The Perl Review

Volume 0 Issue 5 September 1, 2002

Like this issue? Support *The Perl Review* with a donation! http://www.ThePerlReview.com/

Like this issue? Support *The Perl Review* with a donation! http://www.ThePerlReview.com/

Web Access http://www.ThePerlReview.com/ **Email** letters@theperlreview.com **Publisher** brian d foy **Editor** Andy Lester **Technical Editors** Kurt Starsinic, Adam Turoff **Copy Editors** Glenn Maciag, Beth Linker, James Tillman **Contributors** David H. Adler, Phil Crow, brian d foy, Andy Lester, Beth Linker, Allison Randal, Robby Walker

Letters

Send your letters, comments, and suggestions to letters@theperlreview.com

TPR announcement list

Perhaps you could create a subscription list to announce new issue releases? Or did I miss this somewhere?
– *John Mooney*

brian writes: *You actually asked this a while ago, and I have not had an answer until now. We created an announcement-only list on Topica.com where we will post notices of new issues. You have already signed up, but if anyone else wants to sign up send a mail to the-perl-review-annouce-subscribe@topica.com or visit http://www.topica.com/lists/ the-perl-review-annouce/subscribe/?location=listinfo*

About this issue

We have the same things that we have had in past issues and a little bit more. In the last issue, we added the Short Notes section and this month we have reports from this summer's Perl events. We added a new section to list recent addditions to CPAN, and a list of other magazines that have regular Perl content. Andy Lester adds "Extreme Mowing" to our earlier "Extreme Publishing" series, and part-time Perl hacker and full-time Java developer Beth Linker, one of our copy editors, tells us about Java in the latest article on other languages (we did Python already, and hope to do Ruby and Lisp too). We can also take donations now. If you like this issue, which we give away for free, support us with a donation. We will use all the money we raise to turn this into a real print magazine.

Perl on the web

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

Perl Monks
http://www.perlmonks.org/ – Perl discussion forum

Use.perl
http://use.perl.org – Perl news and commentary

Perldoc.com
http://www.perldoc.com – Online, searchable Perl documentation

Write for *TPR*

Have something to say about Perl? *The Perl Review* wants first person accounts about using Perl. If you cannot write a complete article you can write a "Short Note". Want to tell everyone about a book you have read? Write a book review! Were you at a Perl function? Give us a trip report!

We would like to get articles or "Short Notes" on

- Perl & Ruby
- Bioinformatics
- Perl internals
- Cute Perl hacks
- Debugging Perl
- Creating modules
- *and more . . .*

We also like articles aimed at Perl for beginning programmers. Perl people take for granted some things that never make it into books or get passed on to beginners. Do you have something new Perl programmers should know? Perhaps:

- Using templates
- Using configuration files
- Argument processing
- Deciphering documentation

You can get submission guidelines from our website, http://www.theperlreview.com.

Volunteer for *TPR*

We have not turned into a business yet, so we still rely on the generosity and availability of volunteers. If you have something to add to *The Perl Review*, send us a note!

Community News

Send us your news stories at news@theperlreview.com.

Parrot handles 4 Apocalypses

http://dev.perl.org

Dan Sugalski announced last month that Parrot can handle the Perl features Larry Wall outlined in the Perl 6 Apocalypses 1 to 4. You can find the Apocalypses and Damian Conway's Exegesis in the article archive on http://www.perl.com. Parrot can also handle Ruby, Python, BASIC, Scheme, and Forth.

New perl.apache.org Released

http://perl.apache.org

Stas Bekman says: After 10 moons of work a team of mod_perl bees released a shiny new perl.apache.org, with a new face and a new content. Happy mod_perl devotees celebrated this event with many pizzas, lots of beer and some cranberry juice. Further details available at http://mathforum.org/epigone/modperl/yixgoxplix/3D2F03F2.4060509@stason.org.

New search.cpan.org

http://search.cpan.org

Graham Barr released an updated version of the popular CPAN search engine. Most of the improvements deal with some behind-the-scenes performance enhancements, although you will see a slick, new interface too.

Perl Documentation Project

http://documentation.perl.org

Casey West had the idea for an organized documentation effort last year, but waited until this year to jump start the effort. He hopes to improve and expand Perl's documentation to make it easier for users to use it and easy for people to maintain and update it. He also wants to add more non-reference documentation in the spirit of Linux or FreeBSD HOW-TO documentation. Check the website if you would like to contribute.

Jarkko's Perl 5.8 slides available

http://conferences.oreillynet.com/presentations/os2002/hietaniemi.tgz

You can download Jarkko Hietaniemi's Perl Conference slides from the conference website. Jarkko shows what's different with the latest stable distribution of Perl.

Perldoc.com adds Perl 5.8.0

http://www.perldoc.com

Carlos Ramirez added the documentation for Perl 5.8.0, although you can still choose which version of the documentation you want to search.

Perl 5.9 work starts

Hugo van der Sanden took over the pumpking for the next experimental version of Perl, which might lead to the last minor release for Perl 5—version 5.10.

O'Reilly Community Press

http://press.oreilly.com/mysqlref.html

O'Reilly released a printed version of the MySQL documentation under its new imprint, The O'Reilly Community Press whose goal is to provide convenient, printed versions of existing online documentation. Future titles include *The Complete FreeBSD* and *DocBook*.

ActiveState, HP join forces

http://www.activestate.com/Corporate/Communications/Releases/Press1029188474.html

ActiveState and Hewlett-Packard announced in August that they would work together to optimize Perl for the HP-UX operating system. ActivePerl is already available available for HP-UX 11.0 and HP-UX 11i. All enhancements go back into the Perl core.

2002 White Camel Awards

http://www.perlfoundation.org

The Perl Foundation took over the community White Camel Awards started by Perl Mongers in 1999. This year's winners are Tim Vroom for PerlMonks.com, Tim Maher for his work with the Seattle Perl Users Group, and Graham Barr for his work with CPAN and search.cpan.org.

2002 Active Awards

http://www.ActiveState.com/ActiveAwards2002

Matt Sergeant won two Programmer's Choice awards: one for Perl and one for XSLT. Andy Dougherty won the Activators' Choice Award for his long-time involvement with Perl.

MacOS X Conference

http://conferences.oreillynet.com/macosx2002/

O'Reilly & Associates host the first MacOS X conference September 30-October 3 in Santa Clara, CA. Randal Schwartz and brian d foy will talk about Perl on MacOS X and Dan Sugalski will talk about CamelBones and Cocoa. Monday night has a Perl BOF.

New books

Publishers: to have us list your book, send us a note at book_reviews@theperlreview.com.

Writing Perl Modules for CPAN
Sam Tregar; Apress;
1-59059-018-X; July 2002
Andy Lester reviews this book in this issue.

Perl & LWP
Sean Burke; O'Reilly & Associates; 0-596-00178-9; 400 pages; July 2002

Mastering Regular Expressions, 2nd Edition
Jeffrey E. F. Friedl; O'Reilly & Associates;
0-596-00289-0; July 2002

Extending and Embedding Perl
Tim Jenness and Simon Cozens; Manning;
1-930110-82-0; July 2002

MySQL Reference Manual
Michael "Monty" Widenius, David Axmark, and MySQL AB; O'Reilly Community Press;
0-596-00265-3; July 2002

Essential Blogging
Benjamin Trott, et al.; O'Reilly & Associates;
0-596-00388-9; September 2002

LDAP Programming
Clayton Donley; Manning;
1-930110-82-0; October 2002

Perl CD BookShelf, 3.0
Larry Wall, et al.; O'Reilly & Associates;
0-596-00389-7; September 2002

Would you like to review a book? Send your review to book_reviews@theperlreview.com

Perl In the Press

Know of another magazine with regular Perl content? Let us know at letters@theperlreview.com

Linux Magazine – http://www.linux-mag.com/

;login: – http://www.usenix.org/publications/login/

Unix Review – http://www.unixreview.com/

Short Notes

In the spirit of the 5-minute Lightning talks run by Mark-Jason Dominus at various conferences, *The Perl Review* publishes "Short Notes". If you have something that you want to show off without writing an entire article, like a cool Perl trick, a module you just released or something happening in the Perl community, send your short note, between 200 and 400 words, to *short_notes@theperlreview.com.*

Perl 6 Mini Conference

Allison Randal, al@shadowed.net

The Perl 6 design team will meet in Zurich, Switzerland during the second week of September. The Swiss Federal Institute of Technology, Zurich (Eidgenössische Technische Hochschule Zürich or ETH) will host the team for two days of public conference and 4 days of development meetings. The Perl 6 Mini Conference (September 12th-13th, http://perl6.ethz.ch) is an opportunity for the European Perl community to hear first-hand about the future of Perl and to interact with some of the people most involved in shaping it. Community participation is a big part of the Perl 6 development process, so we expect these two days will be as valuable to the design team as they are to the conference attendees.

The development meetings will mainly focus on OO syntax and implementation. We [Larry Wall, Damian Conway, Dan Sugalski, Allison Randal, Hugo van der Sanden] covered many of the basics in June at a series of meetings held just before YAPC::NA. But, there are monsters left to tame in the shape of multiple inheritance, multiple dispatch, attribute definition and operator overloading, to name a few. We also plan to make more progress on retrofitting the first few Apocalypses to the current design. So far, these face-to-face meetings have been immensely beneficial to the design process. The intensity of thought possible when you sit people down in a group for 8 hours can't quite be matched by email or on the telephone.

YAPC::NA::2002

Sarah Burcham, sarah@pound.perl.org

Although I have long considered myself an advocate for the Open Source community and specifically the Perl community, the significance and meaning of that solidified this past year when I signed up to be the local coordinator for the Yet Another Perl Conference America::North::2002 [in St. Louis, MO]. Most significantly, I learned that contributing back to this community is difficult. I realized that the contributors to Open Source are just people who make personal sacrifices because they are committed to a community whose ideals they share. Contributing is hard, draining and not at all like being a Hollywood gladiator. This thought reverberated with those I recently heard from Orson Scott Card during a book signing tour. Speaking of heroes and community, he described an adolescent hero as the autonomous, alienated protagonist. In contrast, the mature hero makes the decision that considers what is best for those in his community. Altruistically, I signed on to organize YAPC because I highly regard the people that make the Perl community thrive. I wanted to give something back. As the critical June 26 conference date approached, stress piled up and I began to question the sanity of signing-up as coordinator. Altruism is difficult when you're tired and have chest pains. People had complaints, opinions, and suggestions. I had two jobs: a conference to coordinate, two hands, and massive sleep deprivation. When Kevin Lenzo—originator of YAPC and Yet Another Society—and volunteers (Elaine Ashton, Ann Barcomb, Simon Cozens, Jos Bouman, Ben Hockenhull, Jacob Kenner, Bill Odom, Nathan Torkington, etc.) arrived, I began to remember the reason why I was motivated in the beginning: the community. The people I professionally and personally identify with most are scattered across the country and the globe. I was glad to be a part of bringing these people together for YAPC America::North 2002. I plan on being one of the first to show up and help the organizers for next year. I already empathize with their chaos.

Perl Conference Trip Report

brian d foy, comdog@panix.com

The Sixth Perl Conference was at the end of July, and though I was there, I didn't see any of the talks since I have either seen them before or knew the subject matter. Since I had a press pass I wandered around bugging anyone who would talk to me.

O'Reilly has "Gone Green", which surprised me—not that they would but that they weren't already since they are based in central California after all. This year, the conference bags and t-shirts were 100% recycled cotton printed with a special, environment friendly Planet Ink. Various other parts of the goodies and handouts came from recycled material. Angela Capo Persinger, O'Reilly's conference planner, hopes to add even more Green items to the O'Reilly attendee bags for future conferences. I like to think that open source can be good for the planet, too.

Apple Computer was a major sponsor this year. An informal poll by Adam Turoff showed that a significant number of attendees with laptops had the shiny new iBooks. Those without their own could use one of the 40 Apple desktop machines in the terminal room.

The Open Source Conference was even more family-friendly this year. Not only did it have a separate kids program (which I did not attend, so you have to ask someone else), but you could buy meal passes for people not attending the conference. I have been at all of the Perl Conferences and have been a speaker for several other conferences, and O'Reilly consistently comes up with new, great ideas that surpass anything other conference promoters do.

While talking with various authors and publishers a story started to emerge—Perl may be losing out to other technologies like PHP in some parts of the world because it does not have documentation in the native language. I heard about this from a few directions and I am still looking into it, but if anyone has anything to add to the rumor please let me know.

Bryan Richard was giving away copies of his new Python magazine, Py, printed on newsprint. It looks nice, has some great articles, and is inexpensive. I might try this for *The Perl Review* soon. I got to buy Jon Orwant some drinks while he told me everything he thought I should know about publishing a magazine as I scribbled notes.

Next year's Perl Conference might be in Portland, Oregon which has the advantage of being the home turf of Randal Schwartz and Tom Phoenix (so expect some interesting Stonehenge promotions), but you don't have to pump your own gas.

Open Source Conference Trip Report

David H. Adler, dha@panix.com

Apple's presence at the Open Source Convention was more obvious than in previous years and not only in terms of machines in the audience. Apple was responsible for the computer room this year and had a large table in the exhibitor's room. O'Reilly has in the works a book on programming OS X with Perl.

One of the sobering facts of the current economic climate is that many people are under- or unemployed. What was very encouraging was seeing a number of people who managed to attend OSCon despite that hurdle. Such people's belief in open source and their general optimism indicate that things are far from as bleak as many might think.

Perl 6 is, naturally, on many people's minds. Damian Conway put many at ease by taking a number of existing programs and showing just how little work is needed to change them from Perl 5 to 6.

