

The Perl Journal

ObjectivePerl: Objective-C-Style Syntax And Runtime for Perl

Kyle Dawkins • 3

Scoping: Letting Perl Do the Work for You

David Oswald • 7

Secure Your Code With Taint Checking

Andy Lester • 10

Detaching Attachments

brian d foy • 14

Unicode in Perl

Simon Cozens • 16

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

***XML Publishing with AxKit* • 20**

Source Code Appendix • 22

LETTER FROM THE EDITOR

A World of Text

Just like in life, being a good citizen in the programming world requires a little work. You have to educate yourself, and find a way to bury differences so that you can unite with others in a common cause. Granted, in the programming world, these causes tend not to be of life-or-death importance (unless you happen to be coding, say, a heart monitor), but even so, making a little effort toward good programmer citizenship can help to spread some good will around the world.

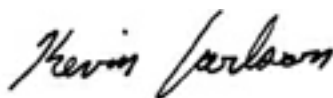
Take Unicode. In the late 1980s, nearly simultaneous discussions at Xerox and Apple about the complexities of multilingual text encoding resulted, eventually, in an industry-wide collaboration on a universal standard for representing the many script systems used around the world. That standard became known as Unicode. The Xerox folks were working on a way to extend the Chinese character set for their customers in Asia; and at Apple, development of Apple File Exchange inevitably led to discussions of the same problems the Xerox team were wrestling with. Collaboration was logical, and over time, nearly every major operating-system vendor joined the effort. In this case, what was good for international communication was also good for international business.

Thankfully, the Unicode standard didn't just extend multilingual cross-compatibility to those who had enough money to be a potential customer base for the computer industry. The open and universal nature of the standard has allowed it to be extended to encompass any human language. So it truly can be a standard for the whole world.

Perl now has pretty robust support for Unicode, so there's little excuse for writing apps that aren't Unicode-aware. (See Simon Cozens's article on page 16.) So why don't more of us do just that? I suspect that some of us still suffer from a bit of monolingual myopia. Despite years of Spanish classes, I myself am shamefully monolingual. But that doesn't explain it all.

Perhaps this is just one area where Perl's quick-and-dirty simplicity can be a drawback. We often begin building a tool in Perl as a simple, quick solution to local and specific problems, and only later realize that our solution has wider applicability. We don't always then return to the design phase and rearchitect our tool as a proper application to serve the needs of that wider audience. Yet, this type of redesign is a prerequisite for widespread code reuse, which both accelerates the growth of a language and extends its usefulness.

Happily, Perl makes much of the work of supporting Unicode trivial, or even occasionally completely unnecessary. Most of the string-processing functions in Perl now work just fine on Unicode text. So check out your current crop of Perl apps—you may find that you are already Unicode-compliant and didn't even know it.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

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

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Wyser

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, Andy Lester

CONTRIBUTING EDITORS

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

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Michael Goodman

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403
650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING

Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT/GROUP DIRECTOR INTERNET BUSINESS

Mike Azara

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES

Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK COMPUTING

ENTERPRISE ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT

MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS

Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

Perl News

Draft Version of Synopsis 9 Released

Larry Wall has posted a draft of Synopsis 9 to the perl6-language list. Although the Apocalypse it refers to hasn't been written yet, the synopsis nevertheless describes it in a confident past tense: apparently the "nonexistent" Apocalypse "discussed in detail the design of Perl 6 data structures. It was primarily a discussion of how the existing features of Perl 6 combine to make it easier for the PDL folks to write numeric Perl." You can read the full Synopsis at <http://groups.google.com/groups?threadm=20040902234740.GA29156@wall.org>.

OSI Honors Larry

The first ever "Grand Master," or gold-level, award to be bestowed under the Open Source Awards program has gone to Larry Wall. "Larry Wall is best known for the Perl programming language," the awards committee noted, "but this award also honors his contribution of the program *patch*, which stimulated Open Source Software development by enabling multiple people to contribute small changes to software in an efficient and elegant manner." The award is accompanied by a prize of \$10,000 and an invitation to serve as a future Elector of the awards committee.

The Open Source Awards were founded in 2003 as a joint project of the Open Source Initiative and CNET, with initial sponsorship from ActiveState and U.S. Venture Partners. Although awards have been announced every quarter, most of these have been at the "Merit," or bronze, level; Larry is the first to receive a Grand Master award. Grand Master honorees "must have made an outstanding contribution to the open-source culture." Either a "unique and exceptional technical innovation" or "a long-standing record of service" can qualify a candidate for a Grand Master award, but the committee looks for candidates who contribute both. Currently serving on the awards committee are Jeremy Allison, Larry Augustin, Jim Gettys, Dr. Marshall Kirk McKusick, Keith Packard, Eric S. Raymond, and Guido van Rossum. David Berland of ZDNet and Tim O'Reilly of O'Reilly and Associates serve as nonvoting members.

Nominations for future Open Source Awards can be sent to osanominations@opensource.org; see <http://opensource.org/OSA/nominations.php> for details.

Feeling Listless?

The perl6-compiler mailing list has at last been launched, prompting some discussion of the current state of work on the compiler.

According to Piers Cawley, who has taken on the job of summarizing the list weekly, "Patrick said that he and Luke (I think) are at the beginning stages of building a basic perl 6 grammar engine that compiles to parrot and handles some basic optimizations. In tandem with that, Patrick was also working on writing a Perl 6 grammar. The plan is (and no plan survives contact with the enemy) to get a working basic grammar engine up in the next month or so then use that to build the Perl 6 parser."

Meanwhile, back at stately perl5-porters, two-year veteran Rafael Garcia-Suarez is seeking someone to take over his summarizing duties. "No particular knowledge of the Perl 5 internals is needed," he notes, "You'll just have to be able to follow the main discussions and to look up the terms you don't get in the pods. Expect 2 or 3 hours of work by week, depending on the traffic. And you'll become a hero. Well, at least I'll owe you beers." Rafael can be contacted at rgarciasuarez@mandrakesoft.com.

Free to a Good Home

The hunt is on to find new maintainers for Simon Cozens's 100 CPAN modules, now that Simon has retired from module maintenance (but not from his regular contributions to *TPJ*, thankfully) in order to train as a missionary. Andy Lester has taken over the job of reassigning them, so e-mail him at modules@petdance.com if you'd like to volunteer; see <http://use.perl.org/~petdance/journal/20904> for details.

