The Perl Review

Volume 0 Issue 1 March 1, 2002

Past, Present, & Future

Web Access http://www.ThePerlReview.com/ **Email** letters@theperlreview.com **Publisher** brian d foy **Editors** David H. Adler, John D. Porter, Andy Lester **Technical Editors** Kurt Starsinic, Adam Turoff **Contributors** Sharon Hopkins, brian d foy, David H. Adler, Dan Sugalski, Jeff Bay, Dan Hoover, Jérôme Quelin, Buffy Summers **Copyright** Format Copyright © 2002 *The Perl Review*, article content Copyright © 2002 by their respective authors.

Letters to *TPR*

Across the pond

If possible, keep this magazine distributed electronically. I like the PDF distribution. Especially as I usually live in Germany and the shipping costs for US magazines are very high and the magazines arrive very late usually if at all.
– *Kai Jendrian, Germany*

brian writes: *We do not know everything that we are going to do in the future, but we are definitely thinking about our international friends. We can cut down on costs and shipping times by printing close to the destination, so we are investigating service bureaus in the UK and possibly elsewhere within the European Community. That still leaves out a great part of the world, but we have to take it one step at a time. If someone can help us print and ship issues in other parts of the world, we would love to hear from you.*

For beginners

I printed out The Perl Review 0,0 today after seeing the blurb on Slashdot, and I can say that, at least the parts I could fathom, I really enjoyed. I am *very* new to Perl, but found a good bit of the Review informative. My suggestion is this, could you please set aside an article each issue that dealt with beginning perl or beginner uses for perl?
– *Shannon Durham, Huntsville, AL*

brian writes: *Although I cannot promise that you will understand every issue, we will try to get in at least one article aimed at beginners in each issue. In this issue check out "Perl One-liners" by Jeff Bay.*

HTML

I'd like to have web access to the new magazine.
– *Names withheld*

brian writes: *You do have web access – or did you mean that you would like an HTML version?*

Several people said something similar, and some even suggested we were lost in the past since we were not using the web. We will probably also offer a delayed HTML version some time in the future, but we want to focus on a print version which means we spend our time laying out the content so that it prints nicely. Print, with its much higher resolution than computer screens, produces something much more beautiful to look at and easier to read. We are interested in the form as much as the content. For those of you that have not thought about this before, I highly recommend the books of Edward Tufte, or one of his seminars if he comes to a town near you
http://www.edwardtufte.com/

Besides, I like reading on the subway where sometimes holding my laptop can be quite precarious.

About this issue

This issue spans three major versions of Perl. The last article, "Camels & Needles" by Sharon Hopkins, takes us back to the days of Perl 4 as Sharon discusses Perl Poetry. *The Economist* published her poems, *listen*, in July 1995. In "Parroty Bits", Dan Sugalski discusses the new interpreter for Perl 6 and why we need something different than Perl 5. Jeff Bay presents a beginner's introduction to one-liners in his article "Perl One-liners" in response to many request for something for Perl novices. Let us know what you think by sending us email at letters@theperlreview.com.

Perl on the web

The Perl Review
http://www.theperlreview.com – the website for this magazine with information for readers and authors.

Perldoc.com
http://www.perldoc.com – online, searchable documentation for Perl versions 5.004 to 5.6.1.

CPAN Search
http://search.cpan.org – search for Perl modules or read the documentation without downloading the module.

Books in articles

Support *The Perl Review* by purchasing books through http://www.theperlreview.com/books.shtml.

Community News

Pair Networks sponsors TPR

http://www.pair.com
Pair Networks, a provider of web sites services, donated an account to *The Perl Review*. The home of this magazine is now http://www.theperlreview.com.

YAPC Call for Participation

http://www.yapc.org/America/cfp.shtml
YAPC::America, June 26-28 in St. Louis, MO, is accepting proposals for talks ranging from a five minute lightning talk to 3 hour tutorials. Any topic is welcome. The submission deadline is May 1.

YAPC::Europe

http://www.yapc.org/Europe/2002/index.html
Germany gets a second Perl event this year. The 2002 YAPC::Europe, hosted by the Munich Perl Mongers, will be at Technische Universität München September 18-20. Registration is now open, and the conference group is accepting proposals for papers and presentations. The conference fee is 89 euros.

The Perl Foundation

http://perl-foundation.org/
The Perl Foundation, an arm of Yet Another Society (YAS), announced that they will financially support Larry Wall with a 2002 Perl Development Grant. YAS also supports Damian Conway and Dan Sugalski.

New O'Reilly User Group Liaison

Marsee Henon takes Denise Oliffe's place as Denise takes a leave of absence

MacPerl v5.6.1b4

http://dev.macperl.org/
Chris Nandor says that the latest build of MacPerl may be the last beta release. MacPerl v5.8.0 is under development as well.

Perl Golf

Tim Gim Yee won the *The Perl Review* February golf challenge by submitting the first 33 stroke solution to convert a base 36 number to its base 10 representation. The base 36 number came from the command line and the output needed to be newline terminated. Tim used the strtol function from the POSIX module which converts numbers from bases 2 to 36 to a long integer, which Perl prints in base 10.

```
#!perl -l
use POSIX;print~~strtol pop,36
```

Besides the POSIX strtol function which performs most of the magic, Tim used the pop() function, the -l switch, and two bitwise negation operators in tandem, `~~`. The pop function grabs the last command line argument, saving two strokes over the conventional use of the shift function. The first ~ puts the strtol into scalar context and then bitwise negates the return value. The second ~ negates that negation returning the original value. The perlop manual page explains the details of the bitwise negation operator. The -l switch sets the output record separator to a newline, which saves two strokes over `"\n"`.

The most noteworthy solutions came from **Andrew Savige** who not only submitted a 33 stroke solution, but a 7918 stroke solution that printed an ASCII art version of *The Perl Review*. He also sent this one that works in three languages – Perl, C, or C++:

```
#include <stdio.h>
#include <stdlib.h>
#define strtol(a,b) strtol(a,0,b)
#define $ /*
use POSIX;$argv[1]=pop;"*/
main(int argc,char *argv[]) //";
{
printf("%ld\n",strtol($ argv[1],36));
}
```

The C comment, /* */, hides the Perl code from C. The double quotes hide the C code from Perl inside a string in void context. The C pre-processor directives start with the Perl comment character, #, and work as written in either language. From C, he defines $ as nothing so that in the printf function Perl sees `$ argv[1]`, but C sees `argv[1]`.

Congratulations to **Patrick Gaskill** for winning the beginner's category with a score of 51.

```
#!perl -l
$_=pop;$b=36*$b+(/\d/?$_:-55+ord)
for/./g;print$b
```

For solutions which did not use the POSIX module, **Karsten Sperling** wins with low 46.

```
map$.=36*$.-55+/\d/*7+ord,pop=~/./g;
print$..$/
```

Dave Hoover and Jérôme Quelin ran this golf challenge and volunteered to run more Perl Golf challenges and set up a SourceForge project to take care of the details. See the results of this challenge or take part in a new one at http://perlgolf.sourceforge.net/.

Complete results can be found on our website.
http://www.theperlreview.com/golf/golf.0.0.txt

Host with pair Networks

- Rock-Solid FreeBSD Unix Reliability
- Edit scripts and compile programs from a secure shell
- Technical, competent support
- Perl 5.6 with over 2000 installed modules
- Free Setup when you switch to us

We've got a web hosting plan waiting for you.

www.pair.com

pair networks

Perl One-liners

Jeff Bay, jlb0170@yahoo.com

Abstract

This article introduces some of the more common perl options found in command line programs, also known as one-liners. I cover the -e, -n, -p, -M, and -w switches, along with BEGIN and END blocks.

1 Creating a one-liner

Most talented unix Perl programmers I have met have a dirty little secret. They cannot resist the allure of the gnarly Perl one-liner for accomplishing short tasks that do not need a complete script.

The -e switch allows me to write Perl scripts directly on the command line. Code listing 1 shows a simple “Hello world”.

Code Listing 1: Hello World

```
prompt$ perl -e 'print "hello world!\n"'
hello world!
```

In code listing 2, something a bit more complex, I take the output from ls, parse it for the file size, and sum the sizes for all files which are not directories.

Code Listing 2: File size sum

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^[dt]/; $sum += (split)[4] } print "$sum\n"'
1185
```

I use several tricks in code listing 2. Normally, I do not write something like this all at once. I build it up a bit at a time to make sure I get the correct output at each step. In code listing 3 I check the output of the command.

Code Listing 3: ls output

```
prompt$ ls -lAF
total 32
drwxrwsr-x   2 jbay     staff        512 Feb 21 09:34 adir/
-rw-rw-r--   1 jbay     staff        395 Feb 21 09:29 afile1
-rw-rw-r--   1 jbay     staff        423 Feb 21 09:29 afile2
-rw-rw-r--   1 jbay     staff        120 Feb 21 09:29 afile3
```

In code listing 4 the output from `ls` becomes the standard input to my script, which simply prints each line. I can see that I get the output I expect, the same thing from code listing 3.

Code Listing 4: ls piped to perl

```
prompt$ ls -lAF | perl -e 'while (<>) { print $_ }'
total 32
drwxrwsr-x   2 jbay     staff        512 Feb 21 09:34 adir/
-rw-rw-r--   1 jbay     staff        395 Feb 21 09:29 afile1
-rw-rw-r--   1 jbay     staff        423 Feb 21 09:29 afile2
-rw-rw-r--   1 jbay     staff        120 Feb 21 09:29 afile3
```

In code listing 5 I want to skip the initial total line and directories, so I want to ignore lines that begin with a "d" or "'t". I add a next before the print statements to skip lines that begin with a "d" or a "t", so my program does not print them.

Code Listing 5: Skip lines

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^[dt]/; print $_; }'
-rw-rw-r--   1 jbay     staff        395 Feb 21 09:29 afile1
-rw-rw-r--   1 jbay     staff        423 Feb 21 09:29 afile2
-rw-rw-r--   1 jbay     staff        120 Feb 21 09:29 afile3
```

Once I know that my script skips the right lines, I split the remaining lines and print the value in column 5, the file size. At this point my program, in code listing 6, prints out the same number I see in the ls output, which is the correct size for each file.

