped binaries, which do not have symbol table information, can reduce the size of an executable by at least another 200 KB.

Building Perl with a shared version of its libraries can also reduce the size of executables, but then you need to include the shared libraries with the program for systems that don't have it already. You must also take care that the version of the library on the target system is the same as the version on the system the script was compiled on.

Table 1 shows the file sizes for three Perl scripts: the hello.p script mentioned earlier, touch.p, which mimics the UNIX touch

```
-L <directory>        # Search <directory> for run-time libraries.
-I <directory>        # Search <directory> for C include files.
-o <name>             # Generate output as file <name> instead of
                          a.out.
-v <level>            # Print verbose status messages while
                          compiling.
-e <script>           # Compile a one-line script.
-r                    # Run the executable after compiling it.
-B                    # Compile into byte code.
-O                    # Use optimized C backend.
-c                    # Generate an linker file (.o).
-h                    # Print help message.
-S                    # Dump C files.
-T                    # Run Perl backend using -T.
-t                    # Run Perl backend using -t.
-static               # Enable -static option.
-shared               # Create a shared library.
-log <filename>       # Save compilation messages in file.
-Wb <options>         # Pass options to the backend
-testsuite            # Optimize for Perl test suite.
```

Figure 1: Perlcc command-line options.

	hello.p	touch.p	textedit.p
Script Size	48	414	43
perlcc Generated C Source File Size	35,408	2,153,018	5,909,243
Generated perlcc Executable Size	1,139,110	2,429,237	4,631,649
GCC Compiled Executable Size	1,016,911	189,468	200,451
Gcc Compiled Size with -O Optimization	1,005,805	1,803,452	

Table 1: File sizes for hello.p, touch.p, and textedit.p.

utility (Listing 2), and textedit.p, a Perl/Tk script for a simple text editor (Listing 3).

Table 1 also lists the sizes of statically linked executables compiled with perlcc, the size of the generated C source files, and the sizes of executables generated by GCC with and without optimization.

The smallest executables are those generated using "gcc -O." Higher levels of optimization actually increase executable size slightly and result in much longer compilation times.

All of the scripts were compiled on a SPARCstation 20 with Solaris 8 and GCC 3.0.3. Other hardware platforms and operating systems produce similar results.

Compiling with perlcc and GCC

The perlcc command-line arguments are similar to those of a C compiler. The following command, for example, generates the executable textedit from the script textedit.p.

```
# perlcc textedit.p -o textedit
```

To generate the C source file textedit.p.c, from the textedit.p script, use the -S command-line argument.

```
# perlcc -S textedit.p
```

The libperl.a static library and the Perl include files are located in directories where Perl, not GCC, can find them. In addition to these directories, you must also specify which system libraries to link with the program. Example 1 shows the options that you must provide to GCC to compile and link the textedit.p script.

```
# gcc -O2 -I/usr/local/lib/perl5/5.8.0/sun4-solaris/CORE textedit.c \
> -L/usr/local/lib/perl5/5.8.0/sun4-solaris/CORE \
> /usr/local/lib/perl5/5.8.0/sun4-solaris/auto/DynaLoader/DynaLoader.a
\
> -lperl -lsocket -lm -o textedit
```

Example 1: GCC command-line options for compiling a Perl script.

When Static Linking is Better

All compiled scripts also require linking to DynaLoader.a library, whether or not the script loads additional Perl libraries. However, the dynamic module loader is so completely integrated into Perl that the interpreter cannot function without it.

If the size of compiled programs is critical or you want to use Perl in embedded environments (although the language designers recommend against it), you should consider building the Perl interpreter with statically linked versions of the libraries you want to include.

```
# cd perl-5.8.0/ext
# gunzip -c Tk800.023.tar.gz | tar xvf -
# mv Tk800.023 Tk
# cd ..
# ./Configure

...

Do you wish to use dynamic loading? [y] n

...

What extensions do you wish to include?
[B ByteLoader ... ] Tk
```

Example 2: Statically building the Perl/Tk libraries in the Perl interpreter.

Example 2 shows how to build Perl with the Tk library modules statically linked. You must first unpack the Perl/Tk source code in the Perl source tree. When you run the configure script, answer "n" to the dynamic loading option and specify which extensions you want to include.

In practice, building a statically linked interpreter requires that you specify all of the extensions you'll need. You'll need to enter all of them when configuring the Perl interpeter.

Configuring Perl in this way allows you to build application-specific executables, but you'll need to experiment to determine the best procedure for building the executable programs and which extensions to include in the interpreter.

Conclusion

Although Perl is primarily an interpreted language, it provides the tools and configuration options to produce compiled standalone programs that do not need the Perl interpreter or libraries to be present.

TPJ

Listing 1

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

print "Hello, world!\n";
```

Listing 2

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

use IO::File;

my $filename = $ARGV[0];

if (! length ($filename)) {
    print "Usage: touch <filename>\n";
    exit 1;
}

if (-f $filename) {  # Update access and modtimes on $filename if
                     # it already exists.
    my $timenow = time;
    utime $timenow, $timenow, $filename;
} else { # create the file
    my $fh = new IO::File $filename , O_CREAT;
    undef $fh;
}
```

Listing 3

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

use Tk;

my $mw = new MainWindow;

my $textwidget = $mw -> Text
    -> pack (-expand => 1, -fill => 'both');

MainLoop;
```

TPJ

Turning HTML into an RSS Feed

Sean M. Burke

In the September 2002 issue of *The Perl Journal*, Derek Valada's article "Parsing RSS Files with XML::RSS" sang the praises of RSS feeds, and showed how even if you don't have any RSS client programs or don't use a web site that aggregates them for you, it takes just a bit of Perl to write your own little utility for viewing the RSS content in your web browser. I can testify that once you get used to having RSS feeds from a few sites, you want all your favorite sites to have them. This article is about what to do when a site doesn't have an RSS feed: Make one for yourself by writing a little Perl tool to get content from the site's HTML.

Sites Without RSS Feeds

Some wonderful web sites provide RSS feeds that make it easy for us to find out when there's something interesting at their site, without actually having to go to that web site first. But some web sites just haven't gotten around to providing an RSS feed. Sometimes this is just because the site's programmers (if there are any!) just happen not to have heard about RSS yet. Or sometimes it's just that the people maintaining the site don't know that so many people actually use RSS and would appreciate an RSS feed—the main sysadmin of one of the Internet's larger web-logging sites recently told me, "I've considered it a bit but haven't found any real compelling reason yet, and no one else has seemed very interested in it"

Hopefully, all the larger sites will come around to providing RSS feeds; but I bet there will always be routinely updated content on the Web that lacks them. In that case, we have to make our own.

The first step in making an RSS feed for a remote site is checking that they don't already have one. Unlike a */robots.txt* file, the RSS feed for a site doesn't have a predictable name, nor is there even any one place on a site where it's customary to mention the URL of the RSS feed. Some sites mention the URL in their FAQ, and recently some sites have started mentioning the URL in an HTML *link* element in the site's HTML, like so:

```
<link rel="alternate" type="application/rss+xml"
   href="http://that_rss_url" >
```

If you can't find an RSS feed that way, search Google for "sitename RSS" or "sitename RDF"—you'd be surprised how effective that is. And if that doesn't get you anywhere, e-mail the site's webmaster and ask for the URL of their RSS. If they get enough such messages, they'll make a point of more clearly stating the RSS feed's URL if they have one, or setting one up if they don't.

But if absolutely none of these things work out, then it's time to roll up your sleeves and write an RSS generator that extracts content from the site's HTML.

Sean is a long-time contributor to CPAN, and is the author of Perl & LWP *from O'Reilly & Associates. He can be contacted at sburke@cpan.org.*

Scanning HTML

Processing HTML to find the bits of content that you want is one of the black arts of modern programming. Not only is each web page different, but as its content varies from day to day, it may exhibit unexpected changes in its template, which your program is hopefully robust enough to deal with. In fact, most of my book *Perl & LWP* is an explanation of the bag of tricks that you should learn to use for writing really robust HTML-scanner programs. You also need a bit of practice, but if you don't have any experience at scanning HTML, then doing so to write an RSS is a great place to start. In this article, I'll stick to just using regular expressions instead of more advanced approaches involving HTML::TokeParser or HTML::TreeBuilder.

The basic approach is this:

```
use LWP::Simple;
my $content_url =
     'http://whatever.url.to/get/from.html';
my $content = get($content_url);
die "Can't get $content_url" unless defined $content;
...then extract things from $content...
```

So, for example, consider http://freshair.npr.org/, the web site for National Public Radio's interview program *Fresh Air*. One page on the site has the listings for the current program, with HTML like this:

```
<A HREF="http://www.npr.org/ramfiles/fa/
  20020920.fa.01.ram">Listen to
   <FONT FACE="Verdana, Charcoal, Sans Serif"
      COLOR="#ffffff" SIZE="3">
   <B> John Lasseter            </B>
</FONT></A>
 ...
<A HREF="http://www.npr.org/ramfiles/fa/20020920
  .fa.02.ram">Listen to
   <FONT FACE="Verdana, Charcoal, Sans Serif"
```