At least one module won't require maintenance: *Acme::OneHundredNotOut* (<http://search.cpan.org/dist/Acme-OneHundredNotOut/OneHundredNotOut.pm>) is Simon's autobiography of his Perl work, from games to UI to mail handling to Perl internals to personal robotics. Simon's continuing adventures in "God School" can be followed at <http://blog.simon-cozens.org/>.

Conference Calls

Venues have been chosen for YAPC::NA::2005 and YAPC::EU::2005: The European conference will be held in Braga, Portugal, while the North American conference is scheduled for the University of Toronto, June 22–24. Also on the calendar is the London Perl Workshop (December 11; call for proposals at <http://london.pm.org/lpw/>); the 7th German Perl Workshop (February 11; call for venues and a preliminary call for proposals at <http://www.perl-workshop.de/>); and YAPC::Taipei 2005 (March 26–27; themed around "Perl in the Enterprise"; proposals in either Chinese or English welcomed at hcchien@hcchien.org).

ObjectivePerl: Objective-C-Style Syntax And Runtime for Perl

I first encountered the Objective-C language on some of the earliest NeXT workstations in the early '90s and had been impressed by its elegance and power. It was very easy to learn and its approach to object-oriented programming was refreshing. Instead of having an all-powerful, strongly typed compiler (like C++, Ada, or Modula-3), it gave the programmer the choice of strong- or weak-typing at compile time. An incredibly rich runtime environment gave the programmer a great deal of flexibility and control over the objects in the system, unlike those other languages.

Then, in the mid-'90s, Java hit the software world with a crash. Java came solidly from the C++ world in terms of its design: A fussy compiler that forced strong-typing and an even fussier and underpowered runtime. Tunnel vision set in and much of the software world somehow convinced itself that Java was the “right” way to do OO and that it was “advanced” and “pure.” That’s one opinion—one with which Perl programmers often disagree.

After being stuck in the world of Java for years, I ended up in the world of Perl due to some great luck. In Perl, I immediately spotted the runtime-on-steroids that I missed from Objective-C. The extraordinary power and flexibility of the Perl compiler/runtime was a breath of fresh air, and I took to Perl like a duck to water.

Unfortunately, as time went by, some of the luster faded. I was disappointed with the Perl concept of objects; or rather, that Perl had a few different concepts of objects and they were all a bit “bent.” An object is a blessed hash? Or a blessed array! Or a tied variable! It was all a bit much. Even though I knew that it was simply a testament to the amazing power of Perl, it didn’t stop me from missing some of the things in Objective-C, and even Java, that were lacking in Perl.

Kyle works for the New York-based consultant firm Central Park Software and can be reached at kyle@centralparksoftware.com.

Things That Perl Didn't Have

- Real instance variables.
- Visibility levels of instance variables.
- Named arguments in method signatures.
- Static versus instance methods.

To me, these are nonnegotiable in an object-oriented language. Let's go through each one and talk about it in relation to Perl.

Real Instance Variables

Perl emulates instance variables in many different ways. For objects that are constructed as blessed hashes, which I'd say is the most common paradigm in Perl, the closest thing to instance variables are the keys to the blessed hash. So if I have an object *\$self*, I refer to its “window” instance variable like so:

```
$self->{window} = SomeWindow->new(10, 10, 100, 100);
```

This produces a lot of extraneous code in the form of *\$self->{blah}* and sometimes makes it hard to catch errors at compile time, even when you're using *strict*, since hash keys are not checked in any way. Object-oriented languages like Java implement instance variables in a very clean way. You declare it once in the class:

```
public String someInstanceVariable = "Hey you!";
```

and then later on, in some instance method, you just use that variable as a local:

```
public String someMethod() {  
    if (someInstanceVariable.equals("Hey you!")) {  
        return "Foo!";  
    }  
    return "Bar!";  
}
```


Visibility Levels for Instance Variables

Data abstraction and encapsulation is critical to the object-oriented model. In Perl, you really have to work hard to encapsulate your data. Damian Conway devotes 30 pages to it in his book *Object Oriented Perl* (Manning, ISBN 1884777791), providing numerous suggestions and techniques. Most Perl programmers trust themselves and others not to break the rules, and everyone's happy. But contrary to what a lot of people think, visibility levels are not for security purposes; they're to make it harder for you to shoot yourself in the foot.

In Perl, I immediately spotted the runtime-on-steroids that I missed from Objective-C

Many object-oriented languages provide at least three levels of visibility for instance variables:

- Public—any object can access this instance variable on its own object.
- Protected—only an instance of this object or an instance of a subclass of this object can access this instance variable.
- Private—only an instance of this object can access the instance variable.

Some languages also define other levels. For example, Java defines “package” to mean that the instance variable is visible to any object in the same package as the owning instance. Since Perl doesn't really have real instance variables (see above), it obviously doesn't have any kind of built-in visibility system, either.

Named Arguments in Method Signatures

This is such a simple and effective concept that it's mind-boggling that the designers of Java missed it, and it's made the life of many a Java developer a total nightmare. But let's take a look at the concept first and what it means to Perl.

Most of us would agree that a method usually has a name, some arguments, a body, and a return value of some kind. There are many variations, of course (methods with no return values, or no name, or no arguments), but this is the general idea. That's all very well, but more often than not we end up with code that looks like this:

```
$newObject->init("Phil", 185, 95, "FRANCE", "BROWN", "GREEN");
```

What is all that? You can't tell from context exactly what the programmer meant by this line of code. You can get the gist, that's for sure. It's initializing *\$newObject* with a slew of values, and we can probably surmise that “Phil” is a name, the next two numbers are probably height and weight (in metric values), then maybe country of birth, then probably hair color, and finally eye color. But maybe it's not “country of birth” but “country of residence”?

And maybe BROWN refers to trousers, not hair color? I think you get the point. Java suffers from this terribly, particularly because of its method overloading: A class could have 10 methods, all called *set*, that take different numbers of arguments. So when you see this in someone's code:

```
myNewObject.set(10, "Tweak", new Integer(12), "Stalefish");
```

you're going to be lost.

The solution to this is named arguments. We see it in Perl fairly regularly (and yes, you should pay particular attention to the names in this example):

```
$newObject->init( -subject => "Phil",  
                 -courseNumber => 185,  
                 -numberOfStudents => 95,  
                 -fieldTripLocation => "FRANCE",  
                 -professor => "BROWN",  
                 -teachingAssistant => "GREEN" );
```

That is much easier to understand than the earlier example, and we see that our original assumptions were in fact totally incorrect, too.