Code Listing 6: Print file sizes

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^[dt]/; print +(split)[4], "\n" } '
395
423
120
```

Finally, I want to sum the file sizes. Code listing 7 accumulates the sum in $sum, then prints it at the end of the program.

Code Listing 7: Sum file sizes

```
prompt$ ls -lAF | perl -e 'while (<>) { next if /^[dt]/; $sum += (split)[4] } print "$sum\n"'
938
```

Now I have the complex perl one-liner that I showed in code listing 2.

2 One-liner input

Perl programs can receive data from standard input or the command line arguments in @ARGV.

2.1 Standard input

Code Listing 8: Skipping comments

```
prompt$ cat afile | perl -e 'while (<>) { print unless /\s+#/ }'
```

The "|" (pipe) symbol takes the output of cat and makes it the standard input of my Perl program. The diamond operator, <>, reads lines from standard input, so this one-liner reads the lines from afile and then prints the lines that do not match the regular expression `\s+#`.

I can also redirect file contents to perl's standard input using the shell redirection operator, <. Code listing 9 produces the same output as the previous example.

Code Listing 9: Input by redirection

```
prompt$ perl -e 'while (<>) { print unless /\s+#/ }' < afile
```

However, the diamond operator can open and directly read the contents of the file specified on the command line so I do not need to redirect the file contents myself. Code listing 10 does not use file redirection, and does the same thing as code listing 9.

Code Listing 10: Input

```
prompt$ perl -e 'while (<>) { print unless /\s+#/ }' afile
```

2.2 Command line arguments

I can access command line arguments using @ARGV. Code listing 11 simply prints whatever is in @ARGV.

Code Listing 11: Print the command line arguments

```
prompt$ perl -e 'print "@ARGV\n"' Foo Bar Bletch
Foo Bar Bletch
```

Suppose I have a file that contains a list of files, one filename per line, that I want to manipulate. I can see the file names when I list the files in code listing 12.

Code Listing 12: The filenames in files.txt

```
prompt$ cat files.txt
afile1
afile2
afile3
```

The unix utility xargs can transpose its standard input into arguments for another command. I want to take this list of filenames and make them the arguments of the wc command so I can count the number of lines in each file. In code listing 13 the xargs command takes its standard input, the list of filenames, and makes them the arguments for wc.

Code Listing 13: Count lines in files

```
prompt$ cat files.txt | xargs wc -l
          54 afile1
          54 afile2
          54 afile3
         162 total
```

Code listing 13 is the same as if I typed this directly, as in code listing 14.

Code Listing 14: Count lines in files

```
prompt$ wc -l afile1 afile2 afile3
```

2.3 Playing with find

In code listing 15 I reimplement the find command option "-type d" using a perl one-liner and xargs. The find command recursively outputs a list of filenames starting from a specified directory and matching certain criteria. In this case, the criteria, "-type d", lists only directories.

Code Listing 15: Using find

```
prompt$ find . | xargs perl -e '@ARGV = grep( -d $_ , @ARGV); print "@ARGV"'
```

The xargs command takes the list of filenames and makes them the arguments to the one-liner in code listing 15. The one-liner then uses a grep expression to filter @ARGV for filenames that are directories using the -d file test operator and then prints the results.

3 Useful command line switches

Perl command line options shorten one-liners by adding automatic processing to the small script I create using the -e option. Perl has many other useful options besides the ones I show. See the perlrun manual page for the details.

3.1 The -e switch

The perl interpreter takes each -e argument as a fragment of Perl code and executes it. Each -e switch on the command line is taken as a line in a script. If I paste the contents of each -e switch into a file, and run Perl on that file, I have the exact same effect as the -e switch. Code listing 16 rewrites code listing 1 with two -e switches.

Code Listing 16: Multiple -e switches

```
prompt$ perl -e 'print "Hello ";' -e 'print "world\n";'
Hello world
```

Each code bit (in the outer single quotes) is a single string that the shell parses as a separate token, so the shell sees the four tokens in code listing 17.

Code Listing 17: Multiple -e switches, as tokens

```
-e
print "Hello ";
-e
print "world\n";
```

3.2 The -n switch

The -n switch wraps a while loop around your program. In code listing 18, the loop reads lines of input with the diamond operator, sets $_ to the contents of each line, then executes the code bits I specify with the -e switch.

Code Listing 18: Using -n

```
while (<>) {
  <-e argument>
  <-e argument>
}
```

In code listing 19 I create my own cat program.

Code Listing 19: Reimplementing cat

```
        prompt$ perl -ne 'print $_' afile
```

3.3 The -p switch

The -p switch does the same thing and prints the value of $_ at the end of each iteration.

Code Listing 20: Using -p

```
while (<>) {
  <-e argument>
  <-e argument>
  print;
}
```

In code listing 20 the loop reads lines in standard input, sets $_ to the contents of each line, executes the -e args, and then prints $_. I can use this to modify lines from an output listing.

For example, I can remove the permissions column on a "ls -l" output file listing. In code listing 21 I substitute the first group of non-whitespace and the space after it with nothing.

Code Listing 21: Remove the first column

```
prompt$ ls -l | perl -pe 's/\S+ //'
```

3.4 Using modules

I can use modules on the command line with the -M switch. The -M<module> switch is the equivalent of including "use <module>;" in the virtual scripts I create. In code listing 22, I use the IO::Handle module to set the standard output autoflush option.

Code Listing 22: Using modules

```
prompt$ cat afile | perl -MIO::Handle -e 'STDOUT->autoflush(1); while (<>) { print }'
```

Normally, I use strict and turn on warnings in my scripts and I can do this in one-liners as well. In code listing 23 I include the strict module with -Mstrict, and turn on warnings by adding -w.

Code Listing 23: Using strict

```
prompt$ cat afile | perl -w -Mstrict -e 'my $var = 17; print $var'
```

If I do not declare $var, the strict module catches it as it does in code listing 24.

Code Listing 24: Undeclared variables

```
prompt$ cat afile | perl -w -Mstrict -e '$var = 17; print $var'
Global symbol "$var" requires explicit package name at -e line 1.
Execution of -e aborted due to compilation errors.
```

In code listing 25, Perl warns about $var which I used without initializing it.

Code Listing 25: Uninitialized variables

```
cat afile | perl -w -Mstrict -e 'my $var; print $var'
Use of uninitialized value at -e line 1.
```

4 Wrestling with the shell

Quote marks, double and single, as well as the dollar sign, are part of the shell syntax. If I need to use these characters in my string, I must escape them. Each shell has a slightly different syntax for its special characters, and different platforms may handle escaping them differently. Code listing 26 shows several examples of escaping shell metacharacters.

Code Listing 26: Escaping shell metacharacters

```
prompt$ echo "the variable \$USER is "\""$USER"\"" "
the variable $USER is "jbay"
```

I can do several things to avoid shell quoting problems. In code listing 27, the program outputs a malformed SQL statement because the literal a3 is not quoted. The single quotes disappear because I used single quotes for my -e code bit, but I need quotes around 'a3' so the SQL parser knows that a3 is a literal string and not a column name.

Code Listing 27: Misquoted SQL

```
prompt$ perl -e 'print "select * from foo where bar='a3'\n"'
select * from foo where bar = a3
```

In code lisitng 28 I use Perl's chr() function to add any literal character (including quotes) using its ordinal ascii value. I can concatenate chr(39), the single quote, with the rest of the SQL string.

Code Listing 28: Using chr() to get literal values

```
prompt$ perl -e 'print "select * from foo where bar=" . chr(39) . "a3" . chr(39) . "\n"'
select * from foo where bar='a3'
```

In code listing 29 I use the generalized quote operators, q and qq, instead of single and double quotes. I can use single or double ticks inside the resulting perl string because they are no longer delimiters.

Code Listing 29: Generalized quotes

```
prompt$ perl -e 'print qq#select * from foo where bar="a3"\n#'
select * from foo where bar="a3"
```

Code listing 30 uses the back-whack character,"\", to escape the quote marks, but the syntax is rather unwieldy – `'\''`. In most shells, I have to close the prior string with the first tick, then put in the literal tick with a back-whack , and then start the next string with a third tick. The shell then concatenates those strings into a single string before it executes them.

Code Listing 30: Escaping quote characters

```
prompt$ perl -e 'print "select count(*) from foo where bar ='\''a3'\''\n"'
select * from foo where bar ='a3'
```

5 Start and End tricks

I can execute code before or after my -e program with the BEGIN or END keyword respectively. The END block in code listing 31 prints the sum after the while loop finishes.

Code Listing 31: END block

```
ls -lAF | perl -ne 'next if /^d/; $sum += (split)[4]; END{ print "$sum\n" }'
```

In code listing 32 the BEGIN executes its block before an implicit loop starts. I can initialize the variable $sum to 1024 before the loop begins if I use a BEGIN block.

Code Listing 32: BEGIN block