On the more conceptual side of the new Perl, Allison Randal gave an introduction to Tagmemics, a linguistic discipline that lies beneath many of the design decisions of Perl 6. Allison only had time to give the most basic description. Hopefully, in the future, she will be able to discuss Tagmemics in more depth.

In addition to their Perl 6 duties, Damian and Allison joined conference program planner Nat Torkington and former pumpking Chip Salzenberg for a session of improv comedy titled "Whose Code is it Anyway?" All the participants demonstrated that they indeed have talents beyond the keyboard and monitor.

Jos Boumans and Ann Barcomb presented CPANPLUS, an alternative to CPAN.pm. Nat Torkington recorded parodies of Apple's "Switch" ads. Rob Spier auctioned Stitch dolls and Sunnydale.pm shirts to benefit the Perl Foundation. Larry Wall based his State of the Onion address on an issue of Scientific American. The Perl Foundation presented the White Camel Awards.

New Modules

http://search.cpan.org
from July 1 to August 30, 2002

Acme-Buckaroo – Buckaroo Banzai Characters Infest Your Code
Acme-Colour – additive and subtractive human-readable colours
Acme-Magpie – steals shiny things
Acme-Morse-Audible – Audio(Morse) Programming with Perl
Acme-No – makes no() work the way I want it to
Acme-Pr0n – expose the naughty bits of modules to the world
Acme-Tao – enforce proper respect for the Tao
Acme-Tie-Handle-Blah – emulates /dev/zero, /dev/null and /dev/urandom
activitymail – CVS activity notification
AI-Categorizer – Automatic text categorization
AI-DecisionTree – Learns decision trees
AI-jNeural
Algorithm-Bucketizer – Distribute sized items to buckets with limited size
Algorithm-Huffman – implements the Huffman algorithm
Algorithm-MarkovChain-GHash – Markov chain generator, glib/C storage
Algorithm-MarkovChain – Markov chain generator
Alzabo – data modelling tool and RDBMS-OO mapper
Apache-AxKit-Language-XSP-ObjectTaglib – Helper for OO Taglibs
Apache-AxKit-Provider-DOM – Base Class For Parsed XML Providers
Apache-AxKit-Provider-PodSAX – Dynamically serve POD as XML
Apache-Blog – weblog handler
Apache-Clean – interface into HTML::Clean
Apache-CookieToQuery – Rewrite query string by adding cookie information
Apache-CustomKeywords – Customizable toolbar for MSIE
Apache-Dynagzip
Apache-FileManager
Apache-ForwardedFor – Set remote_ip to client's ip behind a reverse proxy
Apache-Giza – Giza mod_perl handler
Apache-Mailtrack – keep track of views of HTML newsletters
Apache-SearchEngineLog – Logging of terms used in search engines
Apache-Session-BerkeleyDB – implementation of Apache::Session
Apache-Session-Generate-AutoIncrement – Use monotonically increasing IDs
Apache-Session-Serialize-Dumper – Zip up persistent data
Apache-Session-Serialize-SOAPEnvelope – serialize as SOAPEnvelope
Apache-Session-Serialize-YAML – use YAML for serialization

Apache-SessionManager – simple mod_manage sessions over HTTP requests
Apache-XBEL – mod_perl to transform XBEL into exciting and fooy HTML documents
App-Info – Information about software packages on a system
AppConfig-Std – subclass of AppConfig that provides standard options
Array-Unique – allow only unique values
Astro-IRAF-CL – Perl interface to the IRAF CL interactive session
Astro-STSDAS-Table – access STSDAS format table files
Audio-Mad – Perl interface to the mad MPEG decoder library
Authen-Krb5-Admin – MIT Kerberos 5 admin interface
Authen-NTLM – NTLM related computations
Authen-Perl-NTLM
Authen-SimplePam – Simple interface to PAM authentication
AxKit-XSP-WebUtils – Utilities for building XSP web apps
AxKit-XSP-Wiki – XSP based Wiki clone
Beautifier – pretty printing perl code
Bot-BasicBot – simple irc bot baseclass
Bot-Pluggable – plugin based IRC bot
Bundle-Cobalt – Load modules for Cobalt administration
Bundle-Everything – All of CPAN
Bundle-InterchangeKitchenSink – Most all the modules for Interchange
Bundle-Interchange – Modules nice to have for Interchange
Business-OnlinePayment-2CheckOut
Business-OnlinePayment-eSec
Business-OnlinePayment-SurePay
Business-US_Amort – class encapsulating US-style amortization
Cache-DB_File – Memory cache which, when full, swaps to DB_File database
Calendar-Simple – create simple calendars
Carp-Assert-More – convenience wrappers around Carp::Assert
CGI-AIS-Session – manage sessions via AIS
CGI-AppToolkit – OO application development framework
CGI-Listman – easily managing web subscribtion lists
CGI-pWiki – Perl Wiki Environment
CGI-Widget-Tabs – Create tab widgets in HTML
Chart-Plot-Annotated – Subclass of Chart::Plot for text annotation of data-points
Chess-ICC – manipulate the Internet Chess Club from the command line
Class-ArrayObjects – utility class for array based objects
Class-Composite – Composite patterns
Class-Container – Glues object frameworks together transparently
Class-DBI-Pager – Pager utility for Class::DBI
Class-DBI-Pg
Class-Decorator – Attach additional responsibilites to an object. A generic wrapper
Class-DispatchToAll – dispatch a method call to all inherited methods
Class-Dynamic – Rudimentary support for coderefs in @ISA
Class-Maker – classes, reflection, schema, serialization, attribute- and multiple inheritance
Class-Null – Null Class pattern
Class-Privacy – object data privacy
Class-PseudoHash – Pseudo-Hash via overload
Class-PublicPrivate – Class with public keys with any name and a separate set of private keys
Cobalt-Admin
Compress-LeadingBlankSpaces – Perl class to compress leading blank spaces in (HTML, JavaScript, etc.) web content
Config-Auto – Magical config file parser
Convert-Braille – Package to convert between Braille encodings
Convert-Ethiopic-Lite
Convert-GeekCode – Convert and generate geek code sequences
Crypt-CAST5_PP – CAST5 block cipher in pure Perl
Crypt-CFB – Encrypt in Cipher Feedback Mode
Crypt-Ctr – Encrypt in Counter Mode
Crypt-Mimetic – Crypt a file and mask it behind another file
Crypt-Tea
CSS-SAC – SAC CSS parser
CVSUtils
Data-Hash-Flatten – isomorphic denormalization of nested HoH into AoH
Data-Page-Tied – Tied interface to Data::Page
Data-Page – help when paging through results
Data-Pivoter – Pivot / cross tabulation of data
Data-Pointer – C pointers for perl data types
Data-Serializer
Data-TableAutoSum – Table that calculates the

results of rows and cols automatic
Data-Taxi – Taint-aware, XML-ish data serialization
DBD-PgPP – Pure Perl PostgreSQL driver for the DBI
DBIx-FullTextSearch – Indexing documents with MySQL as storage
DBIx-OracleLogin – takes a string and splits out individual login information (user id, Oracle sid, and password) to be used in a DBI->connect() statement when connecting to an Oracle database
DBIx-Pager – SQL paging helper
DBM-Any – OO interface to AnyDBM_File
DCE-Perl-RPC – protocol composer and parser
DCE-RPC – protocol composer and parser
Debug-FaultAutoBT – Automatic Backtrace Extractor on SIGSEGV, SIGBUS, etc
Devel-Caller – meatier versions of caller
Devel-Carnivore – Spy on your hashes (and objects)
Devel-LexAlias – alias lexical variables
Devel-ModInfo
Devel-StealthDebug – Simple non-intrusive debugging
Devel-TraceSubs
Device-Audiotron – interface with the Audiotron API
Digest-Nilsimsa – Nilsimsa code
Digest-Tiger – implements the tiger hash
DirDB – use a directory as a database
Disassemble-X86 – Disassemble Intel x86 binary code
DocSet – documentation projects builder in HTML, PS and PDF formats
DPKG-Tools
Encode-HanConvert – Traditional and Simplified Chinese mappings
Encode-HanExtra – Extra sets of Chinese encodings
enum-fields – defining constants for use in Array-based objects
Exception-Class-DBI – DBI Exception objects
Expect-Simple – wrapper around Expect
Exporter-Cluster – easy multiple module imports
ExtUtils-AutoInstall – Automatic install of dependencies via CPAN
ExtUtils-Constant – generate XS code to import C header constants
ExtUtils-FakeConfig – overrides some configuration values
ExtUtils-XSBuilder
Fido – send SMS messages to Fido phones
File-Find-Rule – Alternative interface to File::Find
File-Info – Store file information persistently for fast lookup
File-Iterator – iterate across files in a directory tree
File-Random – randomly select of a file
FileHandle-Deluxe – file handle with a lot of extra features
FileHandle-Rollback – FileHandle with commit and rollback
FileMetadata
Finance-Bank-Commonwealth – Front-end to netbank.com.au
Games-AIBots – improved clone of A.I.Wars
Games-Chomp – Playing Chomp and calculating winning positions
Games-Goban – Board for playing go, renju, othello, etc
Games-Maze – Create Mazes as Objects
GD-Polyline – Polyline object and Polygon utilities
Geo-IP – Look up country by IP Address
Geo-PostalCode – Find closest zipcodes, distance, latitude, and longitude
gmuck – The Generated MarkUp Checker
Graph-ReadWrite
Graphics-ColorPicker – Hex color numbers
Hash-Case – base class for hashes with key-casing requirements
HashBang-Lingua-Romana-Perligata
HashBang-ParrotScript
HashBang-YourBrainForOnceDude
HashBang – Write your own language interpreters
Hook-Scope – adding hooks for exiting a scope
HTML-Defaultify – Pre-fill default values into an existing HTML form
HTML-Mason-Request-WithApacheSession – Add a session to the Mason Request object
HTML-RSSAutodiscovery – retreive RSS-ish information from an HTML document
HTML-TableContentParser – Do interesting things with the contents of tables
HTML-TableExtractor – Do stuff with the layout of HTML tables
HTML-TagReader
HTML-TokeParser-Simple – easy to use HTML::TokeParser interface
HTML-XSSLint – audit XSS vulnerability of web pages
HTTP-Size – Get the byte size of an internet resource
IChing-Hexagram-Illuminatus – IChing hexagram

Image-Filter – Apply filters onto images
Image-Magick-Thumbnail – produces thumbnail images with ImageMagick
Image-Shoehorn-Gallery – generate smart HTML slideshows from a directory of image files
Image-Shoehorn – massage the dimensions and filetype of an image
Image-Thumbnail – GD/ImageMagick thumbnail images
Imager-Album – processing Images for output to web
IMDb – query the Internet Movie Database
interface – simple compile time interface checking for OO Perl
IO-File-Log – IO::File abstraction on logging files
IO-NonBlocking – OO interface to non-blocking IO server implementation
IPA – Image Processing Algorithms
IPC-Locker – Distributed lock handler
IPTables
JavaScript-Toolbox
LaBrea-Tarpit
LCC – Content Provider Modules for the Local Content Cache system
Lingua-CS-Num2Word – number to text convertor for czech. Output text is in iso-8859-2 encoding
Lingua-DE-Num2Word – positive number to text convertor for german. Output text is in iso-8859-1 encoding
Lingua-ES-Silabas
Lingua-FI-Transcribe – Finnish transcription
Lingua-JA-Romaji – romaji and kana conversion
Lingua-NL-Numbers – Convert numeric values into Afrikaans
Lingua-Num2Word – wrapper for number to text conversion modules of various languages in the Lingua:: hierarchy
Lingua-Phoneme – MySQL-based accent-lookups
Lingua-RU-Number – Converts numbers to money sum in words (in Russian roubles)
Lingua-Sinica-PerlYuYan – Use Chinese to write Perl
Lingua-TR-Hyphenate – hyphenator for Turkish
Lingua-ZH-CCDICT – interface to the CCDICT Chinese dictionary
Lingua-ZH-CEDICT – Interface for CEDICT, a Chinese-English dictionary
Linux-Pid – Get the native PID and the PPID on Linux
List-Compare – Simple OO implementation of standard Perl code for comparing elements of two lists
LJ-Simple – Simple Perl to access LiveJournal
Locale-Codes
Locale-Maketext-Fuzzy – Maketext from already interpolated strings
Locale-Maketext-Lexicon – Use other catalog formats in Maketext
LockFile-NetLock – FTP based locking using the FTP mkdir command
Log-Detect – Read logfiles to detect error and warning messages
Log-Dispatch-Win32EventLog – Class for logging to the Win32 Eventlog
Log-Log4perl
Log-Simple – Basic runtime logger
Mac-PropertyList – work with Mac plists
Macro – Simple code templating mechnism
Mail-Audit-Attach – Mail::Audit plugin for attachment handling
Mail-LocalDelivery – Deliver mail to a local mailbox
Mail-Miner – Store and retrieve Useful Information from mail
Mail-Procmailrc – interface to Procmail recipe files
Mail-SpamTest-Bayesian – Bayesian spam-testing
Mail-Summary – scan read your mail!
Mail-Vacation – implements vacation program
Mail-Webmail-Yahoo – Enables bulk download of yahoo.com -based webmail
Math-BigInt – Arbitrary size integer math package
Math-BigRat – arbitrarily big rationales
Math-Big – routines (cos, sin, primes, etc.) with big numbers
Math-Business-BlackScholes – Black-Scholes option price model functions
Math-Business-EMA – calculating EMAs
Math-Business-MACD – calculating MACDs
Math-Logic-Predicate – Manage and query a predicate assertion database
Math-MatrixReal-Aug – Additional methods for Math::MatrixReal
Math-MatrixSparse – sparse matrices
Math-NoCarry – no carry arithmetic
MIME-Explode – explode MIME messages
Module-Build – Build and install Perl modules
Module-CoreList – What modules shipped with versions of perl
Module-CryptSource – Encrypt and Decrypt source for Binary Packagers
Module-Signature – Module signature file manipulation
Money-ChangeMaker – make change based on a