But you can take this one step further. If you were to look at the source code to the *init* method above, you would not immediately see what the arguments are:

```
sub init {  
    my $self = shift;  
    ...  
}
```

so if you generated API documentation from your code (which is something I do), you would not see a canonical list of what that method expects. Moreover, since it's really a hash, it has no inherent order.

In Objective-C, this problem is solved by naming the arguments within the method signature itself:

```
- (void) initWithSubject:(char *)subject  
               courseNumber:(char *)courseNumber  
           numberOfStudents:(int)numberOfStudents  
       fieldTripLocation:(char *)fieldTripLocation  
               taughtBy:(char *)professor  
               assistedBy:(char *)teachingAssistant  
{  
    ...  
}
```

So you can tell, both from the method signature and any invocation, what is going on. This also helps with class browsers, for example, where you can browse through the class hierarchy and see clearly which arguments go where for any method.

Static Versus Instance Methods

In Perl, there's no real distinction between static and instance methods. Sometimes it's nice, as a programmer, to be able to delineate methods that work on a class level from methods that work on an instance level. Most static methods in Perl are only static because their first argument is the class name, rather than an instance of the class. It's a matter of taste, but I prefer clearly indicating which methods are static and which are not.

Enter ObjectivePerl

I had many different projects on the go, some for work, some personal, but all Perl. I found myself longing for the day where I could write nice, clean, tight, readable code of the kind that

Objective-C tended to encourage. But I also found myself very attached to the wonderful Perl runtime and its supreme flexibility. So I decided to harness that power and try to solve some of the problems I outlined earlier. But how?

My first decision was a simple one. I like Perl, I like Objective-C, so why not add the familiar Objective-C syntax to Perl, the way it was originally grafted onto C? Thanks to Damian Conway's excellent *Filter::Simple*, this proved to be possible. Well, almost.

ObjectivePerl Syntax

To understand ObjectivePerl syntax, you need a quick primer in Objective-C syntax. The core of Objective-C syntax is the message-passing mechanism. Rather than invoking methods on objects, you send messages to objects. It's a small distinction but it's subtly different from your standard object/method paradigm. Objects might not respond to a message, but that won't necessarily cause your code to yack. Also, objects can forward messages to their delegate objects, for example. Message passing is achieved using this syntax:

```
returnValue = [object message];
```

or with arguments:

```
returnValue = [someObject doThis:this withThat:that
andSomethingElse:thisOtherThing];
```

You are free to embed messages in messages:

```
[someObject setValue:[someOtherObject getValue]];
```

Now, since we can't really use the square brackets in Perl (since they already do more than one job in Perl), I needed to find some not-too-different way to embed ObjectivePerl messaging into Perl. I did some hefty regexp searches of the Perl Standard Library and found that `~[object message]` wouldn't conflict with anything.

So, using *Filter::Simple*, I started writing the parser that would translate this new ObjectivePerl syntax into regular Perl method invocations. (Some of you might wonder why I implemented this as a source filter rather than write a real parser, perhaps using *Parse::RecDescent*. The answer to that is simple: Most of my work is in `mod_perl`, which retains all its loaded modules in memory. I didn't want any large modules—like *Parse::RecDescent*—expanding the memory footprint of my Apache processes.) Here's the basic gist of the filter:

```
use ObjectivePerl;
...your objective perl code...
no ObjectivePerl; # you don't need this if you're at the end of the file
```

After some tweaking, it actually worked, so I started working on the other aspects of ObjectivePerl that I wanted to include. I wrote some parsing goop that allowed the developer to define classes and their member methods using the Objective-C syntax:

```
@implementation ClassName [: ParentClass] [<Protocol, Protocol, ...>]
... class body ...
@end
```

Here's a silly example:

```
use ObjectivePerl;
@implementation MyClass

+ new {
    ~[$super new];
}
```

```
- someMethodWithArgument:$argument andAnother:$another {
    print "Argument: $argument\n";
    print "Another: $another\n";
}

@end
```

Methods are defined by either a leading "+" indicating that they are static methods, or a leading "-" indicating that they are instance methods. Static methods automatically have the variables *\$super* and *\$className* set, and instance methods get *\$super*, and of course, *\$self*.

So now you could instantiate that like this:

```
my $instance = ~[MyClass new];
```

and then invoke its one method like this:

```
~[$instance someMethodWithArgument:"Hey there" andAnother:$someValue];
```

What About Instance Variables?

I still needed to implement instance variables and visibility levels. It seemed like the *Filter::Simple* way of rewriting the source code before it's executed would suit this task, too, so I added the parsing of an instance variable declaration block, like this:

```
@implementation MyClass
{
    $someInstanceVariable;
    $someOtherInstanceVariable;
}

...
@end
```

Variables declared in the instance variable block are special insofar as your instance methods automatically have access to them. For example, you could write accessors like this:

```
- someInstanceVariable {
    return $someInstanceVariable;
}

- setSomeInstanceVariable:$value {
    $someInstanceVariable = $value;
}
```

which is a nice clean way to encapsulate your instance data. Of course, you can use the instance variables in any instance method, not just accessors:

```
- performSomeTask {
    my $taskVariable = $someInstanceVariable * 2;
    return ~[$self performSomeOtherTask:$taskVariable];
}
```

Right now, it's limited to scalars as instance variables, but I don't see that as being much of a problem since I use references all the time.

Instance Variable Visibility

For instance variables to be real, they need to be accessible by instances of the class, but also by instances of subclasses. Furthermore, the developer should be able to restrict access by subclasses to parent class instance variables, using visibility rules. This proved to be the trickiest part of the equation, and the solution is something that I won't go into here. However, the result is that there are two different visibility levels available to the developer:

protected and private. The common visibility level, public, has no application here because of the way Perl dereferences objects. In Java, if instance variable *rabbit* of object instance *hat* is public, then anyone can do this:

```
System.out.println(hat.rabbit);
```

which reaches right into *hat* and pulls out the value of *rabbit*. In Perl, there's no such dereferencing mechanism because objects are mostly blessed hashes, and their instance variables are just specified by the hash keys.