```
ls -lAF | perl -ne 'BEGIN{$sum=1024} next if /^d/; $sum += (split)[4]; END{ print "$sum\n" }'
```

6 References

Chapter 6, "Social Engineering, Cooperating with Command Interpreters", *Programming Perl* – Larry Wall, Tom Christiansen, & Jon Orwant.

The following perl manual pages come with the standard Perl distribution and can be found online at Perldoc.com, http://www.perldoc.com, or from the command line with "perldoc pagename".

- perlrun - perl interpreter options
- perlfaq3 "Why don't Perl one-liners work on my DOS/Mac/VMS system?"

The unix manual pages may be found online at several sites, including http://www.bsdi.com/bsdi-man/, or from the command line with "man pagename".

- find
- wc
- xargs

Extreme Publishing: Postmortems

brian d foy

Abstract

The tenets of Extreme Programming include user stories, short release cycles, and story postmortems. I illustrate these self-referentially by applying to them to how we publish *The Perl Review* but you can apply them to almost any project on which you might work.

1 Project velocity

Extreme programming (XP) identifies parts of the project in terms of "user stories" that describe what the users, in this case *The Perl Review* readers, authors, and editors, need from the project and how they interact with the product. A user story tells the tale of a particular person's interaction with whatever the project produces.

The XP process emphasizes short release cycles and frequent testing so that the development team knows the health of the project on a day-to-day basis, rather than wondering if the system will work when they reach their delivery deadline. Furthermore, the short release cycle allows the users to constantly evaluate if the product meets their user story requirements.

This cycle of daily, or shorter, testing and integration allows the team to constantly evaluate the difficulties involved with the user stories and adjust the project schedule to meet reality. No matter what the project diagrams show or what management might say, project tasks take a certain amount of time. Denying that fact just makes matters worse at delivery time, and does not improve the situation during the next cycle.

The "project velocity" measures how fast these tasks progress. In the last issue of *The Perl Review*, I wrote out my user stories for the production of the journal and made estimates about how long each part would take. I thought that producing the PDF document would take the most time, and that receiving articles from authors would be relatively easy. It turns out that it is actually the other way around. Had I not taken time to think about the differences in my estimates and reality, I would have a more difficult time planning the next issue, and may even make the same mistakes again.

Once I finished the basic research about the things I needed to do with LaTeX, I discovered that a lot of people already had the same idea so I could reuse packages I found on the Comprehensive TeXArchive Network, whose idea inspired the creation of the Comprehensive Perl Archive Network (CPAN). The Google Groups website (http://groups.google.com) saved me hours of work. When I went back to my user stories that required LaTeX work, I gave them a high project velocity. Some things that I estimated a week to complete I finished in an afternoon. Indeed, for this issue, I spent less than an hour with new LaTeXwork, and I knew that I would.

Dealing with authors, however, took me much longer than I expected. Naturally, as any manager does, I failed to account for people's lives outside of *The Perl Review*, which often demands much more of them whether it is their day job or their newborn daughter. The project velocity for submissions is slow, and I need to take that into account in my planning. I need to start the article submission process much sooner than the production process. I need to solicit and secure articles a couple of issues ahead of publication time to keep things moving since the project velocity for submissions is actually longer than the production cycle for my first two issues.

2 Story postmortem

Once the team finishes a particular user story and has had time to not think about it (a cooling-off period), they can evaluate how well they did on the story. Part of that measures the project velocity, but the postmortem gives the team a chance to recognize other strengths or areas for improvement. Some people focus on the weaknesses, but if a team can identify strengths they can emphasize those in the next iteration.

Post-project evaluations occur in many different institutions, including the military. One of the most surprising books I have read is *Business is Combat* written by an United States Air Force fighter jet pilot. In it, James Murphy explained about how everything he does as a fighter pilot is about process and very little about flying, and applies this to the business cycle. No matter how tired he is or how long he was in the air his crew chief interviews him after each flight and he goes to the mission debriefing. Since the mission post-brief happens after absolutely every mission, the grumblings are virtually non-existent.

As software developers, if we decide on a process, and stick to the process, once we get used to it, its just part of the normal work day. In XP these things happen in short cycles, rather than big event quarterly meetings, so the postmortem process can take 30 minutes a week. A half hour should not kill the project schedule, and if done correctly saves that much more in the future.

We do not have this sort of thing as an industry-wide mind set the way professions like law and medicine, or in this case, the military does. Individual enterprises may have internal standards, but in my experience the book goes out the window at crunch time. You cannot improve things if you do not take the time to improve them.

In the United States Army, which makes me follow this process while on active duty, after any training or real-life mission the team conducts an after action review (AAR), which the Army explains extensively in its field manuals. The Army, as well as the other branches of service, has a manual for everything. The US Army has been around for 226 years and had a manual, the Blue Book drill and ceremony manual, before the United States of America won its independence.

The key to the AAR is improvement at both the team and individual levels. The AAR leader starts the session by asking the team what they meant to accomplish, which keeps first things first. The team does most of the talking while the leader maintains the process. From there, the team identifies what actually happened and how that affected the attainment of the goal. The team identifies strengths, which they should maintain, and areas for improvement. The language is important; we made two lists – “Maintain” and “Improve”, not “Success” and “Failure”. Focus on the positive to move things forward.

Our goal with the last issue was to publish a Perl magazine in PDF format. Out of all of the things that we did and some of the things that we did not do, we met our goal. We developed a layout, learned quite a bit of LaTeX, and published some articles. We list those in the “Maintain” list. We cannot forget about those because past performance does guarantee future success. We need to get articles sooner and edit them better. We list those in the “Improve” list. I discuss some of these in the rest of the article.

The postmortem can be as simple as that, especially with a good team that has worked together for a while. The postmortem becomes a fluid part of the process rather than something to schedule or tolerate.

3 Stories for this milestone

In the last issue I mentioned several stories that I thought might be important for this issue, and I solicited comments for the community about how important they might actually be to readers.

3.1 Produce an A4 layout

I asked a couple of people across the Pond how well the US-Letter layout of issue printed on A4 printers. Despite some extra white space, since A4 is somewhat larger than US-Letter, no one complained. If people start to complain about the US-Letter format, I might look at this user story again, but I probably will not have to do that once we get to the print version. I am still thinking about our international readers, though. I would like to produce the print version as close to the regional markets as possible, so a couple of people volunteered to research service bureaus in the European Union. Other markets need other volunteers.

3.2 Sell some ads to cover production costs

We still need to work on this, but besides labor costs, which has so far been volunteer work and nominal printing costs (although when you print it at work it's free), our production costs are very low. I have been researching service bureaus, and when we need to move to print we will have to pay real money to vendors. Once we get that far, we will want a commission-based sales manager to handle that for us. Anyone interested in doing that? I should also mention that Neil Bauman of GeekCruises (http://www.geekcruises.com) has been very helpful with suggestions about the business and production aspects of magazine publication.

3.3 Collect subscribers for future issues

Many readers wrote in to add their name to a list of future subscribers. I still do not know when a print issue will be available, but I do have a better way to collect subscribers. You can visit our web site to add your name to our list of future subscribers (http://www.theperlreview.com/subscribe.shtml).

3.4 Include a book review column

We had a book review in the last issue, and we do not have one in this issue, although we have three that we could have published. We were going to publish a review of *Perl Debugged*, but we want to save that for an upcoming issue we hope to devote to debugging. Some publishers sent me some review copies, and a couple of people have already submitted reviews. You can send us your own reviews (http://www.theperlreview.com/book_reviews.html). Publishers can also send review copies of their books to me (publisher@theperlreview.com).

3.5 Include an ongoing Perl Golf column

A couple of people got really excited about conducting Perl golf competitions, and we received a couple hundred entries. Dave Hoover and Jerome Quelin volunteered to run the competitions and have set up a SourceForge project for things that support that. (http://perlgolf.sourceforge.com). The results of last month's golf challenge appears in this issue.

4 Stories for the next iteration

4.1 Author submission process

Authors need to know what to do and what happens during the editorial process, and the editors need to know how to move that process along. I need to figure out the process because the way I do it now takes too long and causes too much confusion. Since I identified this as the story with the lowest project velocity, I think a lot of my time in the next publication cycle, especially early in the cycle, goes towards making this situation easier.

This is a big task, however. When you encounter big tasks in XP, you create smaller tasks.

4.1.1 Define the process

I need to get together with the editors to codify how we do things, then publish that in the authors guide on the web site. Parts of the process include defining the intended writing style and laying out the technical and editorial review process. All of this starts when an author submits an article proposal from our website.

4.1.2 Let the authors know the status of their article

We are not full time editors, so we do not spend our full-time job interacting with authors. The authors should be able to check the status of their articles on our website. Editors can do this already, so we just need to go a couple of extra steps to turn that around to an author's perspective.

4.1.3 Assign editors to authors

I did not really have a good way to do this, and I did it pretty miserably this time around. Authors should get comments from one person who shepherds the article to publication. The author should know who his editor is, and the editors should know who their authors are. This should move the article along a consistent path rather than several editors trying to turn it into their conception of an article.

4.2 Get some ads

I need to figure out the advertising process. If this magazine makes it to print, advertising revenues can cover some of the operating expenses which means lower subscription fees. The ad sales process is well known and the people who buy ads know what needs to be done, but including ads in the magazine can be tricky. Ad placement, size, and scheduling issues concern me more, so I want to make the mistakes when I can fix them (by regenerating the PDF file) instead of when I have to give the money back (when I mess up in print).

5 Conclusion

I discussed a couple of points involved in the XP process: the project velocity and the postmortem. The project velocity helps us manage time by identifying long-lived tasks, while the postmortem evaluates tasks and identifies areas for improvement. Each of these help us do better during the next iteration. I illustrated these with examples from the publication process for this magazine by initially focusing on what we wanted to accomplish, what we did right, and how we could improve.

6 References

Business is Combat: A Fighter Pilot's Guide to Winning in Modern Business Warfare, James D. Murphy, Regan Books, March 2000.

You can read more about XP in these fine books:

Extreme Programming Explained, Giancarlo Succi & Michele Marchesi, Addison Wesley, 2001.

Extreme Programming Installed, Ron Jeffries, Ann Anderson, Chet Hendrickson, & Kent Beck, Addison-Wesley, October 2000.

Extreme Programming Examined, Giancarlo Succi & Michele Marchesi, Addison Wesley, 2001.

Parroty Bits: Bit 0, The Beginning

Dan Sugalski, dan@sidhe.org

Abstract

I take you on a whirlwind tour of the parrot project, the new interpreter engine that Perl 6 will use. I discuss some of the motivations behind designing a new interpreter engine, and some of the ways that Parrot fixes Perl 5's current limitations.

1 What is Parrot?

Parrot is the interpreter engine behind perl 6. but it is not Perl 6. Perl 6 is a language, the revision of perl that Larry Wall is designing. Parrot, on the other hand, is an interpreter system. They are separate for this version of perl, and their development proceeds separately.

Parrot is object-oriented, dynamically typed, threaded, scalable, platform-independent, and language agnostic. On some platforms, notably x86 and Alpha Linux, x86 and Alpha *BSD, and SPARC Solaris, it has a Just In Time (JIT) compiler.

The name "Parrot" comes from the elaborate 2001 April Fool's joke perpetrated on an unsuspecting world by Simon Cozens and a horde of minions. The joke said that Larry and Guido van Rossum, the creator of Python, got together to design a joint language called Parrot. While it was originally a joke, it set Simon's fertile brain in motion. As so often happens, Life imitates Satire, and Parrot was born.

I want to emphasize that Parrot is language neutral. While Parrot must run Perl 6 code, we designed it to run code from other languages, such as Python and Ruby, that are similar to Perl. Not only does this make Parrot a more capable engine, but it also means we can potentially run code written for other languages. Perl 5 already lets you use libraries of C and C++ code, now Parrot can let you use Python and Ruby libraries in your Perl 6 code.

2 Why Parrot?

We started work a new engine because all the ones available to us now do not cut it in one form or another. The perl 5 engine is over seven years old, and has been extended and expanded well beyond its original design, and it shows. Java's JVM is geared towards statically typed languages – perl is weakly typed and late binding. The .NET framework limits us to the platforms that .NET runs on, ruling out many of the current platforms such as the Cray or IBM's big iron systems. Since nothing else meets our needs – not even Scheme – we designed parrot.

Separating the language and interpreter design gives us a number of advantages. Larry Wall, Perl 6's language designer, is free to spend all his time designing the language and leave the implementation to other people. This is a big help – designing a language the size of Perl 6 is a big task. Designing the language, designing the interpreter, and implementing the interpreter is too big a task for one person.

We can work on the interpreter engine before Larry finishes the language specification, since many of the details of the language do not affect the core design. The interpreter only needs to know the semantics of the language, not the syntax – Parrot needs to know what things Perl 6 does, not how it looks.

A separate interpreter lets us cast a wider net than just Perl. Many of the dynamic languages, including Python and Ruby, have semantics similar to perl. They do pretty much the same stuff perl does and in the same way. Even though they look different, they act the same. Method calls are method calls, and the interpreter does not care whether you use curlies or white space to denote blocks.

3 Parrot's goals

Parrot's design goals are simple.

- Execute programs quickly
- Present a clean extension mechanism
- Present a clean embedding mechanism
- Last for ten years before needing another overhaul

Execution speed is the most important – Parrot is not much use if it is slower than Perl 5. While it is too early to do extensive benchmarks, in our tests Parrot is three times as fast as perl 5, and an optimized version of the test runs about 10 times as fast. That lead will shrink with more substantial benchmarks, but we can readily achieve a target of 20% faster than Perl 5 without counting the JIT or Parrot-to-C compiler, of course, which would be cheating.

The clean extension and embedding interfaces will come as a huge relief to anyone who has had to deal with XS or SWIG. Perl's current extension and embedding interfaces were something of an afterthought, and uncharted territory at the time. It shows.

Longevity, the last goal, is the trickiest, as we have no real way to tell what the future will bring. All we can be certain of is that it will bring things we have not considered, so we designed Parrot's to be as flexible as it can be.

You might be surprised that speed is more important than extendibility. Indeed, the current trend in languages is to design things that are clean and pretty inside, and sweep performance issues under the rug until Moore's Law cleans them up. That is foolish since existing hardware is not getting any faster.

4 Parrot parts

The Parrot interpreter system is broken into four different parts.

4.1 The Parser

The parser takes raw source, whether it is Perl, Python, Ruby, or Scheme, and turns it into an Abstract Syntax Tree, or AST. An AST is a symbolic representation of your program that the compiler and optimizer can manipulate, though not yet anything the interpreter can execute.

4.2 The Compiler

The compiler takes an AST from the parser and translates it into something suitable for the Parrot interpreter to execute. The translation is straightforward, and the target does not have to be the Parrot interpreter. We can, and will, have compiler modules that target .NET and the JVM.

Targeting multiple back ends means your programs are not forced to run on the Parrot interpreter, and provides more cross-language compatibility. You could write part of your Java or C# programs in Perl 6, for example.

We do not throw out the AST after the compiler is done with it. The optimizer and debugger can make good use of it, and you never know when it might come in handy later on.

4.3 The Optimizer

The optimizer takes the bytecode the compiler module produces, along with the AST it was generated from, and makes the bytecode more efficient. Writing an optimizer for Perl is tricky, since Perl's dynamic nature makes an awful lot of things un-optimizable. Many of the classic optimizations assume that data is static, and its behavior is known at compile time, which is definitely not the case with Perl. We can still do things like constant folding (replacing expressions like "5 + 5" with 10) though.

Perl 5's optimizer is hobbled by a few things that we avoid with Parrot. Perl 5 has an integrated compile and run cycle with no way to save the resulting compiled program to disk. There is no point in spending 30 seconds to optimize a program if the optimizations save only 5 seconds.

Parrot, on the other hand, can load bytecode from disk, skipping the compilation phase entirely. While it still may take 30 seconds to optimize your program, and still save only 5 seconds per run, when you save it to disk you only need to run it 7 times to make the time spent worthwhile. When running a program from source, the way Perl 5 does, we will use a minimal set of optimizations by default.

Perl 5's optimizer assumes that programs are small. A few dozen lines of code and a couple of seconds to run through some external data file. There is little point in optimizing a program like that.

We know that modern programs are much larger than that, and might run for hours or days. Thirty seconds or a minute of extra startup time is well worth shaving even 1% off a program that runs for eight hours. Parrot lets you enable full optimizations even with compile-and-go style programs, though that will not be the default.

Finally, you can add type specifiers, such as "Int", to variables in Perl 6, which gives the optimizer more hints as to what your program does so it can optimize better.

4.4 The Interpreter

This is where the really interesting things happen. The interpreter module takes bytecode, either from the compiler/optimizer or loaded from disk, and executes it. Most of our development effort is focused here.

Parrot's interpreter is a register-based design that resembles a hardware CPU more than a traditional interpreter. The core design is predicated on the idea that memory is plentiful but using it is slow, CPU pipelines are deep, and L1 caches are reasonably large. This simplifies code generation and JIT compilation and optimizes better than the traditional stack-based approach. we will cover it in detail in later articles.

The core variable type in the interpreter is a Parrot Magic Cookie, or PMC, which I will discuss in much more detail in another article. We designed PMCs assuming that programs will tie them, share them between interpreters, or overload their operations. We made sure that the cost of overloading operators, thread-safing variable access, or calling tied methods or subroutines would only be paid by the variables that are actually overloaded, shared, or tied.

With perl 5, every time your program does anything with a variable – read from it, write to it, do any sort of math operation with it, turn it to a string, integer, or float – the interpreter has to first check if there is custom code attached to the variable. That is a lot of time spent checking, and wasted time at that, since most programs have a vanishingly small number of tied or overloaded variables.

The interpreter also has support for more complex programming tools, such as coroutines, continuations, and exceptions, as well as threading support built in from the ground up.

5 What works today

The interpreter engine is currently at about the level of your average CPU (about equivalent to an Alpha or PPC CPU). We can do basic math and string operations, flow control, input/output, and stack operations. The core of a regular expression engine is also functional. It is Turing complete, for those folks who care about such things. Operations on simple types – strings, integers, and floating point numbers – work, as do operations on more complex variable types.

The bytecode loader, interpreter core, and assembler work. We can write Parrot programs in a sort of high-level assembly language, turn those programs into bytecode and store it on disk, load that bytecode off disk, and execute it.

Normal perl scalars work as they should. You can create a string variable, turn it into a number to do math with it, then turn it back to a string for printing.

Arrays and hashes mostly work. We can store integers, floats, and strings into arrays and hashes, and we can fetch them back out again. By the time you see this, we should be able to store perl variables in hashes and arrays and fetch them back out again.

We have several small language parsers, including a Perl subset and a Scheme parser. We also have Jako, a new C-style language being developed in conjunction with the interpreter. Our miniperl so far handles simple control structures, most math operations, and scalar variables.

Part of the garbage collection and memory allocation system works. We can allocate memory and we know where its being used.

6 What works soon

The garbage collector does not yet find or collect garbage.

We do not yet have either lexical or global variables you can look up by name. This is waiting on functional hashes – once we have those, globals and lexicals are simple.

Methods and subroutines do not yet work

The parser, compiler, and optimizer are all written in perl, and need perl 5 to work.

7 Where can I get more information?

You can check out the parrot source through anonymous CVS:

```
cvs -d :pserver:anonymous@cvs.perl.org:/cvs/public login
[press enter at the password prompt]
cvs -d :pserver:anonymous@cvs.perl.org:/cvs/public co parrot
```

You can also get a snapshot which we generate every six hours from the CVS repository. Download the snapshot from http://cvs.perl.org/snapshots/parrot/parrot-latest.tar.gz. The source should build on most Unices and Windows (both native and in CygWin) and, if we are lucky, VMS.

The Perl 6 internals list is the canonical place for Parrot development. Send mail to perl6-internals-subscribe@perl.org to subscribe. Additional information's on the web at www.parrotcode.org and dev.perl.org, and you are welcome to drop by #parrot on IRC, on irc.rhizomatic.net.

And, of course, There is always next issue.

The Singleton Design Pattern

brian d foy

Abstract

The Singleton design pattern allows many parts of a program to share a single resource without having to work out the details of the sharing themselves. This article discusses this design pattern, shows several possible implementations, and highlights some Perl modules which use singletons.

1 About design patterns

Design patterns help structure code, although they do not say anything about how to implement it. They solve problems before we start to write code because they affect the design of programs by recognizing the possible abstractions in the problem. More than one pattern may apply, and within any pattern, many ways to implement it.

We typically represent patterns with modules so their implementations are re-usable and abstract and make full use of encapsulation. The rest of the program does not need to know how the module works, and indeed, the more the program knows about the module's workings, the more of a problem we have. Patterns promote loose coupling so that their implementation does not affect the rest program.

2 When to consider using a singleton

Some things, like a particular hardware interface or serial port, or a software construct such as a database connection, can or should only be used once in a program.

Parts of a program which use those resources may not need to know if other parts of the program are already using them, or if those resources have already been used or have yet to be initialized. The small part of my program that decides it needs to use that resource simply uses it. The singleton pattern works out the sharing so that I do not need to do so every time I want to share something, and so that I have one place to maintain the code that works out the sharing.

I may consider using a singleton class if my program design shows any of these symptoms:

- Global variables pass data between parts of the application.
- A resource can only be used once
- Several identical instances in different parts of the program represent the same thing

3 About the pattern

The Singleton pattern provides a single point of access to a particular instance, and a single point of maintenance. If I decide to change the behavior of my class that implements the singleton, perhaps by limiting the total number of uses of the singleton instance, or even re-implementing the class to allow more than one, but still a limited number of, instances, I only need to modify the class. Every program or point in the program that uses the singleton will immediately get the benefits of change without knowing the details.

4 A simple example

To implement the singleton, I need to know if I have already created the instance, and if so, use it. Otherwise I need to create and remember an instance somehow so I can return it the next time my program needs it.

In code listing 1 I create a very simple example of a global counter. My program which uses this module should be able to access the counter instance to increase or inspect its value, and different parts of the program will be able to do this without knowing about each other. If one part of the program adds to the counter, the other part of the program using the counter sees the new value, and vice versa.

Code Listing 1: Counter.pm

```perl
package Counter;

my $singleton = undef;

sub new
    {
    my( $class ) = @_;

    return $singleton if defined $singleton;

    my $self = 0;
    $singleton = bless \$self, $class;

    return $singleton;
    }

sub value
    {
    my $self = shift;
    return $$self;
    }

sub increment
    {
    my $self = shift;
    return ++$$self;
    }

1;
```

In line 3, I declare a lexical variable, $singleton, in which I store my single instance, which the entire Counter package can access, as long as the entire package is in the same file since $singleton is scoped to the file. I initialize it to `undef` so that I know I have not created the instance yet. If it is something other than `undef`, I know it contains an instance. Now I know how to check whether or not I have created the instance – if $singleton is defined, then it contains an instance. Furthermore, I scoped $singleton to my file with my(), so code outside of the file cannot affect it. Indeed, code outside of the class does not even know it exists. In other languages, $singleton might be called a private class variable.

The new() method performs all of my magic. In line 9 I test $singleton to see if it is defined, and if it is I simply return its value. If it is not defined, I have not yet created an instance, so I go through the rest of the method to create the instance and to store it in $singleton.

Two other functions, value() and increment(), let me inspect and increase the counter by one, respectively.

When I use this module in a program, I do not have to do anything special or know anything about how it is implemented. All I have to know is that no matter how many times I try to get an instance by calling new(), I get the same counter, even if I do not know that is what I call a "singleton". Code listing 2 shows a short example of the use of my Counter module.

Code Listing 2: counter.pl

```
 1  #!/usr/bin/perl
 2
 3  use Counter;
 4
 5  my $count  = Counter->new();
 6  my $count2 = Counter->new();
 7
 8  print "The counters are the same!\n" if $count eq $count2;
 9
10  print "Count is now ", $count->increment, "\n";
11  print "Count is now ", $count2->increment, "\n";
12  print "Count is now ", $count->increment, "\n";
13
14  print "Count is now ", $count2->value, "\n";
15
16  print "The counters are the same!\n"
17      if $count->value eq $count2->value;
```

I create two counters in lines 5 and 6, although these are really the same instance. In line 8 I compare the two references just to prove to myself that I have really created a singleton. When you write your own test suite, you need to check to make sure your singletons are really singletons.

Once I know that $count and $count2 are the same, I increment the value through $count and print the result, then do it again with $count2. No matter which one I use, the counter increases by one.

Output for Code Listing 2

```
The counters are the same!
Count is now 1
Count is now 2
Count is now 3
The counters are the same!
```

4.1 Using only one database connection

Suppose that I need to access a database server from several different parts of my program – perhaps to load configuration information when it starts, select some fields based on a user query, and insert logging information at the end. In a large program, especially one with good modular design, these functions might live in different modules that handle each of those abstractions in a decoupled way so that they never know about each other.

I could create a DBI object in each part of the program that needs to talk to the database, but I might end up with several open database connections. This is one of the symptoms I listed earlier – several identical, or close enough to identical, instances. Additionally, each time I open a database connection I incur additional overhead, which may be significant to my program.

Every time I run my program I take up multiple database connections even though I do not need to. The different parts of the program just need to talk to the database. They do not need their *own* connection.

To solve this problem I can create a module to control instantiation of my database objects so that my program only uses one connection. In code listing 3 I create a module named DBI::Singleton which subclasses DBI to control the number of instances. In this case I want a single instance.

Code Listing 3: Singleton DBI

```
 1  package DBI::Singleton;
 2
 3  use base qw(DBI);
 4  use DBI;
 5
 6  my $Dbh = undef;
 7
 8  sub connect
 9      {
10      my $class = shift;
11
12      return $Dbh if defined $Dbh;
13
14      $Dbh = DBI->connect( @_ );
15      }
```

This module looks for similar to my Counter module. I store the singleton instance in the lexical variable $Dbh, and connect() is my constructor. I inherit from DBI in line 5, so DBI.pm handles any other methods called on the DBI::Singleton instance. I also rely on DBI to create the instance since I call its connect() method explicitly.

That is all there is to it for my simple example. When I want a database connection anywhere in my program, I simply ask my DBI::Singleton module for one in the same way as if I were using the DBI module directly.