monetary quantity
MPE-Process – MPE Process Handling
MPE-Spoonfeed – spoonfeeding commands through a message file to MPE programs run as child process
MySQL-Config – Parse /etc/my.cnf and /.my.cnf
MySQL-Diff
MySQL-Easy – make your base code kinda pretty
Mysql-NameLocker – Safe way of locking and unlocking MySQL tables using named locks
Net-Arping – Ping remote host by ARP packets
Net-DHCP
Net-DNS-SEC
Net-Gadu – Interfejs do biblioteki libgadu.so
Net-Rsh – perl client for Rsh protocol
Net-Socket-NonBlock
Net-Whois-RIPE
Netscreen
Neuron – networks
News-Article – Object for handling Usenet articles in mail or news form
News-Collabra
NewsTicker
NexTrieve – Perl interface to NexTrieve search engine software
Oak-Web
Object-Iterate – iterators for objects that know the next element
OpenFrame-AppKit – The OpenFrame AppKit
optimize – Pragma for hinting optimizations on variables
Palm-ListDB – Handler for ListDB databases
PApp-Recipe – blah blah blah
Parallel-Jobs – run jobs in parallel with access to their stdout and stderr
Params-Validate – Validate method/function parameters
Parse-AccessLogEntry – Parse one line of an Apache access log
Parse-Earley – Parse *any* Context-Free Grammar
Parse-RecDescent-Deparse – Turn anobject back into its grammar
PDF-API2-UniMap
PDF-API2 – The Next Generation API for creating and modifing PDFs
PDL-Sparse
PerlIO-via-Base64
PerlIO-via-Include – include other files
PerlIO-via-LineNumber – prefix line numbers
PerlIO-via-MD5 – Create an MD5 digest of a file
PerlIO-via-Pod – Extract plain old documentation
PerlIO-via-QuotedPrint
PerlIO-via-Rotate – encode using rotational deviation
PerlIO-via-StripHTML – strip HTML tags from an input file
PerlIO-via-UnComment – remove comments
PerlIO-via-UnPod – removing POD
Petal – Perl Template Attribute Language
Pixie – The magic data pixie
Pod-HtmlHelp – Interface with Microsoft's HtmlHelp system
POE-Component-Client-DNS – DNS client component
POE-Component-Client-HTTP – HTTP user-agent component
POE-Component-Client-Ping – ICMP ping client component
POE-Component-DirWatch – directory watcher
POE-Component-IRC-Object – slightly simpler OO interface to PoCoIRC
POE-Component-LaDBI – spawns a perl subprocess for non-blocking access to the DBI API
POE-Component-MPG123
POE-Component-RRDTool
POE-Filter-Ls – translates common ls formats into a hashref
POE-Session-Cascading – Stack-like Sessions
POE – multitasking and networking framework for perl
PPresenter – Slide shows written in Perl (or XML)
PPrint – Programmable sprintf. Allows you to associate functions to directives and supplies fairly powerful default directives
Prim – Perl Remote Invocation of Methods (RMI and EJB's for Perl, sort of)
Qmail-Control – interfacing with Qmail's control files
Readonly – Facility for creating read-only scalars, arrays, hashes
Reuters-SSL – Reuters SSL Source Sink Library
RPM-Specfile – creating RPM Specfiles
RPM2 – Perl bindings for the RPM Package Manager API
RTF-Tokenizer – Tokenize RTF
Schedule-Load – Load distribution and status across multiple host machines
Scrape-USPS-ZipLookup – Standardize U.S. postal addresses
Set-CheckList – Keep track of a list of to do items
ShiftJIS-X0213-MapUTF – convert between Shift_JISX0213 and Unicode
Shishi-Prototype – Internal use prototype for the

Shishi regex/parser
Slash-OurNet
SOAP-MIME – Patch to SOAP::Lite to add attachment support. Currently all that is supported the retrieval of attachments from a response
SQL-Generator – Generate SQL-statements
SQL-Snippet – Conatraint-based OO Interface to RDBMS
StateMachine-Gestinanna – provides context and state machine for wizard-like applications
Statistics-Contingency – Calculate precision, recall, F1, accuracy, etc
Statistics-DEA – Discontiguous Exponential Averaging
String-Substrings – extract some or all substrings from a string
Sub-Curry – curry functions
Sub-Lexical – implements lexically scoped subroutines
SVGGraph – creating SVG Graphs / Diagrams / Charts / Plots
SWF-Search – Extract strings and information from Macromedia SWF files
Sys-Headers – Perl interface to system headers
Sys-Mmap
SystemPerl – SystemPerl Language Extension to SystemC
TDB_File – Perl access to the trivial database library
TEI-Lite
Template-Plugin-Clickable – Make URLs clickable in HTML
Template-Plugin-Comma
Template-Plugin-FillInForm – TT plugin for HTML::FillInForm
Template-Plugin-Number-Format – Plugin/filter interface to Number::Format
Template-Plugin-Page – plugin to help when paging through sets of results
Template-Plugin-PerlTidy – Perl::Tidy filter
Template-Plugin-TextToHtml – Plugin interface to HTML::FromText
Term-ANSIScreen – Terminal control using ANSI escape sequences
Term-WinConsole – text based windows management
Test-Class – Easily create test classes in an xUnit style
Test-Extreme – perlish unit testing framework
Test-HTTPStatus – check an HTTP status
Test-Manifest – interact with a t/test_manifest file
Test-ManyParams – test many params as one test
Test-Pod – check for POD errors in files
Test-Reporter – reports test results to the CPAN testing service
Test-Warn – test methods for warnings
Text-Flowed – text formatting routines for RFC2646 format=flowed
Text-Glob – match globbing patterns against text
Text-MessageFormat – Language neutral way to display messages
Text-PhraseDistance – measure of the degree of proximity of 2 given phrases
Text-WagnerFischer – implementation of Wagner-Fischer edit distance
Text-WikiFormat-SAX – Wiki text parser
Thread-Conveyor-Monitored – monitor a belt for specific content
Thread-Conveyor – transport of any data-structure between threads
Thread-Needs – remove unneeded modules from CLONEd memory
Thread-Pool-Resolve – resolve logs asynchronously
Thread-Pool – group of threads for performing similar jobs
Thread-Queue-Any-Monitored – monitor a queue for any specific content
Thread-Queue-Any – thread-safe queues for any data-structure
Thread-Queue-Monitored – monitor a queue for specific content
Thread-Serialize – serialize data-structures between threads
Thread-Tie – tie variables into a thread of their own
Thread-Use – use a module inside a thread only
Tie-CSV_File – ties a csv-file to an array of arrays
Tie-InsertOrderHash – order-preserving tied hash
Time-Piece-MySQL – Adds MySQL-specific methods to Time::Piece
Tk-CanvasFig – Tk::Canvas methods for dealing with figs
Tk-CheckbuttonGroup – widget displays and manages a group of related checkbuttons
Tk-HideCursor – Hide the cursor when it passes over your widget
Tk-RadiobuttonGroup – displays and manages a group of related radiobuttons
Tk-SearchDialog – Dialog for Perl/Tk Text
Tk-StayOnTop – Keep window in foreground
Tk-Workspace – text processor
Tk-WorldCanvas – Autoscaling Canvas widget
Trinket – object persistence and lookup framework

types – pragma for strict type checking
Unicode-Collate – Unicode Collation Algorithm
Unix-Conf-Bind8
Unix-Conf – Front end for class methods in various utility modules under the Unix::Conf namespace
Validate-Net – Format validation and more for Net:: related strings
VCS-CMSynergy – interface to Telelogic CM Synergy
VCS – generic Version Control System access in Perl
Verilog-Perl
Video-DVDRip – GUI for copying DVDs, based on an open Low Level API
Win32-Girder-IEvent
Win32-Scanner-EZTWAIN – interface to EZTWAIN library
Win32-ToolHelp – information about currently executing applications
Win32API-ProcessStatus – obtaining information about processes using the plain Win32 PSAPI
Win32API-Process – handling the processes using the plain Win32 API
Win32API-ToolHelp – obtaining information about currently executing applications using the plain Win32 ToolHelp API
Winamp-Control – control winamp (over the network)
WordNet-QueryData – direct interface to WordNet database
WWW-AllMusicGuide
WWW-Search-AltaVista
WWW-Search-Ebay
WWW-Search-Google
WWW-Search-Lycos
WWW-Search – base class for WWW searches
WWW-SMS – sends SMS using service provided by free websites
WWW-Sundance – Get movie schedules/info from Sundance
X500-DN-Parser – Parse X500 Distinguished Names
XForms-Generator
XML-ASX – Create Advanced Streaming XML files for Windows Media Player
XML-DOMHandler – Implements a call-back interface to DOM
XML-Filter-BufferText – Filter to put all characters() in one event
XML-Filter-Sort – SAX filter for sorting elements in XML
XML-Filter-XML_Directory_2-Base – create XML::Directory to something else filters
XML-Filter-XML_Directory_2XHTML – SAX2 filter for munging XML::Directory::SAX output into XHTML
XML-GDOME – Level 2 DOM gdome2 library
XML-Handler-Dtd2Html – PerlSAX handler for generate a HTML documentation from a DTD
XML-RDDL – Interface to RDDL
XML-Ximple – XML in Perl

Module List Additions

compiled from modules@perl.org
from July 1 to August 30, 2002

ADT, ADT::Queue::Priority, Acme::Morse::Audible, Apache::Htaccess, Apache::Lint, Apache::SessionManager, App::Info, ConfigReader::Simple, DBD::PgPP, Debug::FaultAutoBT, File::Find::Rule, File::Searcher::Similars, Finance::Bank::Commonwealth, GD::Gauge, HTML::LinkExtractor, HTTP::SimpleLinkChecker, IOLayer::MD5, Lingua::ZH::CCDICT, Lingua::ZH::CEDICT, Lingua::Zompist, Lingua::Zompist::Barakhinei, Lingua::Zompist::Cadhinor, Lingua::Zompist::Cuezi, Lingua::Zompist::Kebreni, Lingua::Zompist::Verdurian, Linux::Pid, Log::Dispatch::Win32EventLog, MARC::Record, Mail::Vacation, Mail::Vacation::LDAP, Math::Business::BlackScholes, Net::DNS::SEC, Net::Whois::RIPE, POE::Component::Server::HTTP, Perl6::Parameters, PerlIO::Via::Base64, PerlIO::Via::MD5, PerlIO::Via::QuotedPrint, PerlIO::Via::StripHTML, RTF::Tokenizer, Set::CrossProduct, Test::Manifest, Test::Pod, Test::Reporter, Tie::Toggle, Time::Piece::MySQL, URI::Sequin, XML::OCS, XML::Twig, XML::XForms::Generator,

---

Wonder which modules came with a particular version of Perl?

```
use Module::CoreList;
my $h = $Module::CoreList::version{5.006001};
foreach my $key ( sort keys %$h  ) {
    print "$key $h->{$key}\n";
    }
```

Extreme Mowing

Andy Lester, andy@petdance.com

Abstract

I mowed my lawn yesterday. It was an exercise in good programming. Good programming practices are not a great mystical art. As the publisher of this magazine is fond of noting, they are usually just common sense packaged with a pretty name. So too it can be for you, applying the common sense skills you use outside of work to the problems that you face throughout the day.

1 The Lawn Project

I live on a corner, so my lawn is pretty big. I take at least 90 minute to mow it if I work straight through. Heavy rains over the past few days had turned it into a jungle. I planned to leave work by 5:45 pm so I would have ample time to mow before the sun went down. That did not happen.

I left work at 7:00 pm, with the sun sinking in the sky, and stopped at McDonald's to grab some protein and caffeine. I get tired and crabby if I do not have enough protein, and I figured if I did not eat on the way home, I would have to stop in the middle of mowing. The drive-thru line was long and slow, and I considered just going straight home, but I stuck it out.

At home, I put on my grubby lawn mowing shoes, checked the gas in the mower, and assessed the lawn. I knew I would not have enough time to do the entire lawn before the sun went down, but I could do some of it. Although the front yard was more visible, the back yard was thicker and would be more of a hassle later on, so that is where I started. Usually I mow the entire lawn all at once, going from the front to the sides to the back and around again, but this time I had to limit myself to the back. I set a clear mental boundary on the sides of the house that I would mow up to, and no further.

While I was mowing, my wife and daughter came home from the store. I waved, but kept at my work, making sure I did not get distracted. I knew if I stopped to say "Hi," I would get caught up in something. The girls are important, but this project was time-critical.

By 8:00 pm, the sky was dusky, but I had the back section done. I figured I had about 15 to 20 minutes left before it got so dark that I could not see, so I attacked a long strip by the street to the north. It is separated from the main lawn by the sidewalk, so the difference between mowed and not mowed would not be so obvious. After 10 minutes, I had that part done, had some time, so did the same section on the east. By the time I finished that it was too dark to do any more, so I put the mower away, happy that I had completed half the lawn.

I had applied good programming practices to mowing the lawn, including some key parts of Extreme Programming (XP). I call my methods *XM: Extreme Mowing*.