```
      COLOR="#ffffff" SIZE="3">
    <B> Singer and guitarist Jon Langford    </B>
</FONT></A>
 ... plus any other segments ...
```

The parts that we want to extract are:

```
http://www.npr.org/ramfiles/fa/20020920.fa.01.ram
John Lasseter

http://www.npr.org/ramfiles/fa/20020920.fa.02.ram
Singer and guitarist Jon Langford
```

We can get the page and match the content with Listing 1, whose regular expression we arrive at through a bit of trial-and-error. When run, this happily produces the following output, showing that it's properly matching the three segments in that page (at time of writing):

```
url: {http://www.npr.org/ramfiles/fa/20020920
      .fa.01.ram}
title: {John Lasseter}

url: {http://www.npr.org/ramfiles/fa/20020920
      .fa.02.ram}
title: {Singer and guitarist Jon Langford}

url: {http://www.npr.org/ramfiles/fa/20020920
      .fa.03.ram}
title: {Film critic David Edelstein}
```

We can later comment out that print statement and add some code to write @items to an RSS file.

Now consider this similar case, where we're scanning the HTML in the *Guardian*'s web page for breaking news:

```
...
 <A HREF="/worldlatest/story/0,1280,-2035841,00.html">
 UnsolvedCrimes Vex Afghanistan</A><BR><B>6:50 am</B>
 <P><A HREF="/worldlatest/story/0,1280,-2035838,00
 .html">Christians Show Support For Israel</A><BR><B>
 6:40am</B><P><A HREF="/worldlatest/story/0,1280,-203
 5794,00.html">Schroe der's Party Wins 2nd Term</A>
 <BR><B>5:30 am</B><P>
...
```

It's a great big bunch of unbroken HTML (which I've put newlines into just for readability), but look at it a bit and you'll see that each item in it reads like this:

```
<A HREF="url">headline</A><BR><B>time</B><P>
```

You'll also note that items follow each other, one after another, with no intervening "</p><p>" tags or newlines.

So we cook up that pattern into a regular expression and put it into our aforementioned code template, as shown in Listing 2. When we run that, the code correctly produces this list of items:

```
url: {/worldlatest/story/0,1280,-2035841,00.html}
title: {Unsolved Crimes Vex Afghanistan}

url: {/worldlatest/story/0,1280,-2035838,00.html}

title: {Christians Show Support For Israel}

url: {/worldlatest/story/0,1280,-2035794,00.html}
title: {Schroeder's Party Wins 2nd Term}

 ...and a dozen more items...
```

We're ready to make both of these programs write their *@items* to an RSS feed—except for one thing: URLs in an RSS feed should really be absolute (starting with "http://…"), and not relative URLs like the "/worldlatest/story/0,1280,-2035794,00.html" we got from the Guardian page. Luckily, the URI.pm class provides a simple way to turn a relative URL to an absolute one, given a base URL:

```
URI->new_abs($rel_url => $base_url)->as_string
```

We can use this by adding a *use URI;* to the start of our program, and changing the end of our *while* loop to read like so:

```
$url = URI->new_abs($url => $content_url)->as_string;
    print "url: {$url}\ntitle: {$title}\n\n";
    push @items, $title, $url;
  }
```

With that change made, our program emits absolute URLs, such as these:

```
url: {http://www.guardian.co.uk/worldlatest/
   story/0,1280,-2035841,00.html}
title: {Unsolved Crimes Vex Afghanistan}

url: {http://www.guardian.co.uk/worldlatest/
   story/0,1280,-2035838,00.html}
title: {Christians Show Support For Israel}

url: {http://www.guardian.co.uk/worldlatest/
   story/0,1280,-2035794,00.html}
title: {Schroeder's Party Wins 2nd Term}
```

```
...and a dozen more items...
```

Basic RSS Syntax

An RSS file is a kind of XML file that expresses some data about the site in general, and then lists the details of each story item at that feed. While RSS actually has many more features than I'll discuss here (especially in later versions than the 0.91 version here), a minimal RSS file starts with an XML header, an appropriate doctype, and some metadata elements, like this:

```
<?xml version="1.0"?>
<!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD
      RSS 0.91//EN"
  "http://my.netscape.com/publish/formats/rss-
      0.91.dtd">
<rss version="0.91"><channel>
  <title> title of the site </title>
  <description> description of the site
      </description>
  <link> URL of the site </link>
  <language> the RFC 3166 language tag for this
      feed's content </language>
```

Then there are a number of *item* elements like this:

```
<item><title>...headline...</title><link>...url..
      </link></item>
```

And then the document ends like this:

```
</channel></rss>
```

So the RSS file that we would produce with our *Fresh Air* HTML scanner would look like Figure 1 (shown with with a bit of helpful indenting).

We can break Figure 1 down to three main pieces of code: one for all the stuff before the first item, one for taking our *@items* and spitting out the series of <item>…</item> elements, and then one to complete the XML by outputting </channel></rss>. But first there's one consideration—we can't really just take the code we pulled out of the HTML and dump it into XML. The reason for this is that XML is much less forgiving than HTML, notably with &foo; codes, or "character entity references," as they're called. That is, the HTML could have this:

```
<a href="...">Proctor & Gamble to merge with H&R
      Block</a>
```

That's acceptable (if not proper) HTML—but it's strictly forbidden in XML, and will cause any XML parser to reject that document. In XML, if there's an *&*, it must be the start of a character entity reference, and if you mean a literal "&", it must be expressed as with just such a *&foo;* code—typically as &amp;, but also possibly as &#38; or &#x26;.

Moreover, just because something is a legal HTML &foo; code doesn't mean it's legal in an RSS file. For the sake of compatibility, the RSS 0.91 DTD (at the URL you see in the <!DOCTYPE…> declaration) defined the same &foo; codes as HTML—but that was the HTML of several years ago, back when there were just codes for the Latin-1 characters 160 to 255. This gets you codes like &eacute;, but if you try using more recent additions like &euro; or &mdash;, the RSS parser will fail to parse the document.

So just to be on the safe side, we should decode all the *&foo;* codes in the HTML, and then reencode everything, except using numeric codes (like &#123;), since those are always acceptable to XML parsers. And while we're at it, we should kill any tags inside that HTML, in case the HTML that we captured happens to contain some <br>s, which would become malformed as XML. To do the *&foo;* decoding, we can use the ever-useful HTML::Entities module (available in CPAN as part of the HTML-Parser distribution). Then we do a little cleanup and use a simple regexp to replace each unsafe character (like & or é) with a *&#number;* code. The eminently reuseable routine for doing this looks like Listing 3.

Hooking It All Together

Once we've got the xml_string routine as previously defined, we can then use it in a routine that takes the contents of our *@items* (alternating title and URL), and returns XML as a series of <item>…</item> elements, as in Listing 4.

```
<?xml version="1.0"?>
  <!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
    "http://my.netscape.com/publish/formats/rss-0.91.dtd">
  <rss version="0.91"><channel>
    <title>Fresh Air</title>
    <description>Terry Gross's interview show on NPR</description>
    <link>http://freshair.npr.org/dayFA.cfm?todayDate=current</link>
    <language>en-us</language>

    <item>
      <title>John Lasseter</title>
      <link>http://www.npr.org/ramfiles/fa/20020920.fa.01.ram</link>
    </item>
    <item>
      <title>Singer and guitarist Jon Langford</title>
      <link>http://www.npr.org/ramfiles/fa/20020920.fa.02.ram</link>
    </item>
    <item>
      <title>Film critic David Edelstein</title>
      <link>http://www.npr.org/ramfiles/fa/20020920.fa.03.ram</link>
    </item>
  </channel></rss>
```

Figure 1: File produced by the Fresh Air ***HTML scanner.***

We can test that routine by doing this:

```
print rss_body("Bogodyne rockets > 250&frac12;/
      share!","http://test");
```

Its output is this:

```
<item>
     <title>Bogodyne rockets &#62; 250&#189;/
         share!</title>
     <link>http://test</link>
</item>
```

This is correct, since &#62; is XMLese for "a literal > character," and &#189; is for "a literal 1/2 character." Since this is all working happily, we can make another routine for the start of the XML document, as shown in Listing 5. Then spitting out the bare-bones RSS XML that we're after is just a matter of making the call shown in Listing 6.

Run the program, and it indeed spits out the valid XML shown in Figure 2. To do the same for our Fresh Air program, we just append the same code, changing the parameters to rss_start, as in Listing 7. Running that program returns the RSS expression of our *@items* as shown in Figure 3.

And because we put everything through our xml_escape routine, the XML text is always properly escaped, even if our original HTML scanner regexp happens to trap an HTML tag or malformed *&foo;* code. A trip to http://feeds.archive.org/validator/ allows us to validate our code, making sure that the XML is all properly formed.

That's all there is to making a basic RSS generator program. The only question left is how to have it run.