To declare visibility levels in ObjectivePerl, you simply use the *@protected* and *@private* directives when declaring the instance variables in the declaration block:

```
@implementation MyClass
{
    @protected: $this, $that, $theOther;
    @private: $myDeepestThoughts, $lingerie;
}
...
@end
```

So in this case, all five of those variables are accessible to methods in *MyClass*, but only *\$this*, *\$that*, and *\$theOther* are visible to any subclasses of *MyClass*. This doesn't mean that the variables are not there in subclasses, just that they can't be accessed directly from methods in the subclass. Here's an example to illustrate this:

```
use ObjectivePerl;
@implementation BaseClass
{
    @private: $private;
    @protected: $protected;
}

- protectedValue {
    return $protected;
}

- setProtectedValue: $value {
    $protected = $value;
}

- privateValue {
    return $private;
}

- setPrivateValue: $value {
    $private = $value;
}
@end

@implementation Subclass : BaseClass

- dumpProtectedValue {
    print $protected."\n";
}

- dumpPrivateValue {
    print ~[$self privateValue]."\n";
}
@end
```

Notice that the subclass can transparently refer to the *\$protected* instance variable, even though it did not declare it itself. However,

it cannot refer to *\$private* because that will cause a compile-time error. However, it does actually have the *\$private* variable and can set and get its value using the accessor methods defined by the parent class.

Why Bother?

When George Mallory was asked why he wanted to climb Mount Everest, he famously replied "Because it's there." Well, in many ways, this is the same kind of situation: For the most part, I did it because I could—for fun. I also find it useful, however, and I'm using it with some of my own projects. Your mileage may vary. I like

*Using Filter::Simple,
I started writing the parser that
would translate this new
ObjectivePerl syntax into regular
Perl method invocations*

the clean coding style, but it might not be your cup of tea. I also was intrigued by the CamelBones project (<http://camelbones.sourceforge.net/>) that allows you to write GUI applications for Mac OS X in Perl. It struck me that it would be an easy transition from Objective-C (which is the most common language for application development on Mac OS X) to an Objective-C-like version of Perl. From my experience so far, it is.

What Next?

Right now, there's quite a long to-do list, including:

- Adding visibility levels to methods (like Java has, where a method—like an instance variable—can be private, protected, or public).
- Implementing the Objective-C concept of Protocols (known as "interfaces" in the Java world).
- Allowing nonscalars as instance variables.
- Improving the debugging capabilities, which are presently very rudimentary.

But even in its current form, ObjectivePerl is very usable. Since it doesn't take anything away from regular Perl, you can mix and match at will. If you know how the runtime works, too, you can easily send messages to ObjectivePerl objects from regular Perl, and you can always invoke regular Perl methods from anywhere in ObjectivePerl. You can even subclass regular Perl classes in ObjectivePerl and it will work because underneath it all, your subclass is really just a regular Perl class with a lot of the goop handled for you.

If you're at all curious, go ahead and download it from CPAN and play around. I'd be happy to answer any questions you might have and more than happy to accept help in bug-stomping and improving it!

TPJ

Scoping: Letting Perl Do the Work for You

In 1982, my parents took a comedian's advice and bought a Texas Instruments TI-99/4a computer. It doesn't matter two decades later what computer I grew up with, so much as where it led. I embraced BASIC, not because it was spectacular, but because it was what was immediately available. And there was a lot of support for it out there back then. Armed with articles from *Compute!* and a few books, I began my adventure. Despite the primitive and limited nature of early BASIC implementations, I learned a thought process and developed a passion.

Those of you who remember early BASICs probably have fond memories of developing little gee-whiz programs that presented simple games and solved trivial problems. But despite its popularity, BASIC lacked many of the refinements that the previous 20 years of computer science had developed, refinements that, in the 20 years since, have become de facto standards. Subroutines in primitive BASICs consisted of segments of code accessed via *gosub*. There were no such things as user-defined functions, nor were there user subroutines with parameter lists. All variables were global, all code was on the same playing field, and as you may recall, that was a recipe for confusion as programs grew in size. It didn't help that it was common for us novice programmers to use such descriptive variable names as *x* and *a*. After a few lines of code, it became difficult to remember what different symbols stood for, and even more difficult to remember which symbols were already in use. This situation was mitigated by the fact that computers of the day usually were a little short on memory; a constraint that inhibited the creation of disastrously long BASIC programs. Languages such as Pascal overcame many of BASIC's shortcomings, but were considered beyond the reach of the hobbyists who took to BASIC.

Perl has had the benefit of growing up the brainchild of some very bright minds from our generation of computer science. But Perl isn't too proud to borrow an idea or two from the bright minds behind other programming languages, as well. It comes as no surprise then, that Perl implements many of the ideas of modularization and scoping found in languages like Pascal, C, Modula-II, and C++, ideas reinforced in my computer-science courses. Of course, Perl is equally likely to ignore some of the rules these languages established. But we're going to focus on a feature that Perl has (in my opinion) truly excelled at: scoping.

Dave may be reached via e-mail at daoswald@adelphia.net or on <http://www.perlmonks.org/~asdavido>.

Scope

Perl provides a rich set of tools to control scope that can be broken into three categories: package scope, dynamic scope, and lexical scope. First, let's talk about packages. Packages house namespaces implemented as glorified hashes. It is a common idiom for there to exist a 1:1 relationship between packages and files, but this is only a convention, and it's entirely possible to find packages that span files and files that contain multiple packages. None of that really matters: A package is a namespace and that's all we really need to remember.

A variable that lives in a package's namespace is called a "package global." Often the word "package" is left off, and we talk about "global variables." We almost always mean package globals. Even the main portion of a Perl script exists within a package: package *main*. Perl isn't particularly strict about who can look at a package's globals; other packages can inspect each others' package globals, and these variables can even be exported into other packages, including package *main*. But the general idea is that package globals are scoped to their package. Because of all the tricks we can play with package globals, they're incredibly useful in implementing Perl's modules. But this article is about letting scoping do the work for you. To this end, I want to shift the focus away from package globals. Their overuse also tends to lead to the sort of code we tried to leave behind 20 years ago. So let's move on to discuss the primary subject: lexicals.

I mentioned earlier that Perl provides three primary types of scope: package (the global symbol tables); dynamic (a means of manipulating the global symbol table); and lexical. Lexical scope is built and controlled by blocks. Blocks can exist in the form of files, packages, bare { ... } blocks, loops, conditionals, *eval* blocks, code blocks (as in *map { ... } @array* or *sort { \$b <= \$a } @array*), and subroutines. Where package globals are visible to other packages via the use of fully qualified names (i.e., *\$MyPackage::varname*), lexically scoped variables are not visible to the world outside of their scope. But Perl is a little smarter about how this works than a language such as C. "Auto" variables in C should never have a pointer to them passed outside of their scope. But with Perl, it's perfectly OK to pass references to a lexically scoped variable to the world outside of that variable's scope. More on this in a minute.

Lexical variables are declared using *my()*. They can be initialized at time of declaration. I'm going to assume that this is enough information on how to declare a lexical. If not, have a look at *perldoc -f my* and *perldoc perlsub* for more information. It's a good read.