2 The basics of XM

Mowing the lawn was a large deliverable, with not enough staff to do it in the time allotted. The task itself was not much fun, so staff motivation on the project was not very high. If you have not yet been on a project like this, you are probably on your first project.

2.1 Small deliverables

The lawn project was like any programming project. It has the three constraints of the "iron triangle"—Time, Resources, and Deliverables. If those do not fit, one of them has to give. I knew I could not mow the entire lawn before the sun went down. I could not add any more resources, like adding someone else to the lawn mowing team. I had to reduce the deliverables. I could not do the entire lawn in the time allotted with just me doing it.

Zealots make much of XP and how it keys on early, frequent tests, and that most odious of argument-starters, pair programming. To me, both as a lead programmer and project manager by day, and an XM lawn mower by night, the first tenet of XP is limiting scope and creating frequent small releases. In XM, I have a similar goal. Once it is dark and I cannot mow, I do not want the lawn to look stupid. It is better to have just part of the yard done than it is to have all of the yard half done.

In XP, I want to be able to release at any time. I make sure that everything that has been done is good and working, so that if I hit some deadline, I will not have a half-working project. Even if I have not finished some sections, at least I will have sections that are done. In XM, I made sure that I never bit off a bigger section than I knew I could do in the time allotted (and in XM, there is no project extensions from the sun).

2.2 Focus

I made sure I stuck to the task at hand. I did not get side-tracked by bringing in the groceries or saying "Hi" to Amy & Quinn. I maintained focus while I was in the middle of my subtasks. Because of this focus, if I wanted to stop at the end of each of my three subtasks, I could have.

Programmers, including me, are awful at this level of focus on any project that is not fun. They will go down meandering roads of functionality, or stop with their real tasks in favor of doing ones that are more fun. It is always more fun to work on the graphing module than the data entry drudgery. Down the path of distraction lies the heartbreak of many half-done modules.

2.3 Preparation

Finally, I made sure I did the proper preparation. I could have skipped dinner, but I would have had to spend more time later on in my project. I could have not changed into my lawn shoes, but at the cost of a ruined pair of work shoes. I could have not bothered filling the mower with gas, but chances are I would have run out in the middle of a swath. It was tough to do these preparatory steps when I saw the sun setting and my project time ticking away, but I knew it would be a short-term win and a long-term loss.

3 Conclusion

Solid programming practices are part hard-won lessons and part common sense. It does not matter whether it is Agile Programming or Extreme Programming or whatever the buzzword or book topic of the month is. If I want to stick a label on it, there is no harm any more than referring to my lawn technique as XM.

4 References

Extreme Programming Installed, by Jeffries, Anderson & Hendrickson. 0-201-70842-6. Addison-Wesley, Upper Saddle River, NJ.

Model 621 mower with 6.75 HP Briggs & Stratton engine and 21-inch cutting deck. Cub Cadet. Valley City, OH.

5 About the author

Andy Lester's XM skills have been improving in the years since he bought a house in the far Chicago suburbs. During the day, he leads a team of Perl and PHP programmers on the e-commerce site for the number one vendor of school library books in the US. Sometimes he gets time to maintain his CPAN modules MARC::Record, HTML::Lint, and Carp::Assert::More.

The Lawnotron 3000 provides 21 inches of viewable screen and cutting swath. Powered by AMD and Briggs & Stratton, this baby has an optional math coprocessor and mulching blade.

A Simple Assembly Simulator in Perl

Phil Crow, philcrow2000@yahoo.com

Abstract

As a sometime instructor in Computer Science, I need a way to introduce concepts that only show up in assembly. This led me to design a small assembly language and implement it in Perl. I implemented PAL in an assembler/interpreter pair.

1 Introduction

When I studied Computer Science in the mid 1980s, assembly was a regular part of the college curriculum. C pointers made much more sense after I learned about double indirect post auto-increment. As the faculty sought more employable languages for the curriculum, assembly entered the schedules of fewer and fewer students. Yet, those concepts I learned guide much of my work as a programmer. I want a way for my students in *Introduction to Computer Science* to see at least something of the flavor of assembly so I designed the Pseudo-Assembly Language (PAL) to provide that flavor. I implemented it in both Perl and Java.

2 A First Example

I start with a simple countdown example. In Perl I might write this as a loop as in code listing 1, which I translate to PAL in code listing 2.

Code Listing 1: Countdown in Perl

```
my $i = 10;
loop: while ($i > 0) {
                print "$i\n";
                $i--;
                }
print "liftoff\n";
```

Code Listing 2: Countdown in PAL

```
i:      alloc 1
                store 10 i
loop:   brle  i 0 done
                prt   i
                subt  1 i
                jump  loop
done:   prompt "liftoff\n"
                end   0
```

Lines 1, 3, and 7 need names so I can mention them in other places. They have labels. I used the same syntax for PAL labels as Perl uses for its labels. In fact, most assemblies use this style of label.

Not all lines need labels. If I am not going to refer to the line, it probably should not have a label. I can add optional labels wherever I want. After the label comes the command. When I need to go back to the top of my loop I say `jump loop` in line 6.

One of the commands is actually an assembly directive. Directives instruct the assembler. PAL's one directive is alloc which reserves space for variables. The number after alloc directs the assembler to set aside that number of slots in memory. My i is a scalar, it needs only one slot.

The rest of the commands do something during execution. To assign a value to a variable, use store. Saying `store 5 i` in PAL translates to `$i = 5;` in Perl. I do not use operators in PAL—instead I use commands. Each command takes one or more operands. The store command takes a value and a place to put it. To print literal text in line 7 I say `prompt "some string"`. This is a real Perl string, so I use a newline if I need one. When I want the program to end I use `end 0`. If I need to report an error code to the operating system, I put it in place of 0 in the end statement. The end command wraps Perl's exit function.

To manage a loop, PAL provides conditional and unconditional branches (and people think goto is harmful). For a pretest loop, the conditional branch goes at the top. PAL provides six of these branch commands. The line `loop:  brle i 0 done` says, "go to done if i <= 0." The comparison operator is actually part of the command name. To read the statement, break the operator off of the command name and insert it between the first two operands. I use the same letters as the Perl string comparisons. Table 2 shows the PAL comparison operators.

Table 1: PAL Comparison Commands

command	**operator**
brle	<=
brlt	<
brge	>=
brgt	>
breq	==
brne	! =

PAL input and output is simpler than most other assemblies. I like this for teaching. To see the value of one or more variables, simply say `prt var1 var2 ...`. I get a newline at the end whether I want it or not.

PAL performs arithmetic using add, subt, mult, and div. They all have the same semantics. In PAL `subt 1 i` means what `$i -= 1;` does in Perl. Table 2 shows the PAL math commands. Arithmetic may cause some initial confusion for students, since the receiving variable comes last instead of first.

Table 2: PAL Math Commands

command	**effect on op1 and op2**
add op1 op2	op2 += op1
subt op1 op2	op2 -= op1
mult op1 op2	op2 *= op1
div op1 op2	op2 /= op1

3 Arrays

An array is really one named memory slot followed by several unnamed slots. To get to a slot, I go to the first slot (the one with the name) and index into the array to the slot I want. For a higher-level language the compiler works this out for me. If a C array named grades starts in address 0x918b and I refer to `grades[2]`, the compiler translates this into address 0x918b + 2. In PAL I have to do this offset work myself. The example in code listing 3 reads in 5 grades, stores them in an array, then divides each one by the highest score, and then prints the results.

Code Listing 3: Grading program

```
 1  top:            alloc 1
 2  grades:         alloc 5
 3  gradesind:      alloc 1
 4  i:              alloc 1
 5          store   0               i
 6          store   0               top
 7          store   &grades         gradesind
 8  loop:   brge    i               5       output
 9          take    @gradesind
10          brle    @gradesind      top     finally
11          store   @gradesind      top
12  finally:incr    gradesind
13          incr    i
14          jump    loop
15  output: store   0               i
16          store   &grades         gradesind
17  loop2:  brge    i       5       done
18          div     top     @gradesind
19          prt     @gradesind+
20          incr    i
21          jump    loop2
22  done:   end     0
```

I can use incr (or decr) to add (or subtract) one to my variable. I could just as easily use add 1 (or subt 1). The take command takes one line from standard input and puts it into its argument. For practical purposes, PAL stores all variables as floats.

To get an array I just tell alloc how many slots I want. The array will not grow. Now I need some way to keep track of my place in the array. I used the gradesind variable for that. At the top of each array walking loop, I set gradesind to the address of grades by using the ampersand modifier. To use that address as a pointer, I put the at sign in front of it. This may look like Perl arrays, but in Macro-11 it means follow this address as a pointer.

In the second loop, labeled loop2 in line 17, I avoided a whole statement by incrementing the pointer as I access it, `prt @gradesind+`, instead of using two commands to do the same thing:

```
prt     @gradesind
incr    gradesind
```

The plus sign in `prt @gradesind+` tells PAL to increment the pointer after using it. In Perl I might say something like `print "$some_array[$index++]\n"`.

4 Unrolling Calculations

The following formula represents a paycheck calculation program. For simplicity I assume that everyone works at least 40 hours (and if someone does not, the formula penalizes them!).

```
pay = 40 * rate + 1.5 * (hours - 40) * rate
```

PAL arithmetic is different in two key ways from higher-level languages. First, operations are binary. Second, operations overwrite the value of one of the operands. I need more statements and more variables to convert to PAL, as shown in code listing 4. The end of line comments show the corresponding Perl statement.

Code Listing 4: PAL pay calculation

```
store 40    pay     # pay = 40;
mult  rate  pay     # pay *= rate;
store hours tmp     # tmp = hours;
subt  40    tmp     # tmp -= 40;
mult  1.5   tmp     # tmp *= 1.5;
mult  rate  tmp     # tmp *= rate;
add   tmp   pay     # pay += tmp;
```

I can do this with a little patience and care. For each operator, make a pair of PAL statements and introduce one temporary variable. When I finish, I combine the statements if I can. When I first wrote the above, I used three temporary variables. By rearranging the statements, I ended up with only one extra variable. An optimizing compiler would follow a similar plan.

5 Routines in PAL

When a section of code starts to obscure the flow of the larger program, I move that code into a subroutine. This also allows me to call the routine from multiple places in my program. PAL provides a rudimentary system for this type of extraction.

At any point in the code, I can insert the gosub command whose single argument matches the label of the line where the subroutine starts. At the end of the subroutine, I can insert the ret command to send control back to the caller. Keep in mind that PAL provides only global variables. This severe restriction makes PAL much simpler to describe and implement. A compiler could provide scope for the variables in PAL subroutines.

The PAL program in code listing 5 computes the factorial of 5 using recursion. Code listing 6 is the Perl equivalent. Each repeatedly calls a fact subroutine that takes its input, multiplies the global variable answer by the input, decrements the input, then calls the fact subroutine again. After each program computes the answer it prints it to standard output and exits.

Code Listing 5: PAL factorial

```
answer:  alloc 1
number:  alloc 1
         store   1       answer
         store   5       number
         gosub   fact
         prompt  "5 factorial is "
         prt     answer
         end     0
fact:    brle    number  1       done
         mult    number  answer
         decr    number
         gosub   fact
done:    ret
```

Code Listing 6: Perl factorial

```
my $answer;
my $number;
$answer = 1;
$number = 5;
fact();
print "5 factorial is ";
print "$answer\n";
exit(0);
sub fact {
        if ($number > 1) {
                  $answer *= $number;
                  $number--;
                  fact();
                }
        return;
        }
```

I do not need to use recursion to find the factorial, but this example does show that PAL is happy to recurse. It has a stack to keep track of where ret should send program execution. PAL has no local variables, which severely limits the usefulness of function calls and recursion.

6 Summary of PAL commands

PAL has 22 commands, one of which is a compiler directive. Table 3 shows the commands grouped by type.

Table 3: Summary of PAL commands

Arithmetic	I/O	Flow Control	Memory Directive	Debug
add	prt	brle	alloc	shower
subt	prompt	brlt		
mult	take	brge		
div		brgt		
store		brne		
incr		breq		
decr		jump		
		gosub		
		ret		
		end		

The shower command takes no arguments and dumps out the current program segment at the point where it appears, showing all of the variables with their values and all lines of code. Code listing 7 show the output for the grading program in code listing 3.

Code Listing 7: Output of shower

```
 0 top:       50
 1 grades:    0.9
 2            1
 3            0.7
 4            0.5
 5            0.96
 6 gradesind:        6
 7 i:         5
 8            store 0 i
 9            store 0 top
10            store &grades gradesind
11 loop:      brge i 5 output
12            take @gradesind
13            brle @gradesind top finally
14            store @gradesind top
15 finally:   incr gradesind
16            incr i
17            jump loop
18 output:    store 0 i
19            store &grades gradesind
20 loop2:     brge i 5 done
21            div top @gradesind
22            prt @gradesind+
23            incr i
24            jump loop2
25 done:      shower
26            end 0
The current program pointer is 25
```

7 Internals of the assembly Perl program

7.1 The assembler

When I invoke the assembly program, it reads my source one line at a time, discarding comments and blank lines. If it keeps the line it stores the label (if any) in the symbol table hash keyed by the labels storing the corresponding line number. PAL resolves every label at run time by checking its symbol table hash. It does not need to worry about forward references (a reference that refers to a label not yet encountered).

After dealing with the label, PAL breaks the line into pieces on whitespace. PAL uses the first element as the command. If I use an invalid command, PAL will not notice until run time.

The PAL assembler needs only three command handlers, one for alloc directives (since they are not run time commands), one for prompt (since I do not want my strings split), and one for all other commands.