```
<?xml version="1.0"?>
  <!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
    "http://my.netscape.com/publish/formats/rss-0.91.dtd">
  <rss version="0.91"><channel>
    <title>Guardian World Latest</title>
    <description>Latest Headlines from the Guardian</description>
    <link>http://www.guardian.co.uk/worldlatest/</link>
    <language>en-GB</language>
  <item>
        <title>Unsolved Crimes Vex Afghanistan</title>
        <link>http://www.guardian.co.uk/worldlatest/story/0,1280,-
2035841,00.html</link>
  </item>
  <item>
        <title>Christians Show Support For Israel</title>
        <link>http://www.guardian.co.uk/worldlatest/story/0,1280,-
2035838,00.html</link>
  </item>
  <item>
        <title>Schroeder&#39;s Party Wins 2nd Term</title>
        <link>http://www.guardian.co.uk/worldlatest/story/0,1280,-
2035794,00.html</link>
  </item>
 ...
  <item>
        <title>Yemen Holds 104 Terror Suspects</title>
        <link>http://www.guardian.co.uk/worldlatest/story/0,1280,-
2035650,00.html</link>
  </item>
  </channel></rss>
```

Figure 2: File produced by the* Guardian *HTML scanner.

```
<?xml version="1.0"?>
  <!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
    "http://my.netscape.com/publish/formats/rss-0.91.dtd">
  <rss version="0.91"><channel>
    <title>Fresh Air</title>
    <description>Terry Gross&#39;s interview show on National Public
Radio</description>
    <link>http://freshair.npr.org/dayFA.cfm?todayDate=current</link>
    <language>en-US</language>
    <item>
        <title>John Lasseter</title>
        <link>http://www.npr.org/ramfiles/fa/20020920.fa.01.ram</link>
    </item>
    <item>
        <title>Singer and guitarist Jon Langford</title>
        <link>http://www.npr.org/ramfiles/fa/20020920.fa.02.ram</link>
    </item>
    <item>
        <title>Film critic David Edelstein</title>
        <link>http://www.npr.org/ramfiles/fa/20020920.fa.03.ram</link>
    </item>
  </channel></rss>
```

Figure 3: File produced by the* Fresh Air *HTML scanner, using the xml_escape routine.

Running via CGI or via Cron

There are two main ways to use an RSS generator program—it should either run as a CGI and send output to the browser on demand, or it should be run periodically via cron, and save output to a file that can be accessed at some URL.

The mechanics are simple. If the program is to run as a CGI, just start its output out with a MIME header like so:

```
print "Content-type: application/rss+xml\n\n",
  rss_start(
  ... and the rest, as before ...
```

If you want to save the output to a file, instead do this:

```
my $outfile = '/home/jschmo/public_html/freshair.rss';
open(OUTXML, ">$outfile")
  || die "Can't write-open $ouffile: $!\nAborting";

print OUTXML
  rss_start(
  ... and the rest, as before ...
```

The more complex issue is: Under what conditions would you want to do it one way or the other way? If the program runs as a CGI, it will connect to the remote server to get the HTML as many times as there are requests for the RSS feed. If this is an RSS feed that only you know about, and you access it only a few times a day at most, then having it run as a CGI is just fine.

But if the RSS feed might be accessed often, then it would be more efficient for your server, as well as for the remote server, if you have the RSS updater run periodically via cron, as with a crontab line like this:

```
13 6-17 * * 1-5 /home/jschmo/make_fresh_air_rss
```

That will run the program at 13 minutes past the hour between 6:13AM and 5:13PM, Monday through Friday—and those are the only times that it will request the HTML from the server, no matter how many times the resulting RSS file gets hit. Implicit in those crontab settings is the assumption that we don't really need absolutely up-to-the-minute information (or else we'd set it to run more often, or just go back to using the CGI approach) and that there's no point in accessing the RSS data outside of those hours. Since *Fresh Air* is produced only once every weekday, I've judged that it's very unlikely that their HTML listings page will change outside of those hours.

You should always be considerate of the remote web server, so you should request its HTML only as often as necessary. Not only does this approach go easy on the remote server, it also goes easy on your server (which is running the RSS generator). This is fitting, considering that the whole point of an RSS file is to bring people to the content they're interested in, as efficiently as possible, from the points of view of the people and of the web servers involved.

TPJ

Listing 1

```
use LWP::Simple;
  my $content_url = 'http://freshair.npr.org/dayFA.cfm?todayDate=current';
  my $content = get($content_url);
  die "Can't get $content_url" unless defined $content;
  $content =~ s/(\cm\cj|\cj|\cm)/\n/g; # nativize newlines

  my @items;
  while($content =~ m{
  \s+<A HREF="([^"\s]+)">Listen to
  \s+<FONT FACE="Verdana, Charcoal, Sans Serif" COLOR="#ffffff" SIZE="3">
  \s+<B>(.*?)</B>
  }g) {
    my($url, $title) = ($1,$2);
    print "url: {$url}\ntitle: {$title}\n\n";
    push @items, $title, $url;
  }
```

Listing 2

```
use LWP::Simple;
  my $content_url = 'http://www.guardian.co.uk/worldlatest/';
  my $content = get($content_url);
  die "Can't get $content_url" unless defined $content;
  $content =~ s/(\cm\cj|\cj|\cm)/\n/g; # nativize newlines

  my @items;
  while($content =~
   m{<A HREF="(/worldlatest/.*?)">(.*?)</A><BR><B>.*?</B><P>}g
  ) {
    my($url, $title) = ($1,$2);
    print "url: {$url}\ntitle: {$title}\n\n";
    push @items, $title, $url;
  }
```

Listing 3

```
use HTML::Entities qw(decode_entities);

  sub xml_string {
    # Take an HTML string and return it as an XML text string

    local $_ = $_[0];
    # Collapse and trim whitespace
    s/\s+/ /g;  s/^ //s;  s/ $//s;

    # Delete any stray HTML tags
    s/<.*?>//g;

    decode_entities($_);

    # Substitute or strike out forbidden MSWin characters!
    tr/\x91-\x97/''""*\x2D/;
    tr/\x7F-\x9F/?/;

    # &-escape every potentially unsafe character
    s/([^ !#\$%\x28-\x3B=\x3F-\x7E])/'&#'.(ord($1)).';'/seg;

    return $_;
  }
```

Listing 4

```
sub rss_body {
    my $out = '';
    while(@_) {
      $out .= sprintf
       "  <item>\n\t<title>%s</title>\n\t<link>%s</link>\n  </item>\n",
       map xml_string($_),
           splice(@_,0,2); # get the first two each time
    }
    return $out;
  }
```

Listing 5

```
sub rss_start {
    return sprintf q[<?xml version="1.0"?>
  <!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
    "http://my.netscape.com/publish/formats/rss-0.91.dtd">
  <rss version="0.91"><channel>
    <title>%s</title>
    <description>%s</description>
    <link>%s</link>
    <language>%s</language>
  ],
    map xml_string($_),
      @_[0,1,2,3];  # Call with: title, desc, URL, language!
  }
...and for the end:
  sub rss_end {
    return '</channel></rss>';
  }
```

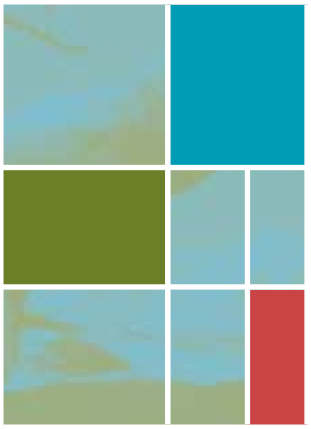
Listing 6

```
print
   rss_start(
     "Guardian World Latest",
     "Latest Headlines from the Guardian",
     $content_url,
     'en-GB',  # language tag for UK English
   ),
   rss_body(@items),
   rss_end()
  ;
```

Listing 7

```
print
   rss_start(
     "Fresh Air",
     "Terry Gross's interview show on National Public Radio",
     $content_url,
     'en-US',  # language tag for US English
   ),
   rss_body(@items),
   rss_end()
  ;
```

TPJ



Better Documentation Through Testing

brian d foy

I document all of my modules and scripts, and although I readily find out about program errors because people are quick to tell me their script either fails to run or does the wrong thing, almost nobody reports problems with the documentation.

Perl has a basic document format called "Plain Old Documentation," or POD. The Perl documentation reader, perldoc, follows the adage, "Be strict in what you output, and liberal in what you accept." It takes most of the bad POD formatting that I give it and turns it into something the user can read. If I make errors in my POD markup, perldoc often silently fixes them. I have the same problem with HTML. I can write bad HTML that the browser fixes so it can display it.

Perl comes with a POD validator, *podchecker*, based on the *Pod::Checker* module, which is part of the Perl Standard Library. Given a file with POD directives, podchecker tells me where I messed up. I tell podchecker which file (not which module, like perldoc) to check and it shows me the POD errors and questionable constructs; see Listing 1.

I am lazy, in the Perly sense of the word, so I want this to happen automatically. As I add new features to modules or move code around, I change the documentation. I may add more documentation, get rid of old explanations, or rearrange it. Anywhere in this process I might introduce a POD error. I do not want to use podchecker every time I make a change.

I could check the documentation just before I release a new version, but I usually forget to do that. I have automated most of the steps to release something to CPAN, so I should automate checking the documentation format as well.

Test::Pod

I wrote the *Test::Pod* module to automatically check my modules and scripts for POD errors. It does not check if I documented everything that I should have, like *Pod::Coverage* does. It simply wraps *Pod::Checker*, which only checks the POD markup, in a *Test::Builder* interface and calls it *pod_ok()*.

Once I have my *pod_ok* function, which I show in gory detail later, I can use it in a test script. This snippet uses *Test::More*, the basis of many new test files. *Test::More* comes with the latest stable version of Perl, 5.8.0, which also comes with *Test::Tutorial*, which explains the basics of testing.

brian 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. He can be contacted at comdog@panix.com.