Reference Counting

Perl, unlike C, handles garbage collection by itself with the reference-counting technique. When a lexical variable is declared, its reference count increments to one. When it falls from scope, the reference count is decremented. If it drops to zero, the variable is garbage collected. This is a bit of an oversimplification, but sufficient for our discussion. Now, what happens if in addition to the named variable, a reference to that variable also exists, perhaps at a broader scope? See the following example:

```
my $ref;
{
    # Create a block-bound lexical scope.
    my $var = 10;
    print $var, "\n";
    $ref = \ $var;      # Create a reference to $var, with a variable
                        # that is declared at broader scope.
}
# Close the block-bound lexical scope.
print $var, "\n";      # Nothing prints. $var is out of scope.
print $$ref, "\n";     # Dereference $ref, and thus, print '10'.
```

You can see from this example that though `$var` has passed out of scope, `$ref`, which holds a reference to `$var`, is keeping the contents of `$var` alive. `$var` is inaccessible by name, but by reference its value is still available. If `$ref` also passed out of scope, the reference count would be decremented again, and reaching zero would result in garbage collection.

Maintainable Style

One of the primary advantages of lexical scoping is that it promotes a maintainable programming style. If a programmer keeps the variable scope narrow, it becomes less important to worry about whether `$idx` is being used elsewhere in a script, so long as any preexisting use isn't needed within the current lexical scope. For example:

```
use strict;
use warnings;
my $var = 10;
{
    my $var = 20;
    print $var, "\n";
}
print $var, "\n";
```

The output will be 20, and then 10. This is because within the narrower lexical block, we've declared and defined a new `$var`, which means that whatever we do to or with `$var`, in that lexical block, the `$var` existing at a broader scope is unaffected. A declaration at narrower scope masks variables of the same name and type that exist at broader scopes. That means that each lexical scope can, if needed, become a new private namespace. `my()` is aptly named; just think of a lexical scope being the person talking: "My `$var` equals 10" (as opposed to some other scope's `$var`).

Lexical scoping obviously may be nested. Narrower scopes will have access to all the variables declared at broader scopes, so long as those variables haven't been masked by a declaration at the narrower scope. But completely separate scopes that aren't nested won't have such access. For example:

```
{
    my $this = 10;
}
{
    print $this, "\n";
}
```

Here we have two separate lexical blocks. They're not nested, thus `$this` is unavailable to be printed.

Destructors

When a lexical variable passes out of scope and its reference count drops to zero, it is garbage collected or destroyed. Normally, this has the simple effect that the memory consumed by the value of the variable is relinquished back to Perl, and the name that accessed it becomes inaccessible. But sometimes there are other side

When a lexical variable passes out of scope and its reference count drops to zero, it is garbage collected or destroyed

effects. Lexical scoping can be put to work taking advantage of those side effects for your benefit.

One example of the destructor doing more than just reclaiming memory is with the `open` command. If the filehandle being opened is a scalar with undefined value, it becomes a lexical filehandle. What happens when a lexical filehandle falls out of scope? The file gets closed implicitly.

```
my $filename = 'somefile.txt';
my $linecount = 0;
{
    open my $fh, '<', $filename or die $!;
    while( my $line = <$fh> ) {
        print $line;
        $linecount++;
    }
}
print $linecount, "\n";
```

This is a complete snippet (though contrived, and not really all that useful). By complete I mean that the filehandle held in `$fh` gets closed as soon as its enclosing lexical block falls out of scope. This leads to a common Perl file-slurping idiom that relies on lexical scoping to close the file being read:

```
my @lines = do{ open my $fh, '<', $filename or die $!;
                <$fh>;
            };
```

Now a file has been opened, its contents slurped into `@lines`, and its handle implicitly closed as the `do{...}` block finishes. The one caveat is that if you open a file for output and let the filehandle close implicitly via the magic of lexical scoping, you won't be able to perform the *or die \$!* error checking on the `close()` function.

Destructors also apply to object-oriented programming. If you define a `DESTROY()` method as part of your object, whatever code exists in that method will be executed as the object's entity reference falls out of scope. For example:

```

package MyPack;
sub new {
    my $class = shift;
    bless \my $self, $class;
}
sub DESTROY {
    print "Goodbye.\n";
}
1;

package main;
{
    my $obj = MyPack->new();
    print "The object was created.\n";
    print "Now we're going to let it fall out of scope.\n";
}
print "See, it was just destroyed.\n";

```

As you see, when the object's last reference falls out of scope, it is destroyed, and the *DESTROY()* method is invoked prior to the final garbage collection. This is useful anytime you have cleanup that needs to take place when an object disappears from existence.

DESTROY() also applies to tied entities. That means that if you use *tie* to tie a scalar to a class, you can define the *DESTROY()* method to carry out some task when the tied scalar falls out of scope.

Closures

No discussion of lexical scoping would be complete without mention of closures. This topic took me a little time to pick up, but it's really not all that complicated.

A closure is a situation where a lexical scope has closed and a reference to a sub defined within that scope is passed to the outside world. That closure sub still has access to the lexical variables that existed within the scope that just ended. Confusing? Look at this:

```

my $subref;
{
    my $value = 100;
    $subref = sub { return ++$value; }
}
print $subref->(), "\n";
print $subref->(), "\n";

```

Now is it a little clearer? *\$value* is inaccessible directly from outside its narrowly defined scope. Yet the reference to the sub created in that scope exists at a broader scope. The sub it refers to has full access to whatever variables existed in the scope in which it was defined. Thus, any time you call up that sub referred to by *\$subref* to do its duty, it is able to act upon *\$value*.

Accessors and Setters

In the object-oriented world, an accessor is an object method that accesses (or returns) data internal to the object. A setter is an object method that sets data internal to the object. Here's an object-oriented example:

```

package MyPack;

sub new {
    bless {}, shift;
}

sub setter {
    my( $self, $val ) = @_;
    $self->{VALUE} = $val;
}

```

```

}

sub accessor {
    my $self = shift;
    return $self->{VALUE};
}
1;

package main;

my $obj = MyPack->new();
$obj->setter("Hello world\n");
my $phrase = $obj->accessor();
print $phrase;

```

Though the topic of setters and accessors isn't strictly a discussion about lexical scoping, I provided that example as a means of introducing the fact that setters and accessors may also apply to closures:

```

my $setref;
my $getref;
{
    my $value;
    $setref = sub { $value = shift; };
    $getref = sub { return $value; };
}
$setref->( 100 );
my $closure_val = $getref->();
print $closure_val, "\n";

```

Putting the Pieces Together

Lexical scoping can and should be used to constrain a variable to the narrowest useful scope. This practice will aid in writing maintainable code; code where a minor change here won't ripple into a major disaster somewhere else, and code where it is easy (or at least possible) for someone to come along after the fact and understand what use a particular variable has.