For alloc, PAL merely initializes the proper number of slots in the program segment array to 0 and moves the starting program pointer up if the allocation happens before the first line of code.

Commands and variable values share the program segment array. Each line of code takes one slot in the segment; each variable takes the number of slots I give as alloc's argument.

For prompt commands, PAL constructs a Perl subroutine call `prompt(STRING);`

For all other commands, PAL constructs a subroutine call from the input with

```
shift(@tmp) . "('" . join("', '", @tmp) . "');";
```

where @tmp received the results of splitting my command (after any label was removed).

For all commands, PAL puts the subroutine call into the proper slot in the program segment array. The whole assembly phase takes about 35 lines of Perl code.

7.2 The interpreter

Once PAL builds the program segment array, execution begins in an infinite loop. In code listing 8, PAL tries to use the current program pointer as a slot number in the program segment array (called commands). If it cannot, the program pointer fell out of the program. If it can, PAL evals the statement in that slot and Perl decides whether the command is valid. If the program pointer moves into the part of the commands array where the variable values live, PAL's eval call causes immediate death. A genuine assembly would try to execute from the data, causing unpredictable results. This would lead to self-modifying code, which is severely risky. PAL does not permit this. At the bottom of the loop, PAL bumps the program pointer up by one.

Code Listing 8: PAL interpreter loop

```
while (1) {
        if ($commands[$progpointer]) {
                eval $commands[$progpointer]; die $@ if $@;
                }
        else {
                die "Program pointer has moved to undefined command $progpointer.\n";
                }
        $progpointer++;
        }
```

Each PAL command is a Perl subroutine defined in the PAL interpreter. Code listing 9 shows the Perl implementation of the PAL add command.

Code Listing 9: Perl implementation of PAL add

```
sub add {
        my ($val, $loc) = @_;
        my $actual_val = _massageloc($val, "retrieve");
        _massageloc($loc, "add", $actual_val);
        }
```

The _massageloc function handles indirect addressing. The first operand is a literal, a variable, or an indirect variable (PAL does not support double indirection). The second operand can be all of those except literal—it must be an lvalue. Both of these might need pre- or post-incrementing.

The symbols hash houses the symbol table. The program dies if it tries to branch to an undefined symbol. If the symbol exists, PAL pushes the current program pointer onto the program pointer stack (ppstack) as in code listing 10. One line 6, PAL decrements the new program pointer to one less than the line the programmer wanted because the last line of the interpreter loop (code listing 8) bumps the program pointer by one.

Code Listing 10: Perl implementation of PAL dispatcher

```
1  sub gosub {
2          my $branch_name = shift;
3          my $branch = $symbols{$branch_name};
4          die "Invalid gosub label in command $progpointer.\n" unless $branch;
5          push @ppstack, $progpointer;
6          $progpointer = $branch - 1;
7          }
```

Return fails if PAL did not see a prior gosub command. Otherwise, PAL pops the program pointer stack to retrieve the return location. PAL does not need to change this value as it did in the gosub function. Control correctly passes to the line following the most recent gosub command. Code listing 11 shows the Perl implementation of the ret command.

Code Listing 11: Perl implementation of PAL ret command

```
sub ret {
        my $oldptr = $progpointer;
        $progpointer = pop @ppstack;
        die "Attempt to return without prior gosub at command $oldptr\n"
          unless $progpointer;
        }
```

7.3 The debugger

PAL provides a debugger. To run in debug mode, I use the -s switch to turn on single-stepping and the -t flag to turn on tracing.

```
perl assembly -st palprogram
```

go - turn off tracing and single stepping; execution proceeds normally.

go lab - turn off tracing and single stepping until PAL reaches a command with label "lab". If PAL cannot find "lab" in the symbol table works like go.

q (or Q or quit) - stop execution immediately.

show var - print the value of the specified variable, which must be in the symbol table.

shower - print a current listing of the program segment, as if the shower command appeared at that point of the source code.

8 References

My guide to PDP-11 assembly was provided by a course taught well by Larry Haffey of Mid-America Nazarene College using the book *Macro-11 Assembly Language* by C. Jinshong Hwang and Darryl E. Gibson, Prentice Hall, 1986.

For more information on PAL commands, see the documentation inside the assembly program in my CPAN directory, or in the CPAN scripts archive.

http://www.cpan.org/authors/id/P/PH/PHILCROW/assembly-0.21.plt

http://www.cpan.org/scripts/Educational/ComputerScience/index.html

What Perl Programmers Should Know About Java

Beth Linker, blinker@panix.com

Abstract

The Java platform is by no means a replacement for Perl, but it can be a useful complement. Even if you do not need to or want to use Java, you should know a bit about it and when you might choose Java or Perl for a project.

1 Description

What should Perl programmers know about Java? It depends. Not knowing Java will not slow you down as a Perl programmer. However, Java is a popular and rapidly growing language, so there's a good chance that you will find yourself using Java systems or writing applications that need to interact with them. In some situations, writing a Java application may be a good way to accomplish something that you cannot easily do in Perl.

This article is not a Java language tutorial. There are plenty of those available, both in books and online (see section 6). Instead, I want to present Java as a development platform with a special emphasis on those aspects of Java that are most complementary to Perl. If you are aware of Java's strengths and weaknesses, you will know when it is potentially worth learning and when you are better off sticking with Perl.

2 What is Java?

Java is an object-oriented cross-platform programming language. Syntactically, Java looks a lot like C and C++. However, Java differs significantly from these languages in that it provides many of the high-level features that you are used to from Perl, such as automated garbage collection and memory allocation. Over the past 5 years, Java has become a popular language for introductory computer science classes because teachers can use it to teach high-level programming constructs before students are ready to wrestle with the details of memory management.

The most important feature of Java is the Java Virtual Machine (JVM), which functions as a "sandbox" or protected area in which Java applications are executed. This makes Java a safe language for applications that run over a network, because the JVM can prevent Java applications from performing hostile operations like deleting files. This is why you seldom hear about Java viruses. Java source code is pre-compiled into Java class files that contain a cross-platform Java bytecode, which are then interpreted by the JVM at runtime. Java applications are often distributed as .class files or in compressed form as Java ARchive (JAR) files.

The JVM can run as a stand alone application, a component of another application, or a plug-in for a web browser. Anyone can build a JVM using Sun's Java Virtual Machine specification. There are several JVMs available for Windows and Unix, including a popular one from IBM. Many other companies have built their own JVMs for proprietary or less mainstream platforms. Ideally, a Java application will be able to run on any system that has a JVM.

3 A Brief History of Java

Java got its start in the research labs at Sun Microsystems and was first unleashed on the world at large in 1995. When Sun first released Java, its promise was “write once, run anywhere”, meaning that you could easily port Java applications to multiple platforms. While early Java applications did run on a variety of systems, building cross-platform GUIs that were true to each system’s native look and feel proved to be a major challenge. Java’s performance was also an issue because, like Perl, it is an interpreted language. The first version of Java also introduced applets–small client-side applications that could be embedded within Web pages and run by a JVM plug-in within the browser.

As adoption of Java increased, Sun continued to improve the language and its many supporting libraries. In 1998, the Java 2 Platform was launched. It made significant improvements to the original Java, including the introduction of the “Swing” GUI classes, which are now used in most Java desktop applications. Just to keep things confusing, Java 2 began with version 1.1 of the Java Development Kit (JDK) and has continued up to the current version, JDK 1.4. Each version of the JDK introduced new APIs and core Java libraries. The JDK is the heart of what is often referred to as the Java 2 Standard Edition, or J2SE. Java applets and desktop applications are usually developed using only J2SE.

In addition to the J2SE, there are two other important components of the Java platform. The Java 2 Enterprise Edition, or J2EE, is Java’s platform for enterprise computing. It is a complement of the Java 2 Standard Edition, not a substitute. It includes Web development standards and APIs such as Java Servlets and Java Server Pages (JSP), as well as distributed computing tools like Enterprise Java Beans (EJB) and Java Message Service (JMS). Web applications, corporate information systems, and other large-scale projects are often developed using a combination of J2SE and J2EE.

The Java 2 Micro Edition (J2ME), launched in 2000, is a stand alone version of Java developed for small devices. Because Sun always intended Java to run on a wide variety of processors, and they originally developed it for embedded computing, J2ME is a natural step in the evolution of Java. J2ME is a scaled-down version of J2SE, with many APIs removed and some new ones added. It is optimized for creating small applications that require few resources and can easily run on cell phones, PDAs, and other small devices.

3.1 How Java Evolves

Sun Microsystems owns Java, so it controls the Java language and its licensing. In addition to determining what goes into the core Java libraries (the JDK), Sun also certifies Java implementations developed by other companies after testing to make sure that they are proper implementations of the platform. This is an important aspect of maintaining Java’s portability. In 1997, Sun filed a lawsuit against Microsoft in which they alleged that Microsoft’s implementation of Java was incompatible with non-Microsoft systems. With the exception of that dispute (which was settled in January, 2001), Java licensing has not been a major source of contention.

While Sun provides the majority of resources to develop Java, other companies also participate through the Java Community Process (JCP). Companies and individuals can join the JCP and participate in the drafting of Java Specification Requests (JSRs). Members of the public are able to review and comment on JSRs before they become official Java standards. Many important parts of the Java platform, including J2ME, Java Servlets, and Java’s XML libraries have been developed through the Java Community Process.

4 A Glimpse of the Java Programming Language

As a Perl programmer, your first impression of Java source code may be that it is terribly verbose. You would be right. The only place you will encounter Java Golf is at resorts in Indonesia. To illustrate, I show the standard Hello World program written in Perl in code listing 12 and the same program written in Java in code listing 13.

Code Listing 12: Perl Hello World

```
#!/usr/bin/perl
print "Hello world!\n";
```

Code Listing 13: Java Hello World

```
public class HelloWorld extends Object{
        public static void

        main(String[] args){
                System.out.println("Hello world!");
                }
        }
```

The Java version requires you to put even the simplest code into a class structure. Our class, HelloWorld, extends the java.lang.Object class (the words “extends Object” could have been omitted in this example, because Object is the default base class for any Java classes that do not have a base class explicitly specified).

The HelloWorld class contains a static method called “main” that takes an array of Strings as input. Java looks for the “main” method when you type “java HelloWorld” on the command line (after compiling HelloWorld.java into a bytecode file named HelloWorld.class). At least one class in an application must have a main() method with this signature so that it can serve as the entry point into the application. This class contains a single method implementation, but classes can also contain constant and variable declarations as well as special methods called constructors that are used to create new instances of the class.

This example’s print statement is also much longer than its Perl counterpart. Perl’s print function defaults to the current system output stream. In Java, the system output stream is a PrintStream object named “out” that is attached to the application’s System object and I need to reference it explicitly.

4.1 How Java Applications are Organized and Distributed

All Java applications consist of one or more classes. You can organize classes into packages. A package is a group of classes that reside in the same directory. Java organizes classes hierarchically. For example, all packages in the core J2SE library begin with the name “java”. The Java string class is named “java.lang.String” and the Java equivalent of a FILE pointer is named “java.io.File”.

Applications classes usually have their own hierarchies. By convention, publishers of Java software name their class hierarchy after its home on the Internet. For example, Java classes released by the Apache Software Foundation belong to the “org.apache.*” hierarchy. This naming convention minimizes the risk of naming conflicts when an application combines modules from different sources. While two classes may have the same name, their fully qualified package names will be different.

The other important function of packages is that you can use them to control access to parts of a class. While you can declare a method or variable in a class “public” (accessible by all) or “private” (accessible by none), the default is to make it accessible to other classes in its package.

You can distribute Java applications in a number of different ways, but the most popular involve distributing class files in a compressed archive. The JAR (Java Archive) file contains a set of class files and some packaging information. In addition to Java class files, JAR files can include graphics and audio files and other assets used by the application. JAR files can also be digitally signed for security purposes. A JAR file can also be made executable, enabling the user to launch the application by running it.

4.2 How Java Compares to Perl

Now that you have gotten a look at Java, I outline how it might compare to Perl if you were trying to choose the right language in which to write a new application. The great thing about this choice is that most of the time you can’t go wrong. There are many projects that could be built equally well using either Java or Perl. You can use both languages to write reliable and maintainable code. The performance overhead of the JVM and the performance overhead of the Perl interpreter are roughly equivalent. So when is it advantageous to use Java, when should you stick with Perl, and when might it make sense to combine the two?

4.2.1 Java’s Strong Points

Implementing an object-oriented solution

By design, Java is a purely object-oriented language. While Perl and C++ make it easy to blend object-oriented and procedural programming styles, Java supports object-oriented design at the expense of procedural solutions. This is frustrating when part of a program is really just a simple sequence of steps and it feels excessive to design one or more classes around it. But when faced with a set of requirements for which an object-oriented solution seems like a natural fit, Java is a great choice. Java provides more design options (interfaces, abstract classes, etc.) and more levels of access control (the package system) than Perl, making it easier to implement the right object-oriented design. Java’s strong typing of objects also makes it easier to enforce rules about what can or cannot be done with data in the system.

Building easy-to-deploy client applications, especially on the Web

Java is a great language for small client applications. This includes applets, which can run within a Web browser, as well as distributed computing clients and other small applications intended for widespread distribution. Because JVMs are available for many platforms and are likely to be installed on many computers already, end users can download and run your Java application easily. The security features of the JVM and the ability to digitally sign JAR files mean that users can trust your Java application not to damage their system by deleting files or spreading viruses.