```
# t/pod.t
use Test::More tests => 1;
use Test::Pod;

pod_ok( ‘blib/lib/ISBN.pm’ );
```

The test fails if *Pod::Checker* finds a POD error in the ISBN.pm file from my *Business::ISBN* distribution.

Most of my distributions have more than one module in it, though. Andy Lester showed me a cool hack to test all of the modules without remembering which modules I had. If I add a new module to the distribution, this test finds it automatically.

```
# t/pod.t
BEGIN {
    use File::Find::Rule;
    @files = File::Find::Rule->file()->name(
        ‘*.pm’ )->in( ‘blib/lib’ );
    }

use Test::More tests => scalar @files;
use Test::Pod;

foreach my $file ( @files )
    {
    pod_ok( $file );
    }
```

The *File::Find::Rule* module allows me to find all of the modules in the named directory and the build library (‘blib/lib’ in this case) very easily. Once I have all the filenames that I want to test in *@files*, I simply loop through *@files* to test each one.

I run this test like any other test file. Once I create the module Makefile from Makefile.PL, I run *make test*, which tells *Test::Harness* to do its magic. It runs all of the *.t files in the t directory, collects the results, and reports what it finds; see Listing 2.

If a test fails, the *Test::Harness* report is different. In this case, I edited the ISBN.pm file to remove an *=over* POD directive so the POD now has an error. *Test::Pod* reports that it found an error at line 400. *Pod::Checker* correctly identifies the problem as a missing *=over*. *Test::Harness* prints a summary of the failed tests at the end; see Listing 3.

Creating the Test Module

Test::Builder is the brainchild of Michael Schwern and chromatic, and in my opinion, it's the best thing to happen to Perl in years. This module allows other people to plug into *Test::Harness* with their own specialized modules. The Comprehensive Perl Archive Network (CPAN) has about 20 specialized Test modules so far.

These specialized modules have the same advantages as any other module—I can write a test function once and use it over and over again. The function standardizes the way I do things and moves more code out of the test files. I take any chance I can get to move code out of the test files. More code means more points of failure. I have to try really hard to mess up *pod_ok()*.

The *Test::Builder* interface is very simple. In my specialized test module, I create a *Test::Builder* object, which is a singleton, so all of the specialized test scripts play well together.

```
my $Test = Test::Builder->new();
```

I create a test function, called *pod_ok* in *Test::Pod*, that tells *Test::Builder* if the test succeeded or failed, and optionally outputs some error information. *Test::Builder*'s *ok()* method handles the result. If I think the test passed, I give *ok()* a true value, and a false value otherwise. The meat of *pod_ok()* is very simple—tell *Test::Builder* the test either passed or failed.

```
sub pod_ok
     {
     $pod_ok = _check_pod();
     if( $pod_ok ) { $Test->ok(1) }
     else          { $Test->ok(0) }
     }
```

Everything else in the function supports those simple statements in the *pod_ok* function. *Test::Builder* takes care of the rest.

To actually test the POD, I created an internal function, *_check_pod*, in *Test::Pod*; see Listing 4. I already defined the constants NO_FILE, NO_POD, ERRORS, WARNINGS, and OK, which described the conditions that *Pod::Checker* can report. The function returns an anonymous hash. The result is the value of the *result* key, error messages are the value for the *output* key, the number of errors is the value of the *errors* key, and the number of warnings is the value for the *warnings* key. The rest of the code puts the right things in the hash.

The first line in the subroutine takes a filename off of the argument stack, and the third line checks the file's existence. If the file does not exist, the function returns an anonymous hash with the result NO_FILE.

The next bit of code ties the variable *$hash{output}* to *IO::Scalar*. *Pod::Checker* can write messages to a file handle, and I want to intercept that output. It shows up in *$hash{output}* instead of the terminal.

I get a POD checker object from *Pod::Checker*, and then parse the specified file. *Pod::Checker* puts all output in *$hash{output}*. The *do* block puts the numbers of errors and warnings in the right keys, then returns the right constant for the condition that *Pod::Checker* reported.

Finally, *_check_pod* returns the anonymous hash that the *pod_ok* function can use to tell *Test::Builder* what happened. The *pod_ok()* function (see Listing 5) does the same thing it did before, although it has to do a little more work to decide if the test passed or not.

The first line takes a file name off of the argument list, and then passes that to *_check_pod*. The second argument specifies my expected result and default to OK. The third argument gives a name to the test that I can see in the verbose output of *Test::Harness*. It defaults to a message that includes the name of the file.

Don't miss

Lightweight Languages Workshop 2002 (LL2)

Saturday, November 9, 2002
Room 34-101, MIT, Cambridge, MA

http://ll2.ai.mit.edu

LL2 will be an intense, exciting, one-day forum bringing together the best programming language implementors and researchers, from both academia and industry, to exchange ideas and information, to challenge one another, and to learn from one another.

Erlang, Python, Needle, Ruby, Laszlo, Perl, and more!

To register, send e-mail to
ll2-register@ai.mit.edu

The rest of the function is a series of *if-elsif* statements, with one test for each possible condition. I can specify an expected result in the second argument. If *Pod::Checker* finds the condition I expected, then the test succeeds, and fails otherwise. If I do not specify an expected condition, *pod_ok* assumes I only want the test to pass if *Pod::Checker* finds neither error nor warnings.

Every branch of the *if-elsif* structure calls *Test::Builder*'s *ok()* method. If that branch represents a success, *pod_ok* passes 1 to *ok()*, and 0 otherwise.

If *pod_ok* fails a test, it also uses *Test::Builder*'s *diag()* method to give an error message.

Conclusion

Testing my work is simple if I use *Test::Builder*. I can create specialized test modules to check all sorts of things other than normal script execution. My modules have better-formatted documentation because *Test::Pod* automatically tells me about problems.

TPJ

Listing 1

```
podchecker /usr/local/lib/perl5/site_perl/darwin/DBI.pm

*** WARNING: (section) in 'perl(1)' deprecated at line 4926 in file
/usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: (section) in 'perlmod(1)' deprecated at line 4926 in file
/usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: (section) in 'perlbook(1)' deprecated at line 4926 in file
/usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** ERROR: unresolved internal link 'bind_column' at line 1819 in file
/usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: multiple occurence of link target 'trace' at line - in file
/usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: multiple occurence of link target 'Statement (string, read-
only)' at line - in file /usr/local/lib/perl5/site_perl/darwin/DBI.pm
/usr/local/lib/perl5/site_perl/darwin/DBI.pm has 1 pod syntax error.
```

Listing 2

```
localhost_brian[3150]$ make test
cp ISBN.pm blib/lib/Business/ISBN.pm
cp Data.pm blib/lib/Business/ISBN/Data.pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/load.t t/pod.t t/isbn.t
t/load....ok
t/pod.....ok
t/isbn....ok 20/21
Checking ISBNs... (this may take a bit)
t/isbn....ok
All tests successful.
Files=3, Tests=25, 358 wallclock secs (116.90 cusr +  2.96 csys = 119.86
CPU)
```

Listing 3

```
localhost_brian[3152]$ make test
cp ISBN.pm blib/lib/Business/ISBN.pm
Skip blib/lib/Business/ISBN/Data.pm (unchanged)
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e"
"test_harness(0, 'blib/lib', 'blib/arch')" t/load.t t/pod.t t/isbn.t
t/load....ok
t/pod.....NOK 1#     Failed test (t/pod.t at line 12)
# Pod had errors in [blib/lib/Business/ISBN.pm]
# *** ERROR: =item without previous =over at line 400 in file
blib/lib/Business/ISBN.pm
# blib/lib/Business/ISBN.pm has 1 pod syntax error.
t/pod.....ok 2/2# Looks like you failed 1 tests of 2.
t/pod.....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/2 tests, 50.00% okay
t/isbn....ok 20/21
Checking ISBNs... (this may take a bit)
t/isbn....ok
Failed Test Stat Wstat Total Fail  Failed  List of Failed
--------------------------------------------------------------------------
t/pod.t        1   256     2    1  50.00%  1
Failed 1/3 test scripts, 66.67% okay. 1/25 subtests failed, 96.00% okay.
make: *** [test_dynamic] Error 35
```

Listing 4

```
sub _check_pod
    {
    my $file = shift;

    return { result => NO_FILE } unless -e $file;

    my %hash    = ();
    my $output;
    $hash{output} = \$output;

    my $checker = Pod::Checker->new();

    # i pass it a tied filehandle because i need to fool
    # Pod::Checker into thinking it is sending the errors
    # somewhere so it will count them for me.
    tie( *OUTPUT, 'IO::Scalar', $hash{output} );
    $checker->parse_from_file( $file, \*OUTPUT);