I also hope to have illustrated that lexical scoping can be used to perform complex tasks through the use of destructors and closures. I encourage you to proceed from here to Perl's POD. In particular, *perlsub* and *perlref* will assist you in gaining a firmer grasp on what lexical scoping is all about. Finally, I hope that its use will help you to get more out of Perl.

TPJ



Secure Your Code With Taint Checking

Your web code is under constant attack. You may not notice it, but go look in your server logs. You'll find lines like this:

```
139.114.236.27 - - [03/Sep/2004:15:03:23 -0500] "POST /cgi-bin/formmail.pl
HTTP/1.0" 404 285
```

Automated robot programs started by no-good crackers are hunting down commonly known scripts at random, throwing bad data at random, trying to poke holes in security.

So what can happen if the bad guys find a poorly written program on a web server? Consider this overly simple example of a guestbook application. Somewhere on my web site, I have a page, `guestbook.html`, that looks like this:

```
Please leave your name and today's date<BR>
Date (MM-DD-YYYY): <INPUT TYPE="text" NAME="date"><BR>
Message: <INPUT TYPE="text" NAME="message"><BR>
<INPUT TYPE="submit">
```

and posts to `guestbook.cgi`.

My intent was to have a user visit the page, type "09-08-2004" in the date field, and "Hi, Andy, this is a great page you've got here" in the message field. Then, my handy little `guestbook.cgi` script processes the input for me and writes it into a date-stamped log file:

```
#!/perl -w
my $date = param('date');
my $logfile = "/var/guestbook/log-$date";
open( my $fh, ">>", $logfile )
    or die "Can't open $logfile: $!";
print ($fh) param('message'), "\n";
close $fh;
```

This script takes the date, builds a filename from it, and dumps the message into the file. It effectively creates a log file for each day that someone visits. But what if my visitor is not so nice? What if, instead of a date, he passes these values:

```
date="../../../../../../../etc/passwd"
```

Andy manages programmers for Follett Library Resources in McHenry, IL. In his spare time, he evangelizes about automated testing, shepherds the Phalanx project (<http://qa.perl.org/phalanx/>), and leads the Chicago Perl Mongers. He can be contacted at andy@petdance.com.

and

```
message="badguy:$1$kG6h18F$VY3ch9Iyi15BypV0c0SZn1:0:0::/bin/bash"
```

My little program will build `$logfile` as

```
/var/guestbook/log-$date../../../../../../../../etc/passwd
```

which is my `/etc/passwd` file. Then, my program will happily write that line to the password file, creating a user record called *badguy* that has root privileges on my box. My script, because I didn't tell it to check the format of the inputs passed in, has given the bad guy complete access to my computer. That's no fun for anyone except him.

What happened? My program got data that I wasn't expecting it to get and hadn't planned for. That assumption leads to the program writing out data to places I didn't intend. A simple fix involving Perl's *taint checking* functionality would have prevented this problem. With the addition of the `-T` command-line switch to the shebang line in `guestbook.cgi`, like so:

```
#!/perl -Tw
```

Perl would have complained as soon as my script was run:

```
Insecure dependency in open while running with -T switch
at guestbook.pl line 4.
```

This tells me that the program is trying to write to a file where the filename is tainted, and thus insecure. Thank Perl for my handy seat belt!

Turning on Taint Checking

Taint checking is a feature unique to Perl. Since it's part of the language runtime environment, I don't have to do anything other than turn it on. Perl will take care of the rest. However, since taint checking is so restrictive, and it doesn't cover everything you might think, it's important to know the details of how tainting works.

Taint checking is turned on whenever a program is run when the effective and real user or group IDs are different, on behalf of someone other than the owner. This is to prevent a program inadvertently performing unsafe actions with the privileges of someone else. When taint checking is on, any unsafe operations will throw a fatal error, and the program will stop.

I can force taint checking to be enabled with the `-T` command-line switch. There's also a `-t` command-line switch, where unsafe operations generate a warning instead of an error, but I never use it. Taint checking is for security; what good is having your program tell you "Hey, I just let you do something bad"? It may have a purpose, but I've never seen it.

Tainting is either on or off for the duration of the program. You can't turn it off while your program is running, like warnings with the `no warnings` pragma or modifying `$^W`. Tainting must also be turned on at the time the program starts with a command-line switch. This might be by calling my program like so:

```
$ perl -T taint-safe
```

or putting the `-T` on the shebang line at the top of the program:

```
#!/perl -T
```

If I call a Perl program with a `-T` in the shebang line, but execute it via Perl without a `-T`, I get the notorious "Too late for `-T`" error:

```
$ cat shebanged
#!/perl -T
....
$ perl shebanged
Too late for "-T" option at shebanged line 1.
```

Perl has to know immediately upon starting that taint checking is on, because the environment must get properly tainted. To get around this, either invoke Perl with the `-T`:

```
$ perl -T shebanged
```

or as a shell script:

```
#!/shebanged
```

"Too late for `-T`" isn't the clearest error message, and I've been trying to get a more obvious one in future versions of Perl. It's also possible to get this error if the `-T` isn't the first switch on the command line or shebang line. Always put `-T` first.

The Rules of Tainting

The principles of tainting are simple:

- Any data from outside the program is tainted. This includes any data that didn't start in the program, including data read from files, sockets, or even directory entries from `readdir()`.
- Tainted data may not be used in any operation that modifies files, directories, or processes, in an `eval`, or when invoking a subshell.
- Only scalars may be tainted. Arrays, lists, and hashes cannot be tainted, but individual elements can. Hash keys are never tainted, even if they are created from tainted scalars. Also, `undef` is never tainted.

Taintedness is pervasive. Any tainted data in an expression renders the whole expression tainted. That's obvious in a case like this:

```
my $filename = $tainted_dir . "/" . $tainted_filename
```

However, even if the expression can never be logically tainted, the taintedness rules apply:

```
my $zero = $tainted_data * 0;
# $zero is tainted, even though it's effectively constant.
```

This rule comes in handy when you want to check a value's taintedness. Here's the canonical `is_tainted()` function from the `perlsec` man page:

```
sub is_tainted {
    return ! eval { eval("###.substr(join("", @_), 0, 0)); 1 };
}
```

My script, because I didn't tell it to check the format of the inputs passed in, has given the bad guy complete access to my computer

Note how all the parameters are joined together into one string, and then a zero-length string is extracted from that. Even though the string is always empty, it will be tainted if any of `@_` is tainted.