Building cross-platform GUI applications

Java is also a great language for cross-platform desktop applications. Java’s core desktop GUI components, the Swing classes, provide all of the native code required to manage windows and other elements on most popular desktop systems. You can configure a Java desktop application to run with the user’s native system look-and-feel or Swing’s own look-and-feel. While users are divided about the merits of the Swing look-and-feel, Java has indisputably made very significant progress on the GUI front since its introduction. Still, Java is only recommended for GUIs where portability is more important than performance. Although Java has been used for games with some success (some of Yahoo!'s most popular online games are written in Java), most Java desktop application are slower than their native counterparts.

Doing intensive work with RDBMS systems

Java and Perl both provide great basic services for working with relational database systems. Java's JDBC and Perl's DBI are good choices for an application that needs to store or retrieve some data using SQL. However, Java offers some additional tools for applications that are heavily data-driven.

Within the database itself, companies are beginning to use Java as a replacement for proprietary stored procedure languages. Oracle's 9i database and application server put a JVM into the database itself and introduced tools that make it easy to store Java objects in a database.

Outside of the database, Java also offers two very useful frameworks for database work. Entity Beans, part of the J2EE Enterprise Java Beans standard, make it easy to build objects that represent entities within the database. Programmers can work with these objects instead of embedding SQL directly into their code when they need to store or retrieve data. This isolates the impact of schema changes to only a few places within the application. Java Data Objects (JDO) is another specification developed by the Java Community Process. JDO lets you write database-independent classes to represent your data and then use deployment descriptors to control the persistence of the data to a RDBMS or XML datastore. Unlike Oracle's tools, you can use both Entity Beans and JDO with database systems from many vendors.

Building applications for wireless devices

The Java 2 Micro Edition (J2ME) is a great platform for building wireless applications. Many manufacturers of PDAs, mobile phones, and other wireless devices have begun including J2ME support over the past year. While J2ME devices use scaled-down versions of the JVM and the core Java libraries, they can be built with a standard Java compiler. Sun provides free tools and emulators for mobile development and a developer who knows standard Java can easily pick up J2ME basics. Because of the storage and bandwidth limitations of mobile devices, it is much easier to take advantage of existing J2ME support than to build mobile applications in Perl.

4.2.2 Perl's Strong Points

Implementing a procedural or hybrid solution

As mentioned above, Java is great for pure object-oriented design. it is also possible to use Java for procedural programming, but such programs often seem slightly awkward because all Java code has to be part of a class. If you only need to use classes in part of a program, Perl will give you much more flexibility.

Working with the Unix shell

Because Sun designed Java to be a portable, cross-platform language it does not provide great tools for accessing the Unix shell. You can use Java's Runtime and Process objects to execute non-Java programs from within a Java application, much like Perl's "system" and "exec" functions, but with considerably more hassle.

Regular Expressions

Java did not have a core regular expression library until the current version, JDK 1.4, was released last year. The java.util.regex package makes it possible to do a lot, but not all, of what you can do in Perl. The API documentation for the java.util.regex.Pattern class contains a good summary of the differences between the two. But while Java is now at least competitive in the regular expression field, the syntax for pattern matching is still unwieldy. The Java example in code listing 14 lacks the simplicity of its Perl equivalent in code listing 15:

Code Listing 14: Java regular expression

```java
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
if (m.matches()){
        System.out.println("matched!");
        }
```

Code Listing 15: Perl regular expression

```perl
if ("aaaaab" =~ /a*b/){ print "matched!\n"; }
```

While I generally do not mind the relative verbosity of Java, it is arguably a liability in code that parses text. Java's regular expressions are tremendously useful on occasion, but they are still a step down if you are accustomed to working with Perl's regular expression features.

4.3 Interoperability

Right now, building Perl and Java programs that can interact is a difficult proposition. Java provides the Java Native Interface and development tools to make it easy to access C code from a Java application, but there is no similar facility for Perl. On the reverse side, the current version of the Perl interpreter does not have facilities for accessing Java objects.

Fortunately, future developments in this area look good. Parrot, the next version of the Perl interpreter, represents a move to a virtual machine architecture that is conceptually similar to the JVM. With the emergence of Perl 6, Perl and Java programmers should be able to write more interoperable code, including both Perl programs that use Java objects and Java programs that use Perl objects. Python and Java developers already enjoy the ability to do this using Jython, a Java-compatible version of the Python interpreter.

5 Conclusion

Perl and Java are both great languages and there is no need to argue over which one is superior. Each has some great strengths as well as some areas in which it is not the best choice. By knowing what Java offers, you can make good decisions about when to take advantage of its benefits. See the references listed below for more information on Java.

6 References

"The Java Tutorial" (http://java.sun.com/docs/books/tutorial/): Sun's online tutorial is a good overview of the Java language. It is also available as a book.

Thinking in Java by Bruce Eckel is slightly out-of-date but provides a great introduction to object-oriented concepts in Java.

http://java.sun.com is Sun's official Java web site

http://www.jcp.org is the web site of the Java Community Process

http://www.onjava.com is a great Java resource from O'Reilly

http://www.jython.org is the Jython web site

7 About the author

Beth Linker is a Java developer at a Boston-based startup company and a volunteer copy editor for *The Perl Review*.

Filehandle Ties

Robby Walker, webmaster@cd-lab.com

Abstract

Perl can tie scalars, arrays, hashes, and filehandles to a user-defined class, to seamlessly extend its abilities without changing its syntax. Code that operates on a normal variable works equally well on a tied variable. I examine output filehandle ties by writing 3 filehandle tie modules.

1 Introduction

Tie-ing a filehandle is the same as tie-ing any other variable type.[1] In code listing 16 I tie the HANDLE filehandle to the fictitious module Module::Name. After the tie, when I use HANDLE like a filehandle, Perl will use Module::Name to determine what to do. The tie() command is a constructor since it returns an object[2]. I do not need to use or save this object since I can simply use the filehandle, but sometimes the object is useful, as I show later.

Code Listing 16: Tie-ing a filehandle

```
$object = tie *HANDLE, 'Module::Name', args,...
```

2 Writing Filehandle Tie Modules

Filehandle tie modules must at least implement the special TIEHANDLE and at least one of the input or output methods. A tied filehandle can respond to any method that a normal filehandle can, although I have to implement the right method to do that. Table 2 shows a summary of filehandle operations and the corresponding methods in the tied module.

2.0.1 Streamlining Tie Modules

I wrote Tie::FileHandle::Base as a base class to provide default implementations of many of the above methods. For example, in code listing 17 I implemented PRINTF to rely on PRINT. If I do not want that, I simply implement another PRINTF in my derived class.

Code Listing 17: Tie::FileHandle::Base PRINTF implementation

```
sub PRINTF { my $self = shift; $self->PRINT(sprintf @_) }
```

[1] See the perltie man page for details on the basic of tie()

[2] I can also get this object from the tied variable with the tied() command: `$object = tied *HANDLE`

Table 1: Tie filehandle equivalencies

Tied Handle	Equivalent To
tie HANDLE, 'Module', @args	$object = Module->TIEHANDLE(@args)
open HANDLE @args	$object->OPEN(@args)
binmode HANDLE $mode	$object->BINMODE($mode)
print HANDLE @args	$object->PRINT(@args)
printf HANDLE @args	$object->PRINTF(@args)
syswrite HANDLE @args	$object->WRITE(@args)
getc HANDLE	$object->GETC
readline HANDLE	$object->READLINE
<HANDLE>	$object->READLINE
read HANDLE @args -OR- sysread HANDLE @args	$object->READ(@args)
seek HANDLE @args	$object->SEEK(@args)
tell HANDLE	$object->TELL
fileno HANDLE	$object->FILENO
eof HANDLE	$object->EOF
close HANDLE	$object->CLOSE

3 Output Multiplexing

In many cases I want to send output of a program not only to standard output but also to a file. However, if I have already written my code it would be a painful and tedious task to replace every print statement with two (one for STDOUT and one for the file), and my code also becomes more difficult to maintain. I could concatenate the data into a string then print it twice at the very end of my program, but this is also tedious (a problem I address later) since i must be careful to lead all paths to those final two print statements.

Although either of the above options is manageable they lack the elegance and simplicity that makes Perl fun. A tied filehandle can take care of this behind-the-scenes so my code does not have to change. In code listing 18 I create a small module, Tie::FileHandle::MultiPlex, whose PRINT method can simultaneously send output to many filehandles. It uses Tie::FileHandle::Base which provides default implementations of the other filehandle methods.

Code Listing 18: Tie::FileHandle::MultiPlex module

```
package Tie::FileHandle::MultiPlex;
use base qw(Tie::FileHandle::Base);

sub TIEHANDLE { my $class = shift; bless [ @_ ], $class }

sub PRINT {
        my $self = shift;
        print $_ $_[0] for @$self;
}

1;
```

The program in code listing 19 combines the STDOUT, OTHER, and HANDLES filehandles in the OUT filehandle through the tie. The TIEHANDLE method in Tie::FileHandle::MultiPlex gets the typeglob references (*STDOUT) as arguments and creates the behind-the-scenes object which Perl ties to the OUT filehandle. When I print to the OUT handle, Perl knows that it is a tied filehandle so it looks in the Tie::FileHandle::MultiPlex module for the PRINT method. The PRINT method cycles through the filehandles in the behind-the-scenes object and sends the output to each one of them.

Code Listing 19: Using Tie::FileHandle::MultiPlex

```
use Tie::FileHandle::MultiPlex;

tie *OUT, 'Tie::FileHandle::MultiPlex', \*STDOUT, \*OTHER, \*HANDLES;

print OUT "stuff to be multiplexed";
```

4 Output Buffering

If I want to capture a program's output and only print it if some condition is met, I can run the code in an eval block and only create output if the eval did not produce any errors. PHP offers this functionality and my only major gripe with Perl is that does not. Of course the beauty of Perl is that not only can I do just about anything, but TMTOWTDI[3].

In code listing 20, I create a tied filehandle module, Tie::FileHandle::Buffer, that saves the output in a string rather than outputting it. I can later retrieve the buffer when I decide I really want to output it. The behind-the-scenes object is a blessed scalar.

Code Listing 20: Tie::FileHandle::Buffer module

```
package Tie::FileHandle::Buffer;
use base qw(Tie::FileHandle::Base);

sub TIEHANDLE {
    my( $self, $class ) = ( '', shift );
   bless \$self, $class;
}

sub PRINT { ${$_[0]} .= $_[1] }

sub get_contents { ${$_[0]} }

sub clear { ${$_[0]} = '' }

1;
```

In code listing 21 I use this module to print to HANDLE, which is really a buffer. When I want to output the data, I call the get_contents method on the tied object. In this example I had a use for the tie object so I saved it, although I did not do that in the earlier example.

[3]There is more than one way to do it

Code Listing 21: Using Tie::FileHandle::Buffer

```
#!/usr/bin/perl
my $object = tie *HANDLE, 'Tie::FileHandle::Buffer';

print HANDLE "Hello handle!\n";

print "Object says: ", $object->get_contents;
```

I use the Tie::FileHandle::Buffer module in the Output::Buffer module in code listing 22. An Output::Buffer object wraps the Tie::FileHandle::Buffer so I can trap program output until I call either flush() or clean(), or until the object goes out of scope, which I specify with the behavior parameter I pass to the new() method. In code listing 23 I chose the FLUSH behavior, and in code listing 24, I chose the CLEAM behavior.

Code Listing 22: Output::Buffer module

```
package Output::Buffer;

# BEHAVIORAL CONSTANTS
use constant WARN  => 2;
use constant FLUSH => 1;
use constant CLEAN => 0;

# EXPORT
our @ISA         = qw(Exporter);
our @EXPORT_OK   = qw(WARN FLUSH CLEAN);
our %EXPORT_TAGS = ( constants => [@EXPORT_OK] );

# DEPENDENCIES
use Tie::FileHandle::Buffer;
use Symbol;
use Carp;

# Create a new output buffer
# Usage: my $buffer = Output::Buffer->new( behavior )
# where behavior is either FLUSH, CLEAN, or WARN
#
#    FLUSH - when the object loses scope, print its buffer
#    CLEAN - when the object loses scope, discard its buffer
#    WARN - when the object loses scope, discard its buffer
#           but issue a warning
sub new {
   my( $class, $behavior )    = @_;
   my $fh     = gensym; # create an anonymous filehandle
   my $object = tie *{$fh}, 'Tie::FileHandle::Buffer';
   # store our behavior, our handle, and the handle we replaced
   bless [ $behavior, $object, select $fh ], $class;
}

# clean the output buffer, discarding its contents
sub clean { $_[0]->[1]->clear; }

# get our contents
sub get_contents { $_[0]->[1]->get_contents; }

```

```
# flush the output buffer, printing its contents
sub flush {
   my $self = shift;

   my $handle = $self->[2];

   # print our contents to our parent
   print $handle $self->get_contents;

   # then discard them
   $self->clean;
}

# Our scope has ended - deal with it by acting out our behavior
sub DESTROY {
   my $self = shift;

   if ( $self->[0] == FLUSH ) {
                  # FLUSH means flush!
                  $self->flush;
   } else {
                  # only WARN carps - and only if there was buffered output
                  carp "Discarded output buffer contents"
                                 if ( ($self->[0] == WARN) && (length($self->get_contents) != 0));
                  # both CLEAN and WARN imply cleaning
                  $self->clean;
   }

   # return the old filehandle to domination
   my $handle = $self->[2];
   select $handle;
}

1;
```

Code listings 23 and 24 show two ways that I can use Output::Buffer. In listing 23 I flush the output at the end of the eval, while in listing 24 I clear $buffer if eval set $@ (indicating an error). Otherwise, when I undefine $buffer in line 10, the DESTROY method outputs the contents of the buffer.