```
my $my_dbh = DBI::Singleton->connect( @arguments );
```

The $my_dbh variable stores a DBI instance since I did not re-bless it into my DBI::Singleton package. If you know about design patterns already, you may recognize a bit of the Factory and Adapter patterns in DBI::Singleton's connect().

How does it work? I call the connect() method in my package. The first argument is the name of the package, DBI::Singleton, which I ignore because I will return a DBI instance. Next, I check to see if the variable $Dbh is defined. I created $Dbh as a lexical variable with my() which means that it is only available inside DBI::Singleton file just as in my Counter example from code listing 1. If $Dbh is undefined, I continue through the method and connect to the database with the remaining arguments left on the argument list. I save the return value from connect() in $Dbh, which will be a DBI instance if the connect() succeeds and `undef`, the original value of $Dbh, if it fails. I still look in $DBI::errstr to see what went wrong, but I can fix that in section 4.2.

Assuming that the connection succeeds, the second time I try to connect to the database with DBI::Singleton, $Dbh is defined, so DBI::Singleton's connect() method returns the stored instance instead of a completely new one. As long as each part of my program that needs a database connection uses DBI::Singleton->connect(), my program should only ever use one database connection.

4.2 Class::Singleton

Andy Wardley created a base module, Class::Singleton, that I can use for any class I intend to be a singleton. If I put Class::Singleton into the inheritance tree of my singleton-to-be module by declaring it with `use base`, I can let Class::Singleton handle most of the details.

Code Listing 4: Using Class::Singleton

```
package DBI::Singleton;

use base qw(Class::Singleton);
use vars qw($errstr);

use DBI;

*errstr = *DBI::errstr;

sub _new_instance
    {
    my $class = shift;

    return DBI->connect(@_);
    }
```

The flow of this method is the same as in code listing 3, but Class::Singleton handles some of the details. I renamed my connect method to _new_instance(), which is a special method that Class::Singleton expects to be in my derived class. When I want an instance, I ask my module for one using the Class::Singleton method named instance().

```
my $my_dbh = DBI::Singleton->instance( @arguments );
```

Since the instance() method does not exist in my DBI::Singleton package, but I inherit from Class::Singleton, perl uses Class::Singleton's instance() method which does all that magic. The first time I call instance(), it looks for the method _new_instance() in my derived class. Class::Singleton expects to receive an instance from this method, and when it gets the instance, it stores so that the next time I call DBI::Singleton's instance() I get the remembered instance rather than a new one.

In line 8, I also fixed my problem accessing the DBI error messages, stored in $DBI::errstr, by aliasing $DBI::Singleton::errstr to $DBI::errstr with a little bit of typeglob magic. Accessing either of those variables accesses the same information.

4.3 Other paths to the same thing

I do not actually have too many other options on implementing a singleton. I need control of the instance so other code can not change it and affect distant parts of the program, and I need to be able to share it. Here are some other ways to implement a singleton. Some of them have disadvantages, but may fit particular situations better.

4.3.1 Global variables

Global variables allow me to pass data throughout a program with little care or thought, and to use those data with similar lack of care as long as I know the global variable name. In Perl I have an easier time with this since any package variable is really a global variable as long as I know the package name, and also some Perl special variables are always global variables.

However, the value of a global variable is only as good as what I last did with it, and as the programmer I can do just about anything I like with a global variable including changing its value, perhaps to something that makes no sense, which then affects other parts of my program. If I use a singleton to control access to this data, each part of the program gets what it expects because I provide a single point of access. In my examples, I hid the data as a lexical variable in my Counter and DBI::Singleton classes so that code outside of the file could not affect its value. Since global variables cannot provide this sort of protection, they are not always a good way to implement singletons.

The Class::Singleton module, which I discussed in section 4.2, uses package variables to store instances since it needs to be able to deal with many different derived classes.

The Win32::TieRegistry module demonstrates a good use of a package variable to represent its singleton. Every recent Microsoft desktop operating system has a registry where it stores useful information about the operation of the computer. Other parts of the system, including applications, can read and write to this registry. Although it is dangerous to change parts of the registry on which the operating system depends, it is useful for applications to store some of their own information in the registry.

The operating system has only one registry which all of the applications and processes share. Accessing the Registry perfectly describes the singleton pattern – many processes sharing one resource. In Win32::TieRegistry, the Registry is represented by a class variable $Registry which is set up when the module is first loaded, and exported by default. All subsequent uses access the same entity. The $Registry variable is actually a reference to a tied hash of hashes which is amazingly flexible in its access to its keys, so other approaches to its implementation would be much more complex and more difficult to understand.

4.3.2 Using class methods instead of instances

I could implement a singleton with a class that does not use an instance. Instead of instance methods, I use class methods. This would work adequately for my Counter example in code listing 1 since I only need to access the counter's value. However, if I implemented my database example with class methods, I would connect to the database and store the DBI instance as before, but the return value would not be the instance. It can be something else, although a value that tells me the success or failure of the action is simple and useful. Every time I want to call a method on the instance, I have to go through a class method. This provides the level of abstraction I need, but can be tedious and unnecessarily complex to implement because I have to go through two levels of method calls (one for the class and one for the instance) to do what I should be able to do directly.

```
my $status = DBI::Singleton->connect( @arguments );
die "Could not connect to database!" unless $status;
```

The first time that I call connect() the class connects to the database and stores the database object in a class variable, as before. The connect() method returns a status value so I know if the method succeeded.

From there, I have to do more work to get the rest of the functionality since I can only call class methods. For example, if I wanted to make a database query, the interface is a class method call.

```
DBI::Singleton->query( $query_text );
```

I create a query method that takes the arguments to the class method and translates it into something that the behind-the-scenes database handle can understand. In my previous instance implementation in code listing 3, the first argument to the method was the instance itself, which I called $self. When I call a class method, the first argument is the name of the class, as a string. In this example, $class is literally "DBI::Singleton".

Code Listing 5: query method wrapper

```
sub query
    {
    my $class = shift;

    my $sth = $Dbh->query(@_);

    return $status;
    }
```

Inheritance works as well in class methods and instance methods. my method does not know the difference without inspecting the first argument to the method to determine if it is a reference (meaning that it is an instance) or a string (meaning that it is a class). I will look at this more later in section 4.3.4.

4.3.3 Using functions instead of instances

I can also implement singletons without any explicit object-oriented programming. The CGI module does this for its functional mode of programming. It maintains an internal CGI object and controls access it to via its functions. If I use the object-oriented interface for CGI.pm, Code Listing 6: Object form of CGI.pm

```
use CGI;

my $input = CGI->new();

my @names = $input->param();
```

If I use the functional approach to do the same thing, CGI.pm still uses an object behind the scenes. No matter which function I use, I end up accessing the same CGI object, so this demonstrates a type of singleton.

```
use CGI qw(:standard);

my @names = param();
```

This has all of the problems of using class methods and the only difference is that the first argument to all of the functions is not an instance or a class name. The CGI module jumps through a couple of hoops to make this work seamlessly.

I have to use the functions in a different way than before. In the instance example perl could find the right methods because the constructor had blessed it into a class. Since my database example instance in code listing 3 was blessed into DBI, when I called a method, like query(), perl knew to look in the DBI package. In the class example, perl knew where to look because the class name was explicitly stated when I typed it into the program. But,if I want to use functions, then I have to tell perl how to find the functions and data. I have two options – give the full package specification for the function, like `&DBI::Singleton::connect( @arguments )` or export (or define) the functions into the current package.

The former works out to be the same as the class method approach save for the missing class name argument, and it has the advantage of showing future maintenance programmers from which package the connect() method comes.

The latter option is a bit tricky. If the functions are in the current package, I have to choose where to store the data which will represent the singleton. In the previous example I used a lexical variable. That does not work anymore because I cannot be sure that I will not unintentionally overwrite or hide something in the current package. In short, I have broken encapsulation and data hiding. If I created my functions in their own package, I could explicitly refer to variables in the original namespace.

```
sub query
        {
        $DBI::Singleton::Dbh->query( @_ );
        }
```

Not only do I have a lot of extra work to do all of the things that instances and classes gave me for free, but I have to expose the innards of my singleton module and lose the protection of the private variables. I would need a compelling reason to implement a singleton this way.

4.3.4 Using all three

Since Perl is not strongly typed, and since all arguments are simply passed as a list of scalars, I can, should I have a lot of extra time on my hands and nothing better to do, write methods that can handle instance methods, class methods, or functions, which I present here as merely for amusement, although it demonstrates a bit of Perl wizardry. I actually know of one developer who implemented such a thing for some enterprise software so that each program could choose its interface.

First, I need to determine which method I am using. If the first argument is a reference, then I use that reference as the instance. If it is a class name, however, I can ignore it. If it is neither, then I can assume I am using the functional style. Sound simple? Not so fast.

How do I figure out if the first argument is an instance? I cannot simply check to see if it is a reference, because references can be used for other things. I also cannot rely on any instance as the first argument being the right sort of instance since a reference might actually be the expected argument. I can check to see if the instance belongs to my package with `UNIVERSAL::isa()`. Even if the first argument passes my isa() test, I could still be wrong unless I completely control how the entire interface works.

```
my $object = '';

if( ref $_[0] and $_[0]->isa( __PACKAGE__ );
    {
    $object = shift;
    }
```

The second case expects a string that represents the name of the class, and if that is test is true, I use the private class variable, which in my DBI::Singleton example in code listing 3 was $Dbh;

```
elsif( $_[0]->isa( __PACKAGE__ ) )
    {
    my $class = shift;
    $object = $Dbh;
    }
```

If none of that works, I can assume that I am using the functional mode, and get the instance from the variable in which I stored it.