    $hash{ result } = do {
        $hash{errors}   = $checker->num_errors;
        $hash{warnings} = $checker->can('num_warnings') ?
            $checker->num_warnings : 0;

           if( $hash{errors} == -1  ) { NO_POD   }
        elsif( $hash{errors}   > 0  ) { ERRORS   }
        elsif( $hash{warnings} > 0  ) { WARNINGS }
        else                          { OK }
        };

    return \%hash;
    }
```

Listing 5

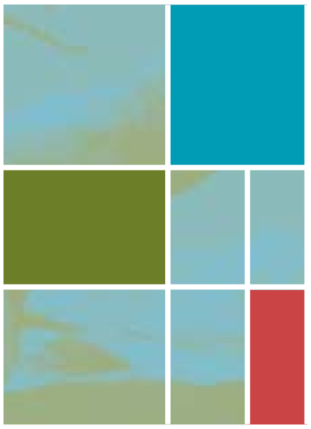
```
sub pod_ok
    {
    my $file     = shift;
    my $expected = shift || OK;
    my $name     = shift || "POD test for $file";

    my $hash = _check_pod( $file );

    my $status = $hash->{result};

    if( defined $expected and $expected eq $status )
        {
        $Test->ok( 1, $name );
        }
    elsif( $status == NO_FILE )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Did not find [$file]" );
        }
    elsif( $status == OK )
        {
        $Test->ok( 1, $name );
        }
    elsif( $status == ERRORS )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Pod had errors in [$file]\n",
            ${$hash->{output}} );
        }
    elsif( $status == WARNINGS and $expected == ERRORS )
        {
        $Test->ok( 1, $name );
        }
    elsif( $status == WARNINGS )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Pod had warnings in [$file]\n",
            ${$hash->{output}} );
        }
    elsif( $status == NO_POD )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Found no pod in [$file]" );
        }
    else
        {
        $Test->ok( 0, $name );
        $Test->diag( "Mysterious failure for [$file]" );
        }
    }
```

TPJ



Really Lazy Persistence

Simon Cozens

It comes up fantastically often—you've got some piece of data stored in a variable; maybe a cache or a hash full of CGI session data. You want the data in the variable to be available next time you run your Perl program. This is the well-known "data persistence problem."

Thankfully, the data persistence problem has a huge number of solutions; if you want to make lots and lots of data persistent, you can look at SQL backended object databases or Perl modules such as *Tangram*, *Alzabo*, *Pixie*, *DBIx::SearchBuilder*, and the like. Or for the much more common small cases where you don't need a full-blown RDBMS to store your data, you can use one of the DBM libraries.

DBM is an ancient UNIX database file format, which stores key-value pairs, much like hashes. There are a bunch of libraries around that implement DBM: The Berkeley database from Sleepycat, *DB_File*, is the most flexible and reliable, but *NDBM* and *GDBM* are two others.

So, we get out *DB_File*, choose some filenames, tie the variables we need tied, and that's it, right? Problem solved.

Well, not exactly, because not everyone has *DB_File* installed; so there's a proxy module called "AnyDBM_File" that tests to see which DBM libraries are actually available. Great.

Actually, there's another snag. One limitation of the DBM format is that it doesn't understand complex data structures—it only stores keys and values as strings. No big deal—the multilevel DBM (*MLDBM*) Perl module sits between Perl and the DBM library, and automatically serializes and unserializes references, making it possible for you to use data structures in your persistent variables.

The Real Problem

Although we've found a workable solution, the upshot of all this is that in order to get a hash to persist, we need to think up a filename, find a safe place to put the file, load the *AnyDBM_File* library, load *MLDBM*, remember the syntax to *tie*, and so on; you end up with code looking like this:

```
use AnyDBM_File;
use MLDBM qw(AnyDBM_File);
use File::Spec::Functions qw(catdir tmpdir);
@AnyDBM_File::ISA = qw(DB_File NDBM GDBM);
# Prefer DB_File
my (%hash, @array);

# Must remember to make sure nobody else is using
# these file names!
my $hash_file = catdir(tmpdir(), "hashdatabase");
my $array_file = catdir(tmpdir(), "arraydatabase");

tie %hash, "MLDBM", $hash_file;
tie @array, "MLDBM", $array_file;
```

That's nine lines of real code just to make two variables persist; and we still have to make sure that no other application decides to choose the same names for its database files. What happened to making easy things easy and hard things possible?

This is the *real* data persistence problem: We have a solution, but it forces us to do a lot of work, and we'd rather be lazy.

The Solution

Enter *Attribute::Persistent*. Here's the same code as above, written to use *Attribute::Persistent*.

```
    use Attribute::Persistent;
    my (%hash, @array) :persistent;
```

As if that wasn't enough, you get an additional guarantee that no other application will corrupt your files. Isn't that more like what we should expect from Perl?

So, you're probably wondering how this works and what it really does. We'll start by looking at Perl's attributes system, where we'll find a story that reflects the "real" data persistence problem.

A Bit About Attributes

Attributes appeared in Perl 5.005 via the now-deprecated *attrs* module. These let you specify attributes on a subroutine, although they were restricted to "locked," which put an exclusive lock around the *sub* in threaded Perl, and "method." Perl 5.005 attributes also had the following ugly syntax:

```
sub blast {
    use attrs qw(locked method);
    ...
}
```

Simon is a freelance programmer and author, whose titles include Beginning Perl *and* Extending and Embedding Perl. *He's the creator of over 30 CPAN modules, a former Parrot pumpking, and an obsessive player of the Japanese game of* Go. *Simon can be reached at simon@simon-cozens.org.*

5.6 tidied this up a little, to the now-familiar syntax:

```
sub blast :locked :method {

}
```

and added user-definable attributes and the ability to apply attributes to variables as well as subroutines.

Unfortunately, the mechanism for doing anything with attributes is rather obscure. If you say

```
my $pencil :color(blue);
```

then a call is made to a class method called *MODIFY_SCALAR_ATTRIBUTES* in the current package. This method receives a reference to the variable or subroutine that is having a method applied, and then all the attribute names in full; it's then expected to return the names of the attributes it couldn't do anything with.

So you ended up with something like

```
sub UNIVERSAL::MODIFY_SCALAR_ATTRIBUTES {
    my ($self, $ref, @attribs) = @_;

    for (@attribs) {
        if (/color\(\w+\)/) {
            tie $$ref, "String::Colored", $1;
        } else {
            push @unknown, $_;
    }

    return @unknown;
}
```

This is OK, but it's not very friendly; it's also especially problematic if you want to handle more than one attribute name in your class—or indeed, if you wanted several modules to each be able to chip in a bunch of attributes they could handle.

So Damian Conway produced a module called *Attribute::Handlers*, which makes the process a little more transparent. Now all you need to do is create a subroutine with the same name as the attribute you're trying to define, and declare that this is an attribute handler. How do you declare that? With an attribute, naturally!

```
use Attribute::Handlers;
sub color :ATTR(SCALAR) {
    my ($self, $symbol, $referent, $attr, $data,
      $phase) = @_;
    tie $$referent, "String::Colored", $data;
}
```

As you can see, this is much easier to deal with, and you also get a lot more information about the attribute: the symbol table entry (typeglob) and the reference that are having the attribute applied; the attribute name, any data ("*blue*" in our example) passed to the attribute, and at what stage of Perl's parsing of the program the attribute was applied.

Since we have a typeglob, we can even get at the variable's name, using a little-known typeglob trick:

```
my $varname = *{$symbol}{NAME};
```

Of course, in this code, the idea of using an attribute to tie a variable is not an uncommon one, and so *Attribute::Handlers* gets even friendlier:

```
use Attribute::Handlers autotie => { color =>
      "String::Colored" };
```

All this infrastructure has made it very easy to implement a load of cool, attribute-based modules: *Attribute::Memoize*, *Attribute::Types*, and *Attribute::Util* being the most interesting three.

But not *Attribute::Persistent*; that had yet another problem. I wanted *Attribute::Persistent* to choose names for the databases that were related to the name of the variable being made persistent. Unfortunately, even though *Attribute::Handlers* provides a lot of information to developers, it can't provide the variable name for lexical variables—they don't live in a symbol table, so we can't use the **glob{NAME}* trick on the symbol table entry to find the variable name.

To get around this obstacle, I used another Damian *module*—*Attribute::Handlers::Prospective*. This used a source filter to pass on more information about the attribute, including—thankfully for me—even the name of lexicals.

The Finished Product

So now we can take a peek inside *Attribute::Persistent* and see how it was done.

The first few lines should not be any surprise to anyone.

```
package Attribute::Persistent;
use strict;
our $VERSION = "1.0";
```

Now we needed a means to identify the user's program uniquely, so that other programs on the same system using *Attribute::Persistent* don't stomp over its databases. We call this the key.

```
my $key;
```

You might think that we should generate this from the filename; I thought that too, but then I realized that it wasn't very unlikely to have two different *foo.pl*s lying around. So if possible, we take an MD5 sum of the source file, which reduces chances of a collision to 1 in 10^{155}; if MD5 collides, we have bigger problems anyway.