Code Listing 23: Using Output::Buffer, with flush

```
use Output::Buffer qw(:constants);

eval {
   my $buffer = Output::Buffer->new( CLEAN );

   # LOTS OF EXCEPTION-PRONE CODE HERE

   # if we got here, everything worked so flush
   $buffer->flush();
};
```

Code Listing 24: Using Output::Buffer, with clean

```
use Output::Buffer qw(:constants);

my $buffer = Output::Buffer->new( FLUSH );

eval {
   # CODE
};

$buffer->clean if $@;
undef $buffer;
```

5 References

The perltie man page

Programming Perl, 3rd Edition, Larry Wall, et al., O'Reilly & Associates.

Many modules on CPAN implement tied filehandles, including the ones that I used in this article,

Tie::FileHandle::Base
Tie::FileHandle::Buffer

and modules from other authors, including:

Tie::Handle
Tie::Syslog
Tie::STDERR
Tie::PerFH

The Iterator Design Pattern

brian d foy, comdog@panix.com

Abstract

The Iterator pattern separates the details of traversing a collection so that they can vary independently. Perl provides some of these parts already, although in some cases I need to provide my own implementations of them.

1 Introduction

Iterators as a design pattern are much more apparent as a pattern in languages that do not have aggregates as first class objects. Perl has lists, along with the variable types arrays and hashes, and it is trivial to go through any of these with built-in Perl structures like foreach, each, map, and so on. The file input operator, <>, iterates through the lines of the file. The readdir() and glob() functions iterate through filenames. In scalar context, the match operator with the global flag, m//g, iterates through matches.

In Perl I do not have to write classes to traverse an array, but I may need to write code to go through my own complex data structures because Perl's built-in control structures, like foreach(), map(), and grep(), do not define an interface that objects can use to control the iteration. They take a list, and I have to know what all of the elements of the list at the same time. A data structure might not even exist—the elements might be represented by an algorithm that determines the next element so that the object does not have to store all of the elements.

Iterators involve three parts: the *data*, the *iterator* that knows how to get the next item, and the *controller* which invokes the iterator. These may not always show up as distinct parts.

The iterator has to know at least two things: how to get the next element and when no more elements are available. Depending on the type of traversal, it may also need to know something about its state or the order it should follow. The controller uses this logic through some sort of interface, which may or may not be apparent, to interact with the data.

Since I separate each of these things when I use this pattern, I can easily change any one of them without affecting the others since they are *loosely coupled.* I reduce their dependence on other to give ourselves more flexibility and to improve the reusability of my code. For instance, I can change the iterator's traversal from depth-first to breadth-first, and the controller says the same since it uses the same interface. I can use the same controller for other iterators with the same interface, or the same iterator with other controllers, which gives me more choices if I decide I need to change something.

2 Types of iterators

Iterators come in two types: those where the something else—a distinct controller—controls the iteration, called *external iterators*, and those where the iterator controls itself, called *internal iterators.* Which one I decide to use depends on what I need to do. As the implementor of an iterator I have to do about the same amount of work in either case, although other programmers benefit, or suffer, from which one I decide to use.

2.1 Internal iterators

With internal iterators, I tell a method or function to perform some operation on each element of a collection because I already know that I will have to visit most or all of the elements, and as long as I do that I do not care how it happens. Internal iterators often combine the controller and iterator aspects into a single thing which simplifies the life of the programmer who uses the iterator in his code.

The map() and grep() built-in functions use internal iterators. They go through all of the elements in the data independent of our control because they combine the iterator and controller. I give these functions a bit of code that they apply to each of the elements in a list. The map() function returns a list of values based on the original list, and the grep() function returns a list of values from the original list that satisfied a condition. The controller and iterator parts are part of the perl core. I do not have to tell these functions how to get the next element in the data, when they have gone through all of the elements, or that they should move on to the next element.

Code Listing 25: Perl's built-in map and grep

```
my @squares = map { $_ * $_ } 0 .. 10;

my @odds = grep { $_ % 2 } 0 .. 100;
```

When I use the File::Find module, I let its find() function iterate through the directories on its own, and it applies the callback function to each file as it finds it.

Code Listing 26: File::Find internal iterator

```
#!/usr/bin/perl
use File::Find ();

File::Find::find( { wanted => \&wanted }, '.' );

sub wanted
	{
	print "$File::Find::name\n" if /^.*\.tex\z/s;
	}
```

File::Find knows how to get to the next element because it keeps track of its place in the filesystem and makes repeated calls for directory contents. When it asks for more files and the filesystem tells it that no more exist, File::Find knows it is done. As the programmer, I do not know any of this though. Once I start the find() function, it moves from element to element on its own.

Modules which implement composite data structures can define methods to iterate over all of their elements to perform an operation, such as a callback function or a Visitor object. The Netscape::Bookmarks module represents the information in a Netscape (and now Mozilla, too) bookmarks file, usually stored in HTML on a local computer, in a data structure so I can do interesting things with the data such as spell-checking, re-arranging, importing new links, converting formats, or link checking. On the insides it is a collection of different objects from the Netscape::Bookmarks classes Category, Link, Separator, and Alias.

The Netscape::Bookmarks::Category module provides a recurse() function that applies a call_back routine to every element in the current category and below, as well as an introduce() routine that passes its elements one-by-one to a Visitor object. The module handles the details of the iteration and the controller for us. Once I start the iterator it goes to completion on its own. Code listing 27 shows how little work the application programmer needs to do to traverse the entire Netscape::Bookmarks structure—indeed, that is the point. Not only does the programmer need to write only a couple lines of code, but if I change the mechanics or the implementation, the programmer does not have to change his code.

Code Listing 27: Netscape::Bookmarks internal iterators

```
use Netscape::Bookmarks;

my $bookmarks = Netscape::Bookmarks->new( $file );

$bookmarks->recurse( \&call_back );

$bookmarks->introduce( $visitor );
```

The recurse() and introduce() methods define a traversal order, and that order is not important to me as long as I get a chance to process each element.

Classes define internal iterators to handle tasks that to operate on every element of the collection in a uniform matter. The iterator treats all of the elements the same way and when the iterator is done with one element it automatically moves on to the next one. I create methods like recurse() when I do not want to do a lot of work at the application level. My scripts which use the module obey the interface, and if I find a better way to do it, the applications do not need to change[1].

2.2 External iterators

Most programmers use external iterators all the time without even knowing it. When I provide the controller and decide when to move on to the next element, I use an external iterator. Since the list is a fundamental Perl concept, Perl naturally has a lot of features to work with lists, and a lot of external iterator idioms.

To go through the elements of a list, I can use the foreach() control structure as an external iterator. I control the iterator because I have the ability to skip items (with next), stop the iteration (with last), or reprocess the current element (with redo). The foreach() controller does not move onto the next element until I let it, which I might do implicitly by not telling it to do something else.

Code Listing 28: External iterations with foreach()

```
foreach my $url ( @urls )
    {
    next if $link->scheme ne 'http';
    redo if $link->domain eq 'www.perl.org';
    last if $link->query =~ m/foo/;
    }
```

[1]You may see a bit of the Facade design pattern here.

Some collections do not exist as lists and so I must explicitly access one element at a time. The while() control structure controls the iteration by repeatedly fetching the next element to process. In this case, I have to know how to get the next element. The DBI interface returns rows from a record set one at a time through the fetchrow_array() method. The DBD driver knows how to fetch the next element, and fetchrow_array() returns false when I have fetched all of the records, signalling the end of the iteration. I can decide to stop the iteration at any point, even if I have not fetched all of the records.

Code Listing 29: The DBI external iterator

```
use DBI;

my $dbh = DBI->new(...);
my $sth = $dbh->prepare(...);

while( my @row = $sth->fetchrow_array ) {
        ... }
```

2.3 One controller, multiple iterators

Since I control external iterators, I can use more than one iterator at the same time. If I want to compare two files, for instance, I can use the line input operator on two different filehandles at the same time. Each time I go through the while loop I get the next item from each of the iterators. In code listing 30 the while controller uses two iterators.

Code Listing 30: One controller, multiple iterators

```
while( my ( $old, $new ) = (scalar <OLD>, scalar <NEW>) ) {
        ... }
```

2.4 One iterator, multiple controllers

I do not have to use the same control for all parts of the iteration. In code listing 31 I read the first line of standard input using the line input operator (the iterator) with an assignment (the controller, if you will) to a scalar variable. Perhaps this line represents the column headings in a flat-file database. After that, I read in the next ten lines with a different controller, the while() loop, after which I go through the remaining lines with grep(). I can only do this because I can decide when to move on to the next element.

Code Listing 31: Multiple controllers

```
my $titles = <STDIN>;

my $count = 0;
while( <STDIN> ) {
        last if $count++ >= 10;
        ... }

my @lines = grep { /Perl/ } <STDIN>;
```

2.5 Working with data not in memory

I also use external iterators when all of the of the data cannot or should not be in memory at the same time. If I work with a tied DBM hash, my hash represents possibly large numbers of keys and values stored on disk. Since the elements of the hash are not in memory, I save space. If I use the keys() or values() functions those potentially large numbers of keys or values are now stored in memory, negating my savings. The each() iterator fetches one key-value pair at a time.

Code Listing 32: Iterating through a DBM file

```
while( my($key, $value) = each %DBM )
    {
    $sum += $value;
    }
```

This is the same idiom as reading a file line-by-line rather than all at once. Since the filehandle potentially delivers more memory than our program can handle or more RAM that our hardware has, most people recommend you read the file one line at a time, like I do in code listing 33. The earlier DBI example in code listing 29 does the same thing.

Code Listing 33: Iterating through a DBM file

```
while( <FILE> )
        {
        ...
        }
```

3 Iterator interfaces

Once I decide that I need to create my own iterator, I have to design an interface for it. The design pattern only shows me the general solution, so I have to look at the specific problem to see how I can apply the general pattern.

3.1 Object methods

Some modules save memory by computing elements only when needed, or fetch data on request from remote sources, like my earlier DBI example in code listings 29 or 32. In these cases the object has a method to return the next element.

The Set::CrossProduct module lets me deal with all of the combinations of elements from two or more sets. For instance, for the two sets (a,b) and (1,2) I get the combinations (a, 1), (a, 2), (b, 1), and (b, 2). The number of combinations is, at most, the product of the number of elements in each set, which means that the number of elements can be very large for even a small number of moderately sized set. Five sets of five items has over 3000 combinations. In code listing 34 I get back all of the combinations at once and store them in @combinations, meaning that I potentially use up a lot of memory even though I later go through the combinations sequentially in the foreach() loop. This has the same problems as reading an entire file into an array.

Code Listing 34: All combinations at once

```
use Set::CrossProduct;

my $cross_product = Set::CrossProduct->new( [ [qw(a b)], [ 1, 2 ] ] );

# get all combinations at once
my @combinations  = $cross_product->combinations;

foreach my $item ( @combinations )
        {
    print "The combination is @$item\n";
    }
```

Set::CrossProduct does not store the combinations in memory though. It simply stores the sets and keeps track of which combination it needs to make next. The combinations() method in code listing 35 has to create the list for me. Set::CrossProduct provides a next() method, the iterator, which lets a controller fetch the next value. It is an external iterator, so I need to provide the controller to make the iterator move from one element to the next so I only have to store one combination at a time. In code listing 35 I use a while() loop to repeatedly fetch the next combination—each time testing the return value of the next() method to see if it returned a combination.

Code Listing 35: Iterate through successive combinations

```
use Set::CrossProduct;

my $cross_product = Set::CrossProduct->new( [ [qw(a b)], [ 1, 2 ] ] );

while( my $item = $cross_product->next )
    {
    print "The combination is @$item\n";
    }
```

3.1.1 When has the iterator finished?

How do I know when no more elements are available? The interface has to signal to the controller that the iterator has gone through all of the elements and that the controller should not ask for any more.

I can return a false value but does not always work. The line input operator, for instance, uses undef to signal the end of input. That means it does not use all the false values for this signal since 0 and the empty string are not undefined. Any value besides undef comes from the data source and is an element of the iteration. In code listing 36 I test specifically for the undef value to see if the input is finished.

Code Listing 36: Line input operator in while

```
while( defined( $line = <STDIN> ) ) {
        ... }
```

Perl has a special idiom for this if I use the default variable $_. I could write it out to look the same as code listing 36 but with $_, or I can write in much more simply as in code listing 37 which does the same thing.

The while() condition tests to see if the item is defined, not that it is true. This is a special case only for when I use the line input operator with $_ in the while() condition.

Code Listing 37: Test for a defined value, not a false one

```
while( <STDIN> ) { # really while( defined( $_ = <STDIN> ) )
        ... }
```

What if undef value is a valid value? I cannot use it to signal the end of the iteration. I might be able to use another value that does not cannot appear in the data, but if any value is valid, I cannot use inspection to decide what to do[2].

I can design an iterator which has another method which tells me the state of the iteration. I check this method before I attempt to fetch the next element to see if any more elements are available, and if none are, the controller knows to stop. In code listing 38, if the has_more_elements() method returns false I stop the iterator.

Code Listing 38: Check if more items are available

```
while( $iterator->has_more_elements )
        {
        $item = $iterator->next;
        ...;
        }
```

More Perl-like methods work too. I can always return a reference to the data instead of the data itself. Even a reference to a false value is true since I test to determine if the variable is a reference instead of checking its value. The iterator returns a non-reference to signal that no more elements are available. The interface is the almost the same as code listing 38 since a reference is always true, even if the data it points to would evaluate to false.

If the next() method always returns a reference, perhaps an array reference, I can tell the difference between undef and the anonymous array of one element that contains the undef value. The values `[ undef ]`, a reference, and undef, are different. Code listing 39 loops until the next() method returns any false value since references are always true.