```
else
    {
    $object = $Dbh;
    }
```

Once I determine how I called the method (or function), I can get on with the real work, although I have to do a few tricks to support all three methods.

Looking at it written out as code I recognize that I am really choosing what $object will be, which is a just a contortion of a switch construct. Perl does not have an explicit switch construct, but I can do just as well with a do {} block which, like any other block, returns the last evaluated expression, as in code listing 7.

Code Listing 7: Doing it all at once

```
my $instance = do {
    if( ref $_[0] and $_[0]->isa( __PACKAGE__ );
        {
        shift;
        }
    elsif( $_[0]->isa( __PACKAGE__ ) )
        {
        my $class = shift;
        $Dbh;
        }
    else
        {
        $Dbh;
        }
    };
```

Putting that into absolutely every method or function, however, would take a lot of typing. A good text editor could do it quickly for me, but the next person who comes along and has to maintain my code may not have my fancy editor or macros so that is not a good idea either. Any time I have to type the same code more than a couple of times it is time for a pattern that only requires me to type it once. One way to do this is to use method dispatch. I could write a wrapper around all methods and functions that does this bit for me, then returns the right thing for the particular mode, but I do not discuss this here.

5 Singletons with a life span

I may want my singletons to disappear after some condition is met. Perhaps I should replace the single, remembered instance after a certain time has elapsed, or after a certain number of uses, or anything else that I can dream up.

Suppose that I want my singleton to be recreated if no other references to it exist. For example, I create some counters as I did in code listing 2, and when all of the counters are no longer in use, perhaps because they have all gone out of scope, the next request for a counter gives a completely new instance, or re-initializes the old instance, as appropriate. In my database example in code listing 3 I may wish to discard my singleton instance if I do not need the database server for a while so that I do not hold open the connection, and reconnect when I need it again.

I now want to modify the Counter example in code listing 1 so that I only get the same instance as long as some other part of the program is currently using it. When no part of the program is using the Counter instance, that is, no other program variables reference it, I want the module to create a completely new instance on the next request, which means the next request's counter starts at 0.

If I count the number of times I return an instance, I know how many variables reference my instance. For now, just assume that is true. In code listing 8 I create a new class variable to store the number of references to my singleton I think there are.

Code Listing 8: Reference counted singleton

```
package Counter;

my $singleton       = undef;
my $reference_count = 0;

sub new
    {
    my( $class ) = @_;

    if( defined $singleton and $reference_count > 0 )
        {
        $reference_count++;
        return $singleton;
        }

    my $self = 0;

    $singleton = bless \$self, $class;

    $reference_count = 1;

    return $singleton;
    }

sub reference_count
    {
    return $reference_count;
    }

sub value
    {
    my $self = shift;

    return $$self;
    }

sub increment
    {
    my $self = shift;

    return ++$$self;
    }

1;
```

In new() I moved things around so that I could count the number of times I returned a reference to my instance, which I store in $reference_count, and I added a class (and instance) method to return the number of references to the singleton.

Now I need to know when the use of a particular reference to my counter has ended so that $reference_count reflects reality. I cannot use DESTROY since it only works on the singleton instance when there are no more references to it, but a reference always exists to it since the class variable, $singleton, always exists, meaning that perl will never call DESTROY on $singleton. I could state explicitly that I have finished using a reference by calling a destructor, perhaps called destroy().

```
$count3->destroy;
```

My destructor, which I have to call explicitly, decrements the reference count and sets the caller to undef.

```
sub destroy
    {
    $_[0] = undef;
    $reference_count--;
    }
```

This is not good enough because I have to pay extra attention to make sure that I call destroy() explicitly and at the right place in the program and at the right time.

```
my $count = Counter->new();

$count->increment;
$count->value;

... stuff ...

$count->destroy;
```

This still is not good enough because I do not really know when a particular use of the singleton has ended – I only know when the programmer has remembered to tell the class he is finished with that instance. For example, suppose that I created a lexical counter, and let the counter go out of scope without calling destroy():

```
if( some condition )
    {
    my $counter = Counter->new();

    $counter->increment;

    ...stuff...

    $counter->increment;
    }
```

Now I have no way to decrement $reference_count because $counter no longer exists. Forcing programmers to do such things explicitly is quite unperly.

I also have no way to keep the programmer from making direct, shallow copies of the instance which I cannot count, and I cannot recognize the use of new() in a void context, so even though I do not store another reference to the instance, new() thinks I did and increments $reference_count.

```
my $count = Counter->new();

# we do not know this happened
my $count2 = $count;

# new() thinks I stored the return value
Counter->new();
```

I can improve this technique with a little black magic. Perl already counts the number of references to data. Perl's reference count will take into account lexical variables going out of scope and void contexts, and it will do it without my intervention. If I can look at this reference count then I do not have to count the references myself.

I can use the SvREFCNT function from Devel::Peek which gives me the number of references to its arguments, although a lot of people think that Devel::Peek is an unreasonable prerequisite for production code, and I agree. Devel::* modules are for development, in my opinion, but I can use it to get where I need to be, and I would not show this to you if I did not have something even better to replace it. So, if I decide to use Devel::Peek, the beginning of my Counter module looks like:

Code Listing 9: Using Devel::Peek

```
package Counter;

use Devel::Peek qw(SvREFCNT);

my $singleton;

sub new
    {
    my( $class ) = @_;

    if( defined $singleton and SvREFCNT($singleton) > 1 )
        {
        return $singleton;
        }

    my $self = 0;

    $singleton = bless \$self, $class;

    return $singleton;
    }
```

I can get rid of anything that deals with $reference_count since perl keeps track of that for me. After I check to see if $singleton is defined, I also check to see how many references to it exist. Remembering that $singleton itself counts for one reference, I test to see if there is more than one. If so, there is another reference somewhere in the program.

If I were really ambitious, I could even implement the reference count code myself in XS code as part of my module and avoid Devel::Peek entirely.

Code Listing 10: XS code to access reference count

```
int
ref_count(sv)
   SV* sv;

   CODE:

      RETVAL = SvREFCNT(sv);

   OUTPUT: RETVAL
```

Some of you might be thinking why I do not use the WeakRef module, since it makes all of this stuff much easier and I do not have to use the Devel::Peek or XS. The author, Tuomas J. Lukka, has marked WeakRef as experimental. Interestingly, it was originally written so that the author could avoid all of the reference counting mess I just went through since it allows Perl to ignore that fact that $singleton counts as one reference and DESTROY it when nothing else references it.

If I want to use WeakRef, the only thing I have to do is weaken $singleton, which signals to perl that $singleton will not count its existence in the reference count. When all of the other, “strong” references to $singleton disappear through Perl’s normal behavior, perl destroys $singleton and I can use DESTROY to do anything that I need to do. Afterwards, $singleton will be undefined and the next time I try to get an instance I will get a completely new one.

Code Listing 11: Using WeakRef

```
 1  package Counter;
 2
 3  use WeakRef;
 4
 5  my $singleton = undef;
 6
 7  sub new
 8      {
 9      my( $class, $count ) = @_;
10
11      if( defined $singleton )
12          {
13          return $singleton;
14          }
15
16      my $self = 0;
17
18      $singleton = bless \$self, $class;
19
20      weaken $singleton;
21
22      return $singleton;
23      }
```

In code listing 11 I weaken $singleton in line 20 with the weaken() function from WeakRef which tells perl not to count the existence of $singleton in the reference count to the data which it represents. Other references to that data, like the reference that I get back from new(), do count. When all of those references disappear, perl destroys $singleton.

6 Memoized (Modified) Singleton

A memoized singleton returns the same instance for the same arguments to the constructor. Previously, any call to my constructors returned the same instance no matter the argument list, but this is inadequate for many situations. I may consider using this modified form of the singleton pattern if I need different instances depending on the argument list, but still want the benefits of the singleton.

6.1 A simple example

Once I have read and parsed a configuration file, I should not have to do it again, although if I need to use it in different parts of the program, I should not have to do too much work to do that. It looks like I should use a singleton to represent the configuration file. If I used a pure singleton in which the class can only have one instance, then I can only have one configuration file. What if I need to look at multiple files, like .profile and .bash_profile when I login to a bash shell, or httpd.conf, access.conf, and srm.conf from apache? A pure singleton will not allow me to represent each of these at the same time from the same class, although they represent the same concept.

I need only one instance for each configuration file so that if I have already parsed the file, I skip all of that work and simply access the information. To do this, I need to remember which files I have already seen. I can modify my previous singleton examples to use a hash instead of a scalar to remember my instances. The keys of the hash will be the names of the configuration file, and its value will be the configuration file instance that goes with it.

I override the new() method from ConfigReader::Simple in code listing 12. In code listing 12, my memoized version of new() turns ConfigReader::Simple into a modified singleton class.

Code Listing 12: Memoized singleton

```
package Config::Singleton;

use base qw(ConfigReader::Simple);

my %Configs = ();

sub new
    {
    my( $class, $file, @args ) = @_;

    return $Configs{$file} if exists $Configs{$file};

    my $object = $class->SUPER::new( $file, @args );
    return $object unless ref $object;

    $Configs{$file} = $object
    }
```

In line 5, I declare the hash %Configs which I use to store the instances I create in new() – one instance per configuration file with the filename as the key and its instance as the value. Inside new(), at line 11, I check to see if the filename key exists in %Configs and if it does, return its value which is the ConfigReader::Simple instance. Otherwise, I continue with new() to create a new instance and store it in %Configs.

For anything more complex, I have to do more work. How do I recognize that two argument lists are the same? I cannot use a list of the key for a hash. I could serialize the list so that I end up with a string, perhaps with something like

```
my $key = join "\0", @_;
```

but what if the character `"\\0"` is a valid within the data? It is improbable that I will experience a collision with this technique, but it is certainly possible. How would I differentiate between the list `( "a\0", "b" )` and `( "a", "\0b" )`?

For most of the cases in which I would use a memoized singleton, the Memoize module available on the Comprehensive Perl Archive Network (CPAN) will do all of the hard work. It digests the argument list and stores the return values for each. I simply tell Memoize which methods to affect, then move on to the next programming problem. In this case I only have to memoize the constructor. In code listing 13 I use Memoize to store the return values of new(); this improves code listing 12 which only used the filename as a key, ignoring any other arguments.

Code Listing 13: Config::Singleton

```
1  package Config::Singleton;
2
3  use base qw(ConfigReader::Simple);
4  use ConfigReader::Simple;
5
6  use Memoize;
7
8  memoize( qw(new) );
9
10 sub new
11     {
12     my( $class, $file, @args ) = @_;
13
14     my $object = $class->SUPER::new( $file, @args );
15     return $object unless ref $object;
16     }
17
18 1;
```

In line 6, I use Memoize, and then in line 8, I use the memoize() function to perform the magic on new(). I can ensure that Memoize does the right thing by testing Config::Singleton the same way I tested Counter.pm in 2.

Code Listing 14: Testing Config::Singleton

```
#!/usr/bin/perl

use lib qw(.);

use Config::Singleton;

my $ref1 = Config::Singleton->new( '.profile' );
my $ref2 = Config::Singleton->new( '.bash_profile' );
my $ref3 = Config::Singleton->new( '.profile' );

print $ref1 eq $ref3 ? "Match (Good)!" : "Not Match (Bad)!", "\n";
print $ref1 ne $ref2 ? "Match (Bad)!"  : "Not Match (Good)!", "\n";
```

Output for Code Listing 14