```
require Digest::MD5;
local *IN;
if (-e $0 and open IN, $0) {
    local $/;
    my $x = <IN>;
    $key = Digest::MD5::md5_hex($x);
    close IN;
} else {
    $key = "Persistent$0";
}
1;
```

This means, of course, that when you change the source code to a program, the MD5 sum changes, and now you can't find your old data any more. Just think of it as a way of automatically flushing the cache when you update your code…

And that's all we do in the *Attribute::Handlers* package. Because we want this attribute to be available to all packages, we have to put the real meat into the *UNIVERSAL* package.

```
package UNIVERSAL;
use Attribute::Handlers::Prospective;
```

Now comes all that rubbish you saw before when handling the DBM file:

```
use File::Spec::Functions (':ALL');
BEGIN { @AnyDBM_File::ISA = qw(DB_File GDBM_File
      NDBM_File) }
```

```
use AnyDBM_File;
use MLDBM qw(AnyDBM_File);
no strict; # Attributes do evil things
```

And finally, we can define the attribute:

```
sub persistent :ATTR(RAWDATA) {
```

The first thing we want to do is find the name of the variable we've been passed. We also demand that this is a lexical:

```
    *{$_[1]}{NAME} =~ /LEXICAL\((.*)\)/ or do {
        require Carp;
        croak("Can only define :persistent on
              lexicals");
    };
```

And now we have the variable name, complete with sigil, in *$1*. We need to use this in two ways: We need to keep the original name, for reporting errors, and we need to munge it into something suitable for use as part of a filename:

```
    my $name = $1;
    my $origname = $name;
```

We need to know what type the variable is, so that we can dereference its reference correctly; this also helps us set up the filename so that *%foo* comes out as *H-foo* and *@foo* comes out as *A-foo*. This way, we can have two different persistent *foo* variables and they'll be kept separate:

```
$name =~ s/^\%/H-/ and $type = '%';
$name =~ s/^\@/A-/ and $type = '@';
```

However, we also allow the user to explicitly give their own name to a variable as an argument to the *:persistent* attribute, which we'll use as part of the filename:

```
     if ($_[4] ne "undef") { $name = $_[4]; }
```

Now we make it filesystem-safe and put it in the temporary directory:

```
$name =~ s/\W+/-/g;
my $filename = catdir(tmpdir(),"$key-$_[0]-$name");
```

Finally, we can do the tie by correctly dereferencing the reference we were passed:

```
tie (($type eq "%" ? %{$_[2]} : @{$_[2]}), "MLDBM",
      $filename)
or do {require Carp; croak("Couldn't tie $origname to
      $filename - $!")};
```

And that, essentially, is all of *Attribute::Persistent*.

Philosophical Aside

We've seen two stories here: the story of *Attribute::Persistent*, and the story of attributes in Perl. There's a way in which these two stories are very similar: We started off with a great idea that got implemented a long time ago, but that suffered from having an interface that didn't allow us to be as lazy as we'd like. So, someone bit the bullet and wrote higher level modules to simplify the interface and make things either more powerful or less cluttered for the end programmer.

In the case of attributes themselves, the basic functionality lay unused for quite a while due to the clumsiness of the interface. However, once Damian produced *Attribute::Handlers* and did most of the work, an explosion of impressive uses for the feature sprung into being soon afterwards.

If you look at other parts of Perl, you'll see the same story; source filters, for instance, were a "little known feature" back in 1996. Paul Marquess's *Filter::Util::Call* module did a great job of exposing the C interface to source filters, but still the feature languished in obscurity. Then Damian again came up with a higher level module, *Filter::Simple*, which truly was simple, and suddenly a wealth of filter-based modules appeared. This happens again and again—with pluggable optimizers, the iThreads system, XS versus *Inline*, and so on.

What can we learn from this? First, interface design is much more important than you might think in terms of "selling" an advanced feature. It may be fantastically powerful, but even though you're writing for other programmers, if nobody's interested in getting their heads around the interface, it'll be largely ignored. Laziness is a Perl virtue, and if you do something clever with Perl, it's important to take the time to allow other people to be lazy with it.

Second, we can learn that there are a bunch of very interesting things going on in Perl or in Perl modules that nobody knows about. Source filters were in Perl for over four years before they became vogue. Take a look around, and you too may be able to create a "middle man" module that does the hard work, allowing others to be lazy in playing around with the next great feature.

Third, we can be thankful that there are people like Damian Conway who are willing to do the hard work of putting obscure and powerful features into easy-to-use packages. As we saw with the history of using DBMs, there's a trend towards more and more abstractions making the end-programmer's life easier. Writing the abstraction code that does the heavy work so another programmer doesn't have to is unglamorous and generally thankless, but it means you'll be doing the biggest favor possible to your fellow developers—you'll be saving them time.

TPJ

Extending and Embedding Perl

Jack J. Woehr

Extending and Embedding Perl, which covers both writing Perl extensions in C and calling Perl from within C, is a significant addition to the literature of one of the world's most widely used programming languages. Authors Tim Jenness and Simon Cozens demystify in precisely measured and carefully annotated terms the most arcane aspect of Perl, that of integrating Perl at the compiler level with other languages, primarily C, but also Fortran. If you already know how to script in Perl, have written modules or want to write your first module, and want to learn how to write modules in C or Fortran or call Perl from C routines, this economical volume is your best and pretty much your only written guide outside of the documentation bundled with the Perl interpreter.

The book is divided into two parts: The first part focuses on extending Perl, the second on embedding Perl. The reader's progress is methodical and orderly. The book starts with creating a Perl module in Perl, teaching just enough C to allow the same module to be written in C and called from Perl. After that, we're fully immersed in Perl internals and the tools Perl provides to match up those internals with the requirements of external languages. Everything is explained in text, with plenty of compilable examples and tidy diagrams that show the layout and relationships of data structures in a familiar and readable style of notation.

The chapters forming this extraordinarily self-contained book are:

1. C for Perl Programmers
2. Extending Perl: An Introduction
3. Advanced C
4. Perl's Variable Types
5. The Perl 5 API
6. Advanced XS Programming
7. Alternatives to XS
8. Embedding Perl in C
9. Embedding Case Study
10. Introduction to Perl Internals
11. Hacking Perl

followed by an appendix on Perl's typemaps, a bibliography, and an API index.

Authored by two individuals who have years of experience in writing published Perl modules and interfacing Perl as part of critical real-world applications, *Extending and Embedding Perl* is characterized by profound expertise, attention to detail, literate writing, and excellent editorial and production values. This book is so necessary that it would be a must-have for the problem domain, even were it less finely crafted; we're lucky that the one book to adequately address the most puzzling facets of Perl happens to be a masterpiece that will keep its value for years.

Extending and Embedding Perl
Tim Jenness and
Simon Cozens
Manning, 2003
384 pp., $44.95
ISBN 1-930110-82-0

The publisher is making the source code for the examples available via the Web. The downloadable package is still being polished as I write, but the package provided for review was accurate, and the whole should surely be complete by the time you read this. The book's web site is http://www.manning.com/jenness/, where you can buy the electronic version of the book online and discuss Perl with author Tim Jenness in a forum with readers.

Tim Jenness is 31. He got his Ph.D. from the University of Cambridge at the age of 24. We spoke with Tim by phone on September 26, 2002.

TPJ**:** What's it like being a computer programmer in Hawaii?
Tim: It's great. I work at the Joint Astronomy Centre in Hawaii (http://www.jach.hawaii.edu/JACpublic/index.html). I am the manager of the JAC high-level software group, responsible for all the high-level software (observation preparation, data processing) at the James Clerk Maxwell Telescope and at the United Kingdom Infrared Telescope, both on Mauna Kea. We have only three or four people in the group and you're allowed your own design, implementation, and testing.

Jack is an independent consultant specializing in mentoring programming teams and a contributing editor to Dr. Dobb's Journal. *His web site is http://www.softwoehr.com/.*

TPJ: Between you and coauthor Simon Cozens, who wrote which parts of the book?
Tim: It's about 55 percent to 45 percent my favor, which is why I'm credited first author. I concentrated on extending Perl and Simon described embedding Perl. Simon got the idea, but he had already written a book, and there was no way he was doing that again on his own!

TPJ: I've done XS and I have never seen before an adequate book on this subject. Is there one other than yours?
Tim: The O'Reilly Associates book *Advanced Perl Programming* (http://www.oreilly.com/catalog/advperl/) had two chapters to cover everything, and that was four years old. Beside that, just the man pages for Perl. There was clearly a niche. I'm not sure if that was because no one had thought of it before. Maybe no one was brave enough before.

In 1995, I had to interface one of our esoteric data-reading I/O libraries. It was not practical to write a big C program just to list out headers and write a little table on the screen. I felt the pain as I learned how to do the interface. I tried to remember that pain when writing the book.

TPJ: In the book, you teach the Perl programmer just enough C to do the tasks in each chapter.
Tim: In the early stages, people asked, "Why are you doing this?" But if you are a Perl programmer, there are similarities that will hurt you when you go to C, that make you wonder why your code is not working.

The goal was that after Chapters 1 and 2, if you have a simple library with a C interface, you can build the Perl interface to it. Initially there was an objection to splitting C coverage as occurs in Chapter 2, but I felt I should give the reader enough C to do a simple interface without having to worry about pointers. I was careful that Chapter 2 doesn't use any of the things—arrays, pointers, strings—that are covered in Chapter 3.

TPJ: It's an exceptionally useful and readable book.
Tim: We could have written a book twice the size, but the theme is to get you in the door.

TPJ: It seems most people in the world are programming in C++, Java, and/or Perl.
Tim: We use Fortran here quite a bit—there are so many scientific libraries we have to use. The JCMT is from the 1980s and the UKIRT is from the 1970s. We still have control systems from 1988.

The users don't see this anymore. Everything the user sees nowadays is on Linux. The UK astronomical community has been using an object-messaging system called ADAM and later DRAMA that they wrote for themselves in the late 1980s. DRAMA can be downloaded off the Anglo-Australian Telescope web site (http://www.aao.gov.au/). You send messages out to tasks on remote systems and they come back and tell you what they are doing. It feels like CORBA. You provide arguments and get back data, synchronously or asynchronously.

In our new stuff we use XML and SOAP and translate to our low-level interface just before the stage where the telescope gets moved. From the user's point of view, you're coming in and using a Linux box with a Java or Perl app. You don't really notice the interface.

TPJ: Do you see much use these days of embedding Perl, of C calling Perl?
Tim: Simon did the embedding because XS is what I knew. But even in XS I find myself doing embedding. Our interface features callbacks, so the C bodies of methods launched by Perl XS have to call back into Perl.

TPJ: So C calling Perl is the yin to the yang of Perl calling C?
Tim: It's an XS module that farms the message off to an event loop in C, which calls back to Perl with the answer. It's a question of, "Who's embedding who?"

TPJ: What's the hot topic in Perl these days?
Tim: I suppose, "Should you be working on getting Perl 6 out the door or on maintaining Perl 5?" They're reengineering the internals so that the back end can be used to parse things like Python natively. Most people seem to be thinking, "Perl 6 sounds interesting, but we can't afford to worry about it yet."

TPJ

Source Code Appendix

Moshe Bar “Using Perl to Stress-Test Server Applications”

Listing 1

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

# Part One Modules and parameters
use Parallel::ForkManager;
use DBI;
srand;

############# configruation part #######################
my $processes=30;
my $transactions=5000;
my $rows=100000;  # rows to be added to the database
#######################################################

my $p;

# Part Two - Seed for DB population
@NAMES = qw/Alf Ben Benny Daniel David Foo Bar Moshe
                                        Avivit Jon Linus Larry Safety First/;

#Part Three - DB environment
my $username;
my $money;
my $id;
my $exists=0;
my $commandlineargs=0;

$SID = $ENV{ "ORACLE_SID" };

if (!$SID) {

printf("No ORACLE_SID environment!\n");
printf("... do not know to which database you want to connect\n");
printf("You have to export the database-name in the environment\n");
printf("variable ORACLE_SID e.g.\n");
printf("export ORACLE_SID [mydb]\n");

exit -1
}

# Part Four - openMosix related. unlocks this process and all its children
sub unlock {

#open (OUTFILE,">/proc/self/lock") ||
#ie "Could not unlock myself!\n";
#print OUTFILE "0";

}

unlock;

# Part Five - Here we connect to the DB and populate the tables with records
sub filldb {

my $dbh = DBI->connect( "dbi:Oracle:$SID",
                         "scott",
                         "tiger",
                         {
                           RaiseError => 1,
                           AutoCommit => 0
                         }
                       ) || die "Database connection not made: $DBI::errstr";
                     for ($loop=0; $loop<$rows; $loop++) {
                             # prepare the random values
                             srand;
                             $p=rand();
                             $username = $NAMES[rand(@NAMES)];
                             $money=rand()*(rand()*100);
                             $id=rand()*1000;
                             $id=sprintf("%d", $id);
                             # insert
                              my $sql2 = qq{ insert into stress
                                            values ($id,'$username',$money) };
                              my $sth2 = $dbh->prepare( $sql2 );

                              $sth2->execute();
                              $sth2->finish();
             }
                     $dbh->disconnect();
```

```
}
# delete old entries from db
sub deldb {
my $dbh = DBI->connect( "dbi:Oracle:$SID",
                         "scott",
                         "tiger",
                         {
                           RaiseError => 1,
                           AutoCommit => 0
                         }
                       ) || die "Database connection not made: $DBI::errstr";

                         # drop
                         my $sql1 = qq{ select TABLE_NAME from user_tables
                                                          where TABLE_NAME='STRESS' };
                         my $sth1 = $dbh->prepare( $sql1 );
                         $sth1->execute();

            # if the table does not exists
                         while( $sth1->fetch() ) {
             $exists=1;
                          }
            if (!$exists) {

             # create
                          my $sql2 = qq{ create table STRESS ( id number,
                                           name varchar2(255), money float) };
                          my $sth2 = $dbh->prepare( $sql2 );

                          $sth2->execute();
                          $sth2->finish();

             } else {

                          # cleanup
                          my $sql3 = qq{ delete from STRESS };
                          my $sth3 = $dbh->prepare( $sql3 );

                          $sth3->execute();
                          $sth3->finish();
            }
            $sth1->finish();
                         $dbh->disconnect();
}

# Part Six - Stress-testing the DB with parallel instances

sub stressdb {
my $dbh = DBI->connect( "dbi:Oracle:$SID",
                         "scott",
                         "tiger",
                         {
                           RaiseError => 1,
                           AutoCommit => 0
                         }
                       ) || die "Database connection not made: $DBI::errstr";
        for ($loop=0; $loop<$transactions; $loop++) {
            # prepare the random values
            srand;
            $p=rand();
            $username = $NAMES[rand(@NAMES)];
            $money=rand()*(rand()*100);

            $id=rand()*1000;
            $id=sprintf("%d", $id);

            # update transactioin
            if ($p<0.3) {

             printf("update\n");

             my $sql = qq{ update stress set name='$username',
                                                     money=$money where id=$id };
             my $sth = $dbh->prepare( $sql );

             $sth->execute();
             $sth->finish();
            }

            # select
            if (($p>0.3) and ($p<0.7)) {
             printf("select\n");
             $sql = qq{ select id, name, money from stress };
             my $sth = $dbh->prepare( $sql );

             $sth->execute();
             my( $sid, $sname, $smoney );

             $sth->bind_columns( undef, \$sid, \$sname, \$smoney );
             while( $sth->fetch() ) {
              # print "$sid $sname $smoney\n";
             }
            }

            # delete + insert
            if ($p>0.7) {
             printf("delete/insert\n");

             # delete first
                         my $sql1 = qq{ delete from stress where id=$id };
                         my $sth1 = $dbh->prepare( $sql1 );

                         $sth1->execute();
             $sth1->finish();

            # insert
                         my $sql2 = qq{ insert into stress
                                          values ($id,'$username',$money) };
                         my $sth2 = $dbh->prepare( $sql2 );

                         $sth2->execute();
                         $sth2->finish();

            }
        }
        $dbh->disconnect();
}

############## main ######################

print "\nStarting the orastress test\n\n";

# check for commandline arguments
foreach $parm (@ARGV) {
 $commandlineargs++;
}
$parm=0;
if ($commandlineargs!=3) {

 # get values from the user
 print "How many rows the database-table should have ? : ";

 $rows=<STDIN>;
 chomp($rows);

 while ($rows !~ /\d{1,10}/){
  print "Invalidate input! Please try again.\n";
  print "How many rows the database-table should have ? : ";
  $rows = <STDIN>;
  chomp($rows);
 }

 print "How many clients do you want to simulate ? : ";
 $processes=<STDIN>;
 chomp($processes);
 while ($processes !~ /\d{1,10}/){
  print "Invalidate input! Please try again.\n";
  print "How many clients do you want to simulate ? : ";

  $processes = <STDIN>;
  chomp($processes);
 }

 print "How many transactions per client do you want to simulate ? : ";
 $transactions=<STDIN>;
 chomp($transactions);

 while ($transactions !~ /\d{1,10}/){
  print "Invalidate input! Please try again.\n";
  print "How many transactions per client do you want to simulate ? : ";

  $transactions = <STDIN>;
  chomp($transactions);
 }

 } else {
 # parse the values from the command line

 $rows = $ARGV[0];
 $processes = $ARGV[1];
 $transactions = $ARGV[2];

 if (($rows !~ /\d{1,10}/) or ($processes !~ /\d{1,10}/) or
                                            ($transactions !~ /\d{1,10}/)) {
  print ("Invalidate input! Please try again.\n");
  print ("e.g.   stress-oracle.pl 10 10 10\n");

  exit -1;
  } else {
  print("got the values for the stress test from the command line\n");
  print("rows = $rows\n");

  print("processes = $processes\n");
  print("transactions = $transactions\n");

 }
}

# cleaning up old db
print("delete old entries from db\n");

deldb();

print("fill db with $rows rows\n");

filldb();

# Part Seven
print("starting $processes processes\n");
my $pm = new Parallel::ForkManager($processes);

$pm->run_on_start(
  sub { my ($pid,$ident)=@_;
  print "started, pid: $pid\n";
  }
);

for ($forks=0; $forks<$processes; $forks++){
  $pm->start and next;
  stressdb;

  $pm->finish;
}

$pm->wait_all_children;
$pm->finish;

print "Simulation finished\n";
```