The exception to the "any tainted data taints the entire expression" rule is the ternary conditional operator, so that in this case:

```
$x = $tainted_value ? "foo" : "bar";
```

`$x` will not be tainted, since this is the same as:

```
if ( $tainted_value ) {
    $x = "foo";
} else {
    $x = "bar";
}
```

Untainting

Once I have tainted data, I want to untaint it to use it. More precisely, I extract the good data from the tainted data using regular expressions and the numeric group match variables.

When I successfully match a string against a regular expression and use the parentheses to group an element of the regex, each subexpression matched is placed into the special variables `$1`, `$2`, and so on.

```
my $graffiti = "867-5309 Jenny";
if ( $graffiti =~ /(\\d\\d\\d-\\d\\d\\d\\d\\d)/ ) {
    $phone = $1;
    # $phone contains "867-5309"
}
```

In this example, I'm looking in a string for a simple phone number of three digits, a dash, and four digits. If it's found, it's put into `$1`. These capture variables are never tainted, and so can be used as part of my extraction of untainted data.


```
my $q = CGI->new;
my $user_input = $q->param( 'phone' ); # tainted
my $phone;
if ( $user_input =~ /^(\\d\\d\\d-\\d\\d\\d\\d)$/ ) {
    $phone = $1;
    # Do something with the phone number
}
```

Note that you must check the return value of the regex match to see if the string matched. It is not sufficient to check to see if *\$1* has a value in it, since a failed regex match will not clear the value in *\$1*. If the prior example was written as:

```
# BAD EXAMPLE: Do not do this
$user_input =~ /^(\\d\\d\\d-\\d\\d\\d\\d)$/;
if ( $1 ) {
    $phone = $1;
    # Do something with the phone number
}
```

then if the regex did not match, *\$phone* would get whatever value was last in *\$1* from the last successful match.

When I untaint, I want my regex to be as restrictive as possible. Note how my regexes have all been anchored at the beginning and end with the *^* and *\$* anchors (or *|A* and *|Z* if you prefer). Without them, my phone number regex would match a string like “something 867-5309 something else.”

If I’m concerned with security, I don’t want to take the approach of “take the good stuff out and leave the rest.” This is especially true in the case of validating form fields that I’ve created myself. If they’re not in the format I’m expecting, it must be because someone is being naughty.

How Not to Untaint

There are two bad ways to untaint. The first is to use an overly permissive regex, like */(.*)/*. That will match the entire string and put it in *\$1*. The result will be untainted, but it might as well still be tainted because nothing will have been checked with regards to its content. Don’t fall prey to this trap: Write your untainting expressions to be strict. The time will pay off later in added security.

The other bad way to untaint is to use the value as the key to a hash, and then retrieve the key. Perl’s hash keys aren’t full-blown scalars but a special case of string, and thus have no taintedness associated with them. Therefore, doing something like:

```
my @commands = <STDIN>; # tainted input from STDIN

# Hash keys aren't tainted
$commands{$_} = 1 for @commands;

foreach ( sort keys %commands ) {
    # Execute the erroneously untainted command
    # VERY dangerous!
    system( $_ );
}
```

is extremely dangerous. This code reads commands from standard input, untaints the strings, and then executes them via the system command. It has removed the tainting without checking the content of the strings.

Web and Tainting

Now that you know the security of having taint checking watch your back, you’ll want to start using it in as many apps as possible. For CGI programs, it’s simple: Put a *-T* in the shebang line of your program, and Perl will respect it.

If you’re using *mod_perl*, you’ll need to have the *PerlTaintCheck* directive set to *On* in your *httpd.conf*. This will turn on taint checking for every *mod_perl* handler under Apache. Recall that taint checking is on at the time the Perl interpreter starts. Since *mod_perl* actually embeds the interpreter in an Apache *httpd* executable, Perl starts when Apache starts. This can cause problems with multiple developers, or multiple applications on the same server where one app is careful with its untainting, and the other isn’t.

*Any tainted data in an
expression renders the whole
expression tainted*

Taint Checking with *CGI::Untaint*

Although you may be able to get by with the aforementioned *is_tainted()* function, there are a number of CPAN modules available to help your journey into enhanced security.

Untainting is most commonly used in CGI programs, so it’s no surprise to find *CGI::Untaint* at the top of the pack. *CGI::Untaint* combines the extracting of query parameters and untainting into one operation. For example:

```
my $q = new CGI;
my $untaint = CGI::Untaint( $q->Vars );
my $id = $untaint->extract( -as_integer => 'id' );
```

The *\$untaint* object is initialized with all the CGI parameters. Then, the *extract* method looks for a parameter called “*id*,” and validates it as an integer. If the *id* parameter is there, and passes the regex as an integer, *extract* returns the value, untainted. In any other case, it returns *undef*. You can also check for formats like printable characters, hexadecimal numbers, and so on.

The beauty of *CGI::Untaint* is its extensibility. The framework allows a programmer to add other formats of data to check for validity. The CPAN has add-on modules for data forms such as ISBNs, host names, IP addresses, and e-mail addresses. You can also write your own for your specific formats.

Even if you use *mod_perl*, you can still use *CGI::Untaint* by passing the args from your Apache request:

```
sub handler {
    my $request = shift;

    my $untaint = CGI::Untaint( $request->args );

    # Untaint as shown before...
}
```

Taint Checking with *DBI* and *Class::DBI*

Databases cause some problems for the programmer who’s not careful with taint checking. Since there’s no meaning of “tainted” outside of Perl, writing tainted data into a database and then reading

101 Perl Articles!



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

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

\$9.95 For subscribers to
The Perl Journal

\$12.95 For nonsubscribers to
The Perl Journal

\$24.95 To subscribe to
The Perl Journal and
receive *101 Perl Articles*

Go to

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

now!

it back out effectively untaints the data without checking it for validity. Worse, the data is in the database for an indeterminate amount of time after your program finishes running, so other programs might come along later and use it, not knowing that it hasn't passed stringent checking. Fortunately, since DBI 1.31 and higher, DBI provides support for tainting.