```
Match (Good)!
Not Match (Good)!
```

6.2 Connecting to multiple databases

In my pure singleton example in code listing 3 I shared a single database connection. The first time that the program asked to connect to the database my connect() method passed the argument list on to the DBI's connect() method, then saved the returned database handle if the connect() succeed. Each subsequent time I tried to connect I got a reference to the same database connection even if I had asked to connect to a different database. I had simply ignored the arguments altogether. However, it is completely reasonable to connect to multiple databases, or the same database multiply, in certain situations. I have frequently kept two database handles active as I read from one to write to the other.

I need to modify my connect() method in the DBI::Singleton package from code listing 3 and I have to modify my $Dbh to store multiple database handles. I can do the same thing that I did in code listing 12.

The mod_perl folks solved that problem with Apache::DBI, which is an example of a memoized singleton.

Before mod_perl, one of the problems with web databases was that each CGI script used its own connection to the database server, and setting up that connection could be the major bottleneck in the program, making the web site seem slow. The mod_perl extension to Apache solved many of the performance problems with CGI scripts, but you still had to connect to the database. Since mod_perl processes stayed in memory between web requests, everything that needed a database connection maintained its own connection, causing resource use problems.

Apache::DBI stores one database connection that all parts of the process can share. It even pings the database server and re-establishes the connection if need be. You might already use Apache::DBI without knowing it since DBI looks for it when it is loaded with Apache, and will forward connection requests to Apache::DBI if apache has already loaded it. You can inspect the Apache::DBI::connect() source code to see how the authors implemented their singleton.

7 Conclusion

The Singleton design pattern structures code so that different parts of a program can share a resource without having to work out the details of the sharing. I showed several ways to implement this pattern, but the importance of the pattern is in the structure of the code rather than in its implementation, and different situations may need different implementations.

8 References

You can get all modules discussed in this article from the Comprehensive Perl Archive Network (CPAN).

- CGI
- Win32::TieRegistry
- ConfigReader::Simple
- DBI
- Apache::DBI
- Memoize

You can read more about design patterns in:

Design Patterns, Erich Gamma, Richard Helm, Ralph Johnson, Jon Vlissides, Addison Wesley, 1995.

Camels and Needles: Computer Poetry Meets the Perl Programming Language

Sharon Hopkins, Telos Corporation

Abstract

Although various forms of literature have been created with the assistance of a computer, and even been generated by computer programs, it is only very recently that literary works have actually been written in a computer language. A computer-language poem need not necessarily produce any output: it may succeed merely by fooling the parser into thinking it is an ordinary program. The Perl programming language has proved well-suited to the creation of computer-language poetry.

[*Editor's note: Sharon originally presented this at the 1992 Usenix Winter Technical Conference, at it was published in it's proceedings.*]

1 Introduction

Over the past few decades, considerable work has been done in the area of computer-generated writing. Projects have ranged from programs designed to find simple anagrams, to dedicated pangram [1] generators, to entire poems or stories (usually nonsensical) produced by stringing together chains of words that occur next to each other in human-generated texts. One French group [2] went so far as to experiment with computer-generated writings that could be read in both French and English. A number of science fiction and fantasy stories have been written in the form of flow charts, or designed to appear as if they had been written in some programming language yet to be invented. But until very recently, little or no attempt has been made to develop human-readable creative writings in an existing computer language. The Perl programming language has proved to be well suited to the creation of poetry that not only has meaning in itself, but can also be successfully executed by a computer.

1.1 Why Computer Poetry?

The question naturally arises: Why write poetry in a computer language at all? Computer-language poetry is essentially no different from any other formal poetry, except that the rules of the computer language dictate the form the poem must take, and provide a mechanism for measuring success. If the poem executes without error, it has succeeded. The great advantage formal poetry has over free verse is the balance provided between familiarity and strangeness, stasis and innovation. When reading rhymed poetry, for example, the reader develops an expectation of how each line will end. The poet's task is to satisfy that expectation, by providing a rhyme, while surprising the reader with a word or meaning outside of his expectations. With

[1] A pangram is a sentence that contains each letter of the alphabet, a definite number of times. See Dewdney, A.K. "Computer Recreations: A computational garden sprouting anagrams, pangrams and few weeds." Scientific American. Vol. 251, Num. 4, pp. 20-27, October 1984.

[2] Motte, Warren F., ed. Oulipo: A Primer of Potential Literature. University of Nebraska Press, 1986.

computer-language poetry, the surprise is enhanced: the reader develops expectations as to what is going to come next based on familiarity with the standard uses of that language's vocabulary. In Perl, if the word read() appears, the person reading that Perl script will naturally expect the items in parentheses to be a filehandle, a scalar variable, a length value, and maybe an offset. The poet, finding read() in the Perl manual or reference guide, is more likely to write `read (books, poems, stories)`. In the same way, sin() (of sine, cosine, and tangent) becomes sin, the downfall of man. Another advantage to writing computer-language poetry, in addition to the challenge provided by the constraints of the form, is that you can get some creative writing done and still look like you're working when the boss glares over your shoulder.

1.2 Why Perl?

The Perl programming language provides several features that make poetry writing easier than it might be in other languages. COBOL poetry, for example, would always have to start with IDENTIFICATION DIVISION. While that would make an excellent opening for one poem, it's hard to see what could be done for the next one. Besides, getting a volunteer to write poetry in COBOL is likely to be impossible. We do have one example of a poem in FORTRAN (Fig. 1), but FORTRAN, too, is tightly constrained in that most of the interesting things have to be said up front. Layout of FORTRAN poems is also tricky, given the limitations on what can safely be put in the first few columns. One of the reasons for Perl's popularity is that you don't have to say what you are going to say before you say it. In addition, Perl has in its vocabulary something on the order of 250 words, many of which don't complain when abused. Shell poetry is an attractive possibility, with plenty of good words available. Unfortunately, most of the words available in a shell script are likely to return non-poetic messages at artistically inappropriate moments. In Perl, there is a decent supply of usable words, white space is rarely critical, and there is almost always "More Than One Way To Do It". And since Perl is interpreted rather than compiled, you never have to keep ugly binaries around pretending to be your poem; in addition, poem fragments can be tested quickly by running perl interactively, or by feeding lines to perl on the command line via the perl -e switch.

1.3 A Little History

Randal Schwartz probably deserves credit for inspiring the first-ever Perl poems. Randal's "Just another Perl hacker,"(JAPH) signature programs prompted me to suggest to Larry Wall that he write a JAPH that would also be a haiku. By lunchtime that day (mid-March of 1990), Larry had produced the world's first Perl poem (Fig. 2):

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

In this poem, the `q` operator causes the next character (in this case a newline) to be taken as a single quote, with the next occurrence of that delimiter taken as the closing quote. Thus, the single-quoted line `Just another Perl hacker,` is printed to `STDOUT`. In Perl, the `unless $spring` line is mostly filler, since `$spring` is undefined. In poetical terms, however, `$spring` is very important: haiku poetry is supposed to specify (directly or indirectly) the season of the year. As for the `q` operator, that reads in English as the word "queue", which makes perfect sense in the context of the poem.

2 A Few Ways to Do It

The above poem demonstrates a particular difficulty with Perl poetry. Since Perl variables begin with `$` , `@` , or `%` , the aspiring Perl poet must decide whether or not these type markers should be pronounced. In the "Just another Perl hacker," poem, pronouncing `$` as "dollar" is necessary, as haiku by definition consist of one 5 syllable line, one 7 syllable line, and a final 5 syllable line. Notice that `STDOUT` must also be pronounced "Standard Out" for the poem to work. Most Perl poetry that has been written so far has tried to work Perl's special characters into the poems, reading `$` as "dollar", and `@` as "at". Similarly, punctuation marks (semicolons at ends of lines, parentheses, etc.) are most often worked into the text of a Perl poem. An alternative approach to Perl poetry punctuation has been to line all extraneous markings (quotes, semicolons, etc.) along the right hand margin, and to ignore dollar signs and the like when they occur within the body of the poem. This can help make Perl poetry look more like English, but it tends to be messy, with random bits of unmatched punctuation remaining scattered through the poem.

2.1 Working Perl Poems

There are basically two types of Perl poetry: those poems that produce output, and those that do not. The former, not surprisingly, are much harder to write. After the first Perl haiku, the only examples of "useful" Perl poetry are Craig Counterman's "Ode to My Thesis" (Fig. 3) and "time to party" (Fig. 4). "Ode to My Thesis" is readable in English, parses in Perl, and produces text output summarizing and concluding the theme developed within the poem. In order to hide items like the `>&2` construct, and other unpronounceables, most of the punctuation has been relegated to the far right of the screen or page. Disposing of inelegant punctuation has the added advantage of obscuring the intent of the program, making it hard for the casual observer to guess what output the poem will produce. The theme of "Ode" interacts nicely with the Perl vocabulary, as words like `write`, `study`, and `sleep` seem natural in a poem dedicated to the joys of completing a dissertation.

In "time to party", Craig uses Perl's file-handling mechanism to make the poem produce output: in this case, creating an output file called 'a happy greeting' which contains the signature line for the poem. Other methods for producing output from a Perl poem might include system calls with useful return messages, or output from the `die()` and `warn()` commands. For input obfuscation, various tricks can be played on the Perl copyright notice, which is accessed through the `$]` special variable. This technique has been used in at least one JAPH script, and could be made to work in a Perl poem using Craig's format, with non-poetic constructs moved out of the line of vision.

2.2 "A poem should not mean but be."[3]

Non-useful Perl poetry, my own specialty, has two basic visual formats: the first type looks more like standard English free-verse poetry, and tends to be built on individual functions and operators that are selected to create a particular mood or image; poems in the second format tend to be more stanza-oriented, and to take a sort of word-salad approach, using any and all Perl reserved words that can be made to fit. Most poems of the first type, those with a more free-form linear organization, tend to be what I refer to as "keyword" poems. The poems are built around one or more Perl reserved words that also carry a weight of meaning in English. Examples of this type of poem are my `listen` and `reverse` poems (Figs. 5 & 6).

[3]From "Ars Poetica", by Archibald MacLeish. This statement is frequently used as a justification for writing poetry completely lacking in communication value.

In "listen", a second-person poem speaking directly to the reader, most of the weight of the poem is carried by Perl reserved words: `listen()`, `open()`, `join()`, `connect()`, `sort()`, `select()`. Since Perl parses bare words as quoted strings, extra words can be included in each expression to provide added meaning in English without causing syntax errors. Thus, `join()` becomes `join (you, me)`, and `connect()` becomes `connect (us, together)`. Because Perl rarely worries about white space, the poem can be arranged on the page for a stronger visual impact. For example, the statement `do not die (like this) if sin abounds;` can be broken with a newline (making the phrase read as separate lines in English) without confusing Perl. In addition, the comma operator can be used to stretch out Perl statements, and relieve the monotony of a poem filled with semicolons.

The poem "reverse" makes use of Perl's conditional (if-then-else) operator, `?:` , to provide variation in tone, so that the poem reads more as a conversation, not merely as a set of commands or instructions. In this poem, words like `alarm`, `warn`, `sin`, `die`, and `kill` are used to create tension, and to increase the poem's emotional impact. The fact that all of these are perfectly ordinary and mundane functions in Perl heightens their surprise value when used in a poem. At the same time, the oddity and unfamiliarity of Perl syntax, when read in English, gives the poem a sinister and mysterious flavor. At the end of the poem, the keyword `reverse` (used as `reverse after`) is intended to give the poem an ironic twist, making it clear that nothing previous in the poem can be trusted.

The critical process for the keyword type of perl poem is weeding out words and phrases that do not enhance the meaning of that particular poem. I have one Perl poem, called "limits", that is composed almost entirely of lines that occurred to me while I was writing "listen", but which did not seem to fit the "listen" poem thematically. It can be difficult, however, to strike a balance between meaning and mystery; it's hard to determine when you have said enough to make the poem coherent as poetry without sacrificing the integrity of the program. Because it's easy to include quoted material in Perl programs, one is often tempted to simply wrap a pair of quotes around everything that won't pass the Perl parser. The typical end-result is a poem that is neither surprising enough to be interesting in English nor Perlish enough to be interesting as a Perl program. My poem "rush" (Fig. 7) unfortunately falls into this category.

The word-salad type Perl poetry is often more interesting than the keyword-based Perl poetry, both from a programming and from a literary standpoint. Two example poems of this type are Larry Wall's "Black Perl" poem and my "shopping" poem (Figs. 7 & 8). Unlike the more loosely organized Perl poems, these stanza-based poems are generally created using the "kitchen sink" approach. Working from a list of all of Perl's reserved words (and pieces of words), the goal is to fit as many as possible into the Perl poem while maintaining a consistent theme or image. This type of poem is likely to be more humorous than those with a more restricted vocabulary, and the end results are surprisingly readable. Producing a "kitchen sink" Perl poem can require considerable ingenuity on the part of the programmer-poet, since many Perl words do not fit easily into English-language poetry. For example, Perl's `system()` function becomes "system-atically" in the poem "shopping", and `unlink()` is used in `unlink arms` in "Black Perl". "Black Perl" is particularly remarkable since it was written before poetry optimizations (allowing bare words to be interpreted as quoted strings) had been added to Perl.

The visual impact of the "kitchen sink" type of Perl poem also differs from that of the keyword-based poems. In both "Black Perl" and "shopping", the action of the poem is divided into separate visual blocks, each with an identifying label. In Perl, statement labels can be any single word (preferably upper case), followed immediately by a colon. Statement labels provide an excellent method for inserting extra English words into a Perl poem while providing the poem with additional structure and cohesiveness. While this technique is also used in the keyword type poems, it is not as evident as in the more stanza-oriented Perl poetry.

2.3 "All poems are language problems." [4]

When writing poetry that is meant to run without producing any output, one useful trick is to make sure the program exits without executing all of its statements. In the "Black Perl" poem, the program actually exits about a third of the way through the first line. The rest of the poem will still have been checked for syntax errors, but the various remaining `die`, `warn`, `kill`, and `exit` functions are never actually called. Likewise, in the "shopping" poem, the goto statement at the end of the first stanza causes Perl to skip past the second stanza, which would otherwise provide an ugly warning message. This poem also exits before its end, thereby avoiding an error message in the `later:` stanza, at `goto car` (since subroutine `car` does not exist). Judicious use of the `.` (concatenation) `,` (comma) and `?:` (conditional) operators can help with semicolon avoidance. Similarly, playing with white space can help make a Perl poem more readable.

3 Pros and Cons of Perl Poetry

Writing poetry in Perl provides a number of literary "pluses". The nature of computer languages, where most of the available words are commands, forces the computer-language poet to write in the imperative voice, a technique that is otherwise usually avoided. But the use of short, imperative words in a poem can actually serve to heighten the drama, especially with meaning-laden words such as `listen`, `open`, `wait`, or `kill`. In Perl, all but the last statement in a program require a concluding semicolon; leaving the final statement of a Perl poem bare of punctuation can help the poem's meaning seep into the reader's subconscious (this technique is used in all of the non-working perl poems included with this paper).

I had hoped that learning to write poetry in Perl would be of assistance in learning more conventional Perl programming, but this has not proved true in practice. I have absorbed a fair amount of Perl syntax, and a little bit of Perl "style", but in general Perl programs that read as poetry tend not to be in idiomatic Perl. For example, constructs that are very important in Perl, such as the `$_` special variable, and associative arrays, are difficult to work into Perl poetry. Perl semantics in particular get short-changed, as a main goal of Perl poetry is to use reserved words for unexpected purposes. One learns how many arguments each function takes, but not what a given function is supposed to do. In addition, use of all-lower-case unquoted words is a bad habit to get into, as new reserved words might be added to Perl at any point. A sloppy "if it parses, do it" mentality is not generally a useful programming style to adopt.

4 Conclusion

With only three or four practicing Perl poets, the field is still too new, and too small, for a thorough study of Perl poetry. Other computer languages boast an even smaller set of practicing poets, making cross-language poetry comparisons impossible to undertake. Perl is currently the language of choice for writing computer-language poetry, but this may be more by accident than by design. Clearly, Perl is suitable for writing poetry, but not ideal. As Perl continues to spread, and more people begin to express themselves in Perl, it is hoped that more poetry will be written in Perl by people of various backgrounds and writing (or hacking) styles. Poetry in other programming languages (even COBOL) would also be welcome; it may be that some other language, already in existence, would be even better suited to poetic flights than Perl is. It is hard to imagine, however, another programming language sufficiently quirky to attract the sort of hacker who would willingly program in poetry.

[4]From "When the Light Blinks On", an article by Eliot T. Jacobson in rec.arts.poems, message-ID <4437@oucsace.cs.OHIOU.EDU>. Original author unknown.

5 Availability

For more information regarding perl poetry, or for copies of perl poems written by Sharon Hopkins, e-mail sharon@wall.org.

Other authors whose works are discussed here must be contacted individually regarding further reproduction of their poems.

The first part of the title for this paper, “Camels and Needles”, is taken from a passage in the Bible where the statement is made that “it is easier for a camel to go through the eye of a needle than for a rich man to enter the kingdom of God.” (Matthew 19:24) Writing perl poetry is kind of like shoving camels through needles. . . .

Figure 1: *program life*, David Mar, 18 Mar 1991