Robert Kiesling "perlcc & Compiling Perl Script"

Listing 1

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

print "Hello, world!\n";
```

Listing 2

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

use IO::File;

my $filename = $ARGV[0];

if (! length ($filename)) {
    print "Usage: touch <filename>\n";
    exit 1;
}

if (-f $filename) {  # Update access and modtimes on $filename if
                     # it already exists.
    my $timenow = time;
    utime $timenow, $timenow, $filename;
} else { # create the file
    my $fh = new IO::File $filename , O_CREAT;
    undef $fh;
}
```

Listing 3

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

use Tk;

my $mw = new MainWindow;

my $textwidget = $mw -> Text
    -> pack (-expand => 1, -fill => 'both');

MainLoop;
```

Sean M. Burke "Turning HTML into an RSS Feed"

Listing 1

```
use LWP::Simple;
  my $content_url = 'http://freshair.npr.org/dayFA.cfm?todayDate=current';
  my $content = get($content_url);
  die "Can't get $content_url" unless defined $content;
  $content =~ s/(\cm\cj|\cj|\cm)/\n/g; # nativize newlines

  my @items;
  while($content =~ m{
  \s+<A HREF="([^"\s]+)">Listen to
  \s+<FONT FACE="Verdana, Charcoal, Sans Serif" COLOR="#ffffff" SIZE="3">
  \s+<B>(.*?)</B>
  }g) {
    my($url, $title) = ($1,$2);
    print "url: {$url}\ntitle: {$title}\n\n";
    push @items, $title, $url;
  }
```

Listing 2

```
use LWP::Simple;
  my $content_url = 'http://www.guardian.co.uk/worldlatest/';
  my $content = get($content_url);
  die "Can't get $content_url" unless defined $content;
  $content =~ s/(\cm\cj|\cj|\cm)/\n/g; # nativize newlines

  my @items;
  while($content =~
   m{<A HREF="(/worldlatest/.*?)">(.*?)</A><BR><B>.*?</B><P>}g
  ) {
    my($url, $title) = ($1,$2);
    print "url: {$url}\ntitle: {$title}\n\n";
    push @items, $title, $url;
  }
```

Listing 3

```
use HTML::Entities qw(decode_entities);

  sub xml_string {
    # Take an HTML string and return it as an XML text string

    local $_ = $_[0];
    # Collapse and trim whitespace
    s/\s+/ /g;  s/^ //s;  s/ $//s;

    # Delete any stray HTML tags
    s/<.*?>//g;

    decode_entities($_);

    # Substitute or strike out forbidden MSWin characters!
    tr/\x91-\x97/''""*\x2D/;
    tr/\x7F-\x9F/?/;

    # &-escape every potentially unsafe character
    s/([^ !#\$%\x28-\x3B=\x3F-\x7E])/'&#'.(ord($1)).';'/seg;

    return $_;
  }
```

Listing 4

```
sub rss_body {
    my $out = '';
    while(@_) {
      $out .= sprintf
       "  <item>\n\t<title>%s</title>\n\t<link>%s</link>\n  </item>\n",
       map xml_string($_),
           splice(@_,0,2); # get the first two each time
    }
    return $out;
  }
```

Listing 5

```
sub rss_start {
    return sprintf q[<?xml version="1.0"?>
  <!DOCTYPE rss PUBLIC "-//Netscape Communications//DTD RSS 0.91//EN"
    "http://my.netscape.com/publish/formats/rss-0.91.dtd">
  <rss version="0.91"><channel>
    <title>%s</title>
    <description>%s</description>
    <link>%s</link>
    <language>%s</language>
  ],
    map xml_string($_),
      @_[0,1,2,3];  # Call with: title, desc, URL, language!
  }
...and for the end:
  sub rss_end {
    return '</channel></rss>';
  }
```

Listing 6

```
print
   rss_start(
     "Guardian World Latest",
     "Latest Headlines from the Guardian",
     $content_url,
     'en-GB',  # language tag for UK English
   ),
   rss_body(@items),
   rss_end()
  ;
```

Listing 7

```
print
   rss_start(
     "Fresh Air",
     "Terry Gross's interview show on National Public Radio",
     $content_url,
     'en-US',  # language tag for US English
   ),
   rss_body(@items),
   rss_end()
  ;
```

brian d foy "Better Documentation Through Testing"

Listing 1

```
podchecker /usr/local/lib/perl5/site_perl/darwin/DBI.pm

*** WARNING: (section) in 'perl(1)' deprecated at line 4926 in file /usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: (section) in 'perlmod(1)' deprecated at line 4926 in file /usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: (section) in 'perlbook(1)' deprecated at line 4926 in file /usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** ERROR: unresolved internal link 'bind_column' at line 1819 in file /usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: multiple occurence of link target 'trace' at line - in file /usr/local/lib/perl5/site_perl/darwin/DBI.pm
*** WARNING: multiple occurence of link target 'Statement (string, read-only)' at line - in file /usr/local/lib/perl5/site_perl/darwin/DBI.pm
/usr/local/lib/perl5/site_perl/darwin/DBI.pm has 1 pod syntax error.
```

Listing 2

```
localhost_brian[3150]$ make test
cp ISBN.pm blib/lib/Business/ISBN.pm
cp Data.pm blib/lib/Business/ISBN/Data.pm
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/load.t t/pod.t t/isbn.t
t/load....ok
t/pod.....ok
t/isbn....ok 20/21
Checking ISBNs... (this may take a bit)
t/isbn....ok
All tests successful.
Files=3, Tests=25, 358 wallclock secs (116.90 cusr +  2.96 csys = 119.86 CPU)
```

Listing 3

```
localhost_brian[3152]$ make test
cp ISBN.pm blib/lib/Business/ISBN.pm
Skip blib/lib/Business/ISBN/Data.pm (unchanged)
PERL_DL_NONLAZY=1 /usr/bin/perl "-MExtUtils::Command::MM" "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/load.t t/pod.t t/isbn.t
t/load....ok
t/pod.....NOK 1#     Failed test (t/pod.t at line 12)
# Pod had errors in [blib/lib/Business/ISBN.pm]
# *** ERROR: =item without previous =over at line 400 in file blib/lib/Business/ISBN.pm
# blib/lib/Business/ISBN.pm has 1 pod syntax error.
t/pod.....ok 2/2# Looks like you failed 1 tests of 2.
t/pod.....dubious
        Test returned status 1 (wstat 256, 0x100)
DIED. FAILED test 1
        Failed 1/2 tests, 50.00% okay
t/isbn....ok 20/21
Checking ISBNs... (this may take a bit)
t/isbn....ok
Failed Test Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------------------------
t/pod.t        1   256     2    1  50.00%  1
Failed 1/3 test scripts, 66.67% okay. 1/25 subtests failed, 96.00% okay.
make: *** [test_dynamic] Error 35
```

Listing 4

```
sub _check_pod
    {
    my $file = shift;

    return { result => NO_FILE } unless -e $file;

    my %hash    = ();
    my $output;
    $hash{output} = \$output;
    my $checker = Pod::Checker->new();

    # i pass it a tied filehandle because i need to fool
    # Pod::Checker into thinking it is sending the errors
    # somewhere so it will count them for me.
    tie( *OUTPUT, 'IO::Scalar', $hash{output} );
    $checker->parse_from_file( $file, \*OUTPUT);

    $hash{ result } = do {
        $hash{errors}   = $checker->num_errors;
        $hash{warnings} = $checker->can('num_warnings') ?
            $checker->num_warnings : 0;

           if( $hash{errors} == -1  ) { NO_POD   }
        elsif( $hash{errors}   > 0  ) { ERRORS   }
        elsif( $hash{warnings} > 0  ) { WARNINGS }
        else                          { OK }
        };

    return \%hash;
    }
```

Listing 5

```
sub pod_ok
    {
    my $file     = shift;
    my $expected = shift || OK;
    my $name     = shift || "POD test for $file";

    my $hash = _check_pod( $file );

    my $status = $hash->{result};

    if( defined $expected and $expected eq $status )
        {
        $Test->ok( 1, $name );
        }
    elsif( $status == NO_FILE )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Did not find [$file]" );
        }
    elsif( $status == OK )
        {
        $Test->ok( 1, $name );
        }
    elsif( $status == ERRORS )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Pod had errors in [$file]\n",
            ${$hash->{output}} );
        }
    elsif( $status == WARNINGS and $expected == ERRORS )
        {
        $Test->ok( 1, $name );
        }
    elsif( $status == WARNINGS )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Pod had warnings in [$file]\n",
            ${$hash->{output}} );
        }
    elsif( $status == NO_POD )
        {
        $Test->ok( 0, $name );
        $Test->diag( "Found no pod in [$file]" );
        }
    else
        {
        $Test->ok( 0, $name );
        $Test->diag( "Mysterious failure for [$file]" );
        }
    }
```