For each DBI handle, either database or statement, you can set the *TaintIn* or *TaintOut* parameters. If the *TaintIn* parameter is True, data being sent into DBI will be checked for taintedness. Currently, this includes "all the arguments to most DBI method calls," but may change in the future. The converse is the *TaintOut* parameter, which taints "most data fetched from the database" before it gets returned to your program. This most closely resembles the general principle of "any outside data is tainted." Finally, the *Taint* parameter is a combination of both *TaintIn* and *TaintOut*.

If you're wrapping your DBI calls with the excellent *Class::DBI* module, or its base class *Ima::DBI*, your tainting checks are taken care of for you. They turn on DBI's *Taint* parameter automatically.

Tainting for Module Authors and *Test::Taint*

Module authors should make sure their modules properly handle tainted data. If a module can't handle running under taint mode, it might be useless to a user who wants to run it in a taint-enabled application.

If you want to make sure that your module untaints data properly, or have code that returns data that the user expects to be tainted, you'll want to look at *Test::Taint*. Based on the standard *Test::Builder* framework, *Test::Taint* provides functions for verifying the functionality of your module.

For example, say I wrote a module that untaints hex values and I want a test file to verify its behavior. First, the test file will make sure that it's running under taint checking, create a test value, taint that value, and verify it's tainted:

```
taint_checking_ok();

# Make a tainted ID
my $hex = "deadbeef";
taint( $hex );
tainted_ok( $hex );
```

Then, the real testing happens. The *validate_hex* function, the one that I wrote, is called and the script verifies that the *\$newhex* value is untainted and is the correct value:

```
# Check your validator
my $newhex = validate_hex( $hex );
untainted_ok( $newhex );
is( $newhex, 'deadbeef' );
```

Test::Taint provided all of these test functions and makes checking a breeze.

Wrap-up

Since web-based attacks are trivial to implement and indiscriminate in their targets, it makes sense to use Perl's unique taint-checking feature by default in all your web code. Taint checking also provides a simple way to remind you not to do unsafe things with your code. If you write code with *use warnings* and *use strict* by default, start adding *-T* to those defaults.

TPJ



Detaching Attachments

brian d foy

I read my mail with Washington University's PINE and I do it on a remote server. I've been doing it that way for years and it works for me. Since I travel so much, I never know where I might be or which computer I might be using, but I can usually ssh to my shell account.

I can't look at most attachments since PINE is just a text thing. It isn't going to show me pictures or translate Word documents. Since I read this stuff on a remote account, I usually have to save the attachments, then move them into my web space so I can download them on whatever computer I may be using.

I never felt annoyed or motivated enough to fix this—until now, that is. I don't want to give up PINE, but I want to view the attachments with as little pain as possible. I wrote a script that I can pipe a message to using PINE's *enable-unix-pipe-cmd* feature (which you may have to turn on). When I read a message with an attachment that I want to save, I display the full message with the *h* command (full header mode) so PINE displays the entire message including all the attachments. I can scroll through all of this if I like, and it's the same thing my program will get as input. With the *|* command, PINE prompts me for a program name to send the message to; I called this program "detach" and placed it in my personal bin directory. I also created a symlink named "d" to reduce typing. This program is shown in Listing 1.

The program is deceptively short. I start off with the usual Perl invocation and *strict* and *warnings* declarations. The *ExtUtils::Command* module provides me a *mkpath()* function that acts like a portable *'mkdir -p'* so I can easily create new directo-

ries. The *File::Spec::Functions* module has the handy *catfile()* function that joins together path parts according to the preference of the current operating system. The *File::Path* module also has an *mkpath()* function, but it kills the script if it fails. I don't like that sort of interface, so I stick to *ExtUtils::Command*.

I try to use this whenever I need to work with paths so my script will have fewer portability issues. Both of these modules have come with Perl for a long time so feel free to use them liberally.

I also pull in *MIME::Parser* to do all of the hard work, which makes the script as short as it is. It knows how to parse and save the attachments. Although I normally don't like this sort of tight coupling and multiple action interface, in this case, it is just what I want.

Before I used this script, I defined the environment variable *ATTACHMENT_ROOT* in my shell configuration file. I have this set to a directory in the protected space of my web directory so I can download the file through a web browser. That takes care of the download hassle I had before. If I don't set this variable, I use my home directory.

I read the message from STDIN with one big chunk. The *do{}* idiom is a favorite of mine. It acts sort of like an inline subroutine: I get a block of code to define a scope and it returns the last evaluated expression. In this case, I set the *\$/* (the input record separator) variable to nothing, so the diamond operator (*<>*) reads the entire file at once. I store the whole message, headers and all, in *\$message*.

Once I have the message, I want to figure out where to store my attachments. I think everyone names their files the same thing, so I have a bunch of files named "tpr-article.doc" from different authors. I decided to create a directory for each sender, so I send the entire message to my *from()* subroutine to pull out the e-mail address, then store it in *\$from*. My e-mail address extractor is very simple, and it's the same thing I wrote about in my last *TPJ* article, "Pipelines and E-mail Addresses" (*TPJ*, August 2004). It is

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

not a complete (or “correct”) solution, but it’s good enough and it hasn’t failed me yet. If I was worried about weird e-mail address or *From* lines, I could use one of the e-mail parsing modules.

Before I go any further, though, I want to make sure that the string in the *\$from* address is something safe for a path. The *MIME::Parser::Filer* module, which does a lot of the work my script doesn’t show, can take a string and ensure it’s something I can safely use. If it can’t rescue a bad e-mail address, it returns *undef* and I use “malformed-sender” as a default.

I use *catfile()* to join my base directory and sender directory. As I said earlier, *catfile()* does the right thing for the operating system. In the next line, I use the *do{}* block to create a scope that I can modify with *unless()*. If the directory already exists, I skip this step, but if it doesn’t, I put my new directory name into a *local()* version of *@ARGV* then call *mkpath()*, which uses the val-

ues in *@ARGV* as its arguments. It’s an odd bit of history, but that’s the way it works.

Once the directory exists, I tell *MIME::Parser* that I want to use that directory to store the files. This is where my attachments will show up. After I set the directory, I call *parse_data()* and all the interesting stuff happens.

The files end up in one of my private web directories, so I go to that URL, which is just a directory listing, then choose the link to the e-mail address of the message. There are my attachments. It’s not as easy as clicking a link in web mail, but now everything that everyone sends me gets stored in an easily accessible web directory.

TPJ

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

Listing 1

```
#!/usr/local/bin/perl5.8.0
use strict;
use warnings;

use ExtUtils::Command qw(mkpath);
use File::Spec::Functions qw(catfile);
use MIME::Parser;