```
        program life
        implicit none
        real people
        real problems
        complex relationships
        volatile people
        common problems
        character friendship
        logical nothing
        external influences
        open (1,file=friendship,status='new')
2       format (a,'letter')
        write (1,2) 'her'
c       what happens
        if (nothing) write (1,2) 'her again'
          continue
        if (nothing) then
          close (1,status='delete')
        else
          open (2,file='reply',status='old',readonly)
          read (2,2) friendship
          close (2,status='keep')
          close (1,status='save')
        end if
        end
```

Figure 2: *perl haiku (untitled)*, Larry Wall, 18-Mar-1991, lwall@jpl-devvax.jpl.nasa.gov

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

Figure 3: *Ode to My Thesis*, Craig Counterman, April 27, 1991 ccount@athena.mit.edu

```
# Ode to My Thesis, a Perl Poem
# (must be run on Perl 4.0 or higher)
<<birth ;
G
   r
      o
         w
            t
               h
re-
birth

seek                                                                 (
        enlightenment, knowledge, experience                         );

goto MIT;

sleep "too little", study $a_lot,
wait, then                                                           .
        "B.S.",

leave. then, return to                                               ;
               MIT                                                   :
now,
       $done = ’a Ph.D.                                              ">&2;

warn pop @mom, " I’ll be here a while                                \n";

study, study, do study;

push                                                                 (
        myself, computers, experiments                               ),

read                                                                 (
        data, references, books                                      ),

study,
        write,
                write,
                        write,

do more if time                                                      ;
redo if $errors                                                      ;

do more_work if questions_remain                                     ;

$all_are_answered? yes.
```

```
now :
         write,
         chop if length $too_great ;

format                                                                =
           Thesis
.
                         tell all,
                         done, finally                                .
                         now, do rest                                 .
shout.

      and                                                             .
                hear                                                  .
                        it                                            .
                                 ‘echo                                "

               Now I am $done‘
```

Figure 3.1: Output from *Ode to My Thesis*

```
I’ll be here a while
      Thesis
      Thesis
      Thesis

            Now I am a Ph.D.
```

Figure 4: *time to party*, Craig Counterman, April 27, 1991 ccount@athena.mit.edu

```
# time to party
# run using perl 4.003 or higher

<<;
done with my thesis

shift @gears;
study($no_more);
send(email, to, join(@the, @party, system));

open(with, ">a happy greeting");
time, to, join (@the, flock(with, $relaxed), values %and_have_fun);
connect(with_old, $friends);
rename($myself, $name = "ilyn");
$attend, local($parties);

pack(food, and, games);
wait; for (it) {;};

goto party;
open Door;
send(greetings, to, hosts, guests);
party:

tell stories;
listen(to_stories, at . length);
read(comics, $philosophy, $games);

seek(partners, $for, $fun);
select(with), caution;
each %seeks, %joy;

$consume, pop @and_food;
print $name .$on .$glass;

$lasers, $shine; while ($batteries) { last;};

time; $to, sleep
sin, perhaps;

$rest,
$till .$next .$weekend;
```

Figure 5: *listen*, Sharon Hopkins

```
APPEAL:

listen (please, please);

    open yourself, wide,
     join (you, me),
    connect (us,together),

tell me.

do something if distressed;

     @dawn, dance;
     @evening, sing;
     read (books,poems,stories) until peaceful;
     study if able;

     write me if-you-please;

sort your feelings, reset goals, seek (friends, family, anyone);

         do not die (like this)
         if sin abounds;

keys (hidden), open locks, doors, tell secrets;
    do not, I-beg-you, close them, yet.

                    accept (yourself, changes),
                    bind (grief, despair);

    require truth, goodness if-you-will, each moment;

select (always), length-of-days

# Sharon Hopkins, Feb. 21, 1991
# listen (a perl poem)
```

Figure 6: *reverse*, Sharon Hopkins

```
first:

tempted? values lie:

    do not, friend, alarm, "the flock";
     wait until later;
         warn "someone else"

& die quietly if-you-must;


then:

sin? seek absolution?
if so, do-not-tell-us, "on the sly";

    print it "in letters of fire",
    write it, "across the sky";

kill yourself slowly while each observes;
 do it (proudly, publicly) if you-repent;

     tell us,
     all,
     everything;


reverse after



# Sharon Hopkins, Feb.27, 1991
# reverse (a perl poem)
```

Figure 7: *rush*, Sharon Hopkins

```
'love was'


&& 'love will be' if
               (I, ever-faithful),
do wait, patiently;

"negative", "worldly", values disappear,

@last, 'love triumphs';

    join (hands, checkbooks),
    pop champagne-corks,

"live happily-ever-after".

     "not so" ?
      tell me: "I listen",
                                   (do-not-hear);

push (rush, hurry) && die lonely if not-careful;

"I will wait."


&wait



# Sharon Hopkins, June 26, 1991
# rush (a perl poem)
```

Figure 8: *Black Perl*, Larry Wall

```
BEFOREHAND: close door, each window & exit; wait until time.
    open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
    reverse its length, write again;
     kill spiders, pop them, chop, split, kill them.
        unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
    kill them, dump qualms, shift moralities,
     values aside, each one;
         die sheep! die to reverse the system
          you accept (reject, respect);
next step,
    kill the next sacrifice, each sacrifice,
     wait, redo ritual until "all the spirits are pleased";
    do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*).
return last victim; package body;
    exit crypt (time, times & "half a time") & close it,
     select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
    wait, wait until time;
     wait until next year, next decade;
         sleep, sleep, die yourself,
          die at last
```