Code Listing 39: Return a reference

```
while( my $ref = $iterator->next ) # [ undef ] works, undef doesn't
        {
        my @items = @$ref;
        ...
        }
```

3.2 Custom controllers

The Object::Iterate module defines some controllers for these sorts of interfaces so I can interact with the object just like I do with lists for foreach(), map(), and grep(). It defines iterate, igrep, and imap which

[2]Mark Jason Dominus calls this the Semi-predicate Problem

look almost just like the perl built-in functions, but takes an object that can respond to a couple of special method names. Code listing 40 shows the iterate(), imap()[3], and igrep() functions.

Code Listing 40: Object::Iterate controllers

```
use Object::Iterate qw(iterate igrep imap);

iterate { print "$_\n" } $some_object;

my @output = imap { ... } $some_object;

my @filtered = igrep { ... } $some_object;
```

Each of these functions goes through all of the elements of the object through the object's interface. Without hints from the object, the Object::Iterate module uses the special object methods __next__ and __more__ to get the next element and determine if more elements exist. The object's class has to implement these methods itself, and the three controllers work with any object that follows the interface. The functions act as internal iterators just like the example in Section 2.1. Once I start them they go through the entire structure without further control from me.

Code listing 41 shows the implementation for the iterate() function. It takes an anonymous subroutine as its first argument which lets it mimic the syntax for map {}[4]. The object over which it will iterate is the second argument. In lines 5-8, iterate() ensures that the object has the right special methods. In the while() loop, iterate() does the same thing as in code listing 38.

Code Listing 41: Object::Iterate::iterate

```
1  sub iterate (&$)
2          {
3          my( $sub, $object ) = @_;
4  
5          croak( "iterate object has no $Next() method" )
6                  unless UNIVERSAL::can( $_[0], '__next__' );
7          croak( "iterate object has no $More() method" )
8                  unless UNIVERSAL::can( $_[0], '__more__' );
9  
10         while( $object->__more__ ) {
11                 local $_;
12 
13                 $_ = $object->__next__;
14 
15                 $sub->();
16                 }
17         }
```

[3]Mark Jason Dominus came up with similar methods for his "Iterators and Generators" talk, but implemented them differently. For awhile I felt bad about choosing "imap" for a name since its also a well-known protocol, but after I saw that he choose the same name, I did not feel so bad.

[4]See the Prototypes section in perlsub

3.3 Closures

I do not necessarily need modules and classes to create iterators either. I can use a closure to hold all of the information. If I want to iterate over odd numbers, an infinite series which I cannot ever completely store in memory, I can create a closure that returns the next odd number each time I call it. It maintains its own state and I avoid all of the overhead of method lookups.

Code Listing 42: Closures as iterators

```
my $odds = do { my $next = -1;  sub { $next += 2; return $next } };

while( my $number = $odds->() )
    {
    print "The next number is $number\n";
    }
```

Some people call closures “inside-out object”. Objects are data with behavior while closures are behavior with data, so they make handy iterators. I combine the data and iterator portion to create the closure. Each time I dereference the closure, I get back the next value. The closure comprises the iterator and the data portion, while I supply the controller.

3.4 Tied scalars

Tied scalars must have a FETCH method, but nothing specifies what I have to do or which data I have to return with that method. The Tie::Cycle module ties an anonymous array to a scalar so that each time I access the scalar’s value, I get the next item from the array, and when I get to the end of the array it goes back to the beginning. The controller is the use of the scalar on the right-hand side of an expression, the FETCH method defines the iterator, and the anonymous array stores the data. In this case, the tied scalar combines the iterator and the data, although I still provide the controller because I use program logic to decide when to access $colors even though I do not use an explicit controller.

I initially created Tie::Cycle to handle alternating colors in rows of HTML tables. I grew weary of creating bugs when I changed the colors or their number, and the amount of distracting code that had to go into calculating an index that stayed within the bounds of the defined elements of an array. All I wanted was the next color, and I wanted that to be simple. Tie::Cycle handles the annoying parts for me, and I can reuse it wherever I need it. Code listing 43 shows a typical use to shade rows of HTML tables with varying levels of gray. Each time I access the tied variable $colors, on line 7, the Tie::Cycle module advances along the anonymous array I gave it as an argument on line 3.

Code Listing 43: Tie::Cycle

```
1  use Tie::Cycle;
2
3  tie my $colors, ’Tie::Cycle’, [qw(aaaaaa cccccc ffffff)];
4
5  foreach my $row ( @rows )
6      {
7      my $row_color = $colors;
8
9      print <<"HTML";
<tr>
    <td bgcolor="$row_color">
    ...
    <td>
</tr>
HTML
    }
```

3.5 Tied filehandles

I can attach almost any data to a filehandle with a tie. I have to implement some of the functionality myself, such as determining what the next "line" is, but once I have done that little bit of work I can use all of Perl's filehandle iteration framework. This can be especially beneficial to new programmers who can work with filehandles but have yet not used Perl's more advanced features.

In code listing 44 I tie a normal scalar to an input filehandle so I can read the scalar line-by-line or one character at a time just as if I were reading from a real file. Code listing 45 shows its use in a program. The READLINE function defines how to read a line (or several lines in list context), and the GETC function defines how to read one character. Perl uses READLINE when I use the file input operator <> and GETC when I use the getc() function. In each case, the module removes the piece that I read, so our scalar gets shorter and shorter (unless I add to it by some other means, which I might want to do if I create an in-memory buffer).

I can decide how to read my lines. In this case, I avoid an annoying chomp by not returning the current value of the input record separator, $/ (a newline by default, but maybe something different) which in code listing 44 I assign to $EOL in line 14 and use in the regular expression, although outside the memory parentheses, in line 21.

Code Listing 44: Treat a scalar as a file

```
 1  package Scalar::Iterator;
 2
 3  sub TIEHANDLE
 4      {
 5      my( $class, $text ) = @_;
 6
 7      bless \$text, $class;
 8      }
 9
10  sub READLINE
11      {
12      my $self = shift;
13
14      my $EOL = $/;
15      if( length $$self > 0 )
16          {
17          my @lines = ();
18
19          while( length $$self > 0 )
20              {
21              $$self =~ s|(.*?)$EOL||s;
22              print "Matched $1\n";
```

```
            push @lines, $1;
            last unless wantarray;
            }

        return wantarray ? @lines : $lines[0];
        }
    else
        {
        return;  # undef signals the end
        }
    }

sub GETC
    {
    my $self = shift;

    return $1 if $$self =~ s|(.)||s;

    return;  # undef signals the end
    }
```

Code Listing 45: Using a scalar as a file

```
use Scalar::Iterator;

my $data = ...;

tie *MY_DATA, 'Scalar::Iterator', $data;

my $line = <MY_DATA>;
print "Got one line [$line]\n";

my $char = getc( MY_DATA );
print "Got next character [$char]\n";

print "Got rest of lines:\n", <MY_DATA>;

```

I can change the way that I go through the scalar. I can change what I mean by “line” and “char” to be something else. After all, the computer does not really know what these things are. In code listing 46, I change READLINE to read the next sentence, and GETC to read the next word. This is more complicated than it sounds if I wanted to do this to arbitrary text, so I have to do more work than I do for the general case.

Ever wanted to put a regular expression into $/? Well, now I can. In code listing 46 I conveniently used $EOL as the end-of-line marker in my example, and I put it at the end of my regular expression in READLINE. If I put regular expression special characters in there, the s/// operator will interpret them as regular expression sorts of things. In this case I think I have reached the end of the sentence when I run into the next punctuation character in the class `[.!?]`. This time I include the end-of-record marker, the sentence ending punctuation, with the sentence. At the same time I collapse consecutive whitespace to a single linear space.

I have to make a minor change to make GETC read words. For this example, I pretend that words only have alphabetic characters and ignore special cases like contractions, abbreviations, and hyphenated words. In that case, GETC only has to return the next sequence of letters while skipping over non-alphabetic characters it finds first.

My data has not changed. It still can be anything that I like, but I can easily change how I go through it, since all the bits of the iterator stay separate from the object (not really an instance, in this case). If I decide to change how I go through the object, the iterator is the only thing that changes. I can even define several different iterators and use them at the same time if I like. I do not have to do much more work to turn our sentence reader into a paragraph reader, and so on. Tied filehandles have infinite uses as iterators, but beware—tied filehandles can be slow. From ease-of-use, flexibility, and speed, I only get to choose two.

Code Listing 46: Sentences and words

```perl
sub READLINE
	{
	my $self = shift;

	my $EOL = '[.!?]';
	if( length $$self > 0 )
		{
		my @lines = ();

		while( length $$self > 0 )
			{
			$$self =~ s|(.*?$EOL)||s;
			my $sentence = $1;
			$sentence =~ s/\s+/ /g;
			push @lines, $sentence;
			last unless wantarray;
			}

		return wantarray ? @lines : $lines[0];
		}
	else
		{
		return;  # undef signals the end
		}
	}

sub GETC
	{
	my $self = shift;

	return $1 if $$self =~ s|[^a-z]([a-z])||i;

	return;  # undef signals the end
	}
```

4 Perl Modules which represent iterators

Besides the modules I used as examples, several other modules express the Iterator design pattern.

4.1 XML::*, HTML::*

Some of the XML and HTML modules represent stream-based parsers. I give them a big chunk of text, and the module breaks it into pieces and gives me the chance to interact with the pieces. The parsers act as the iterators and the controllers, while I supply behavior for the items they encounter. The parsers know how to get the next item.

4.2 File::ReadBackwards

The File::ReadBackwards is a simple iterator that gives me the lines from a file one at a time, only starting at the end and working its way to the beginning. I can use its object interface or its tied filehandle interface. I supply the external iterator, and the module knows how to get the next item.

4.3 Tie::DirHandle

The Perl built-in function readdir() is an iterator. In scalar context it gives me the next filename from the directory handle, and in list context give me back all of the filenames. It defines the logic of the traversal and I supply the controller. This modules hides the readdir() so I can use the filehandle iterators to interact with the directory handle. I get the next filename with the line input operator instead of readdir().

4.4 Tie::IxHash

Normally, hashes do not preserve the order of the elements I add to them. Perl stores the key-value pairs in a, well, hash tree, for easy lookup. Several modules, including Tie::IxHash, remember the order of hash operations, including addition of keys, so I can get the keys back in the order that I added them. It uses a tied hash to define the logic of traversal, and I can then use the standard Perl external iterators idioms to traverse the hash.

5 Conclusion

The Iterator design pattern has three parts: the data, the iterator, and the controller. In most cases Perl supplies the iterator and controller with its built-in functionality. In some cases, I have to write my own iterator to decide how to get the next item in the data and to decide when the iteration is complete. As with other design patterns, the implementation is just an expression of the pattern, and there is more than one way to do it. Which way I choose depends on my particular problem.

6 References

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

7 About the author

brian d foy is the publisher of *The Perl Review* and a Perl trainer for Stonehenge Consulting Services, http://perltraining.stonehenge.com.

Book Reviews

Writing Perl Modules for CPAN

Sam Tregar, Apress, 1-59059-018-X
August 2002; 312 pages
reviewed by Andy Lester

Sam Tregar's *Writing Perl Modules for CPAN* is an excellent introduction to the world of modules, and the community of Perl and open source software in general. He surveys all aspects of module installation, creation, documentation, testing, packaging, distribution, and maintenance in a clear, readable style.

Tregar starts outside of the context of CPAN, and examines the techniques of writing good, reusable code and, when appropriate, objects. The only misstep here is in introducing overloaded operators, a technique which is obfuscatory at best to the beginner. It should have appeared later in the book, if at all.

Design and construction are based on solid practices, emphasizing design, portability and documentation. A crash course in POD emphasizes the importance of writing documentation first. I'd like to have seen more about writing tests at the same time as code.

Module distribution and maintenance is key to the ongoing success of a module and its usefulness to the community, and Tregar handles these topics well. He explains the ins and outs of MakeMaker, discusses dealing with PAUSE, and using rt.cpan.org for tracking defects and requests in modules. It's great to see a book on Perl discuss topics about the community.

The "Great CPAN Modules" would have been a fine idea if it had more modules, and more explanation of their utility. The chapter is all text with very few examples. (Hint to any publisher who's reading: Do a book on great Perl modules and dissect them, *a la* the source code commentary books from Coriolis.)

Clearly, there's a lot of ground to cover in 312 pages, and Tregar does not dig into the depths like Jenness and Cozens' *Extending & Embedding Perl*. In particular, the sections on Inline and XS can't give more than a taste of what's possible.

As usual, the Apress layout is clean, readable and attractive. However, there are far too many distracting footnotes. Almost every module mentioned is noted "Written by $author, available on CPAN." Pages with more than five footnotes are common.

Still, for the programmer who wants to write solid, reusable code, and possibly contribute to the Perl community as well, this book is a must-have.

Programming The Network With Perl

Paul Barry; Wiley; 0-471-48670-1
March 2002; 394 pages
reviewed by Andy Lester

Paul Barry's *Programming The Network With Perl* is only the second book on the subject in Perl, and is a welcome alternative to Lincoln Stein's excellent *Network Programming With Perl*. Barry comes at network programming from the ground up, starting in with TCP/UDP and working through sockets and into application protocols like HTTP, SMTP and telnet. Stein has more topics and depth (in twice the pages), but Barry covers topics like SNMP, packet sniffing and mobile agents. If you already have Stein, and don't need any of Barry's unique topics, you can probably do without. Otherwise, Barry is a fine choice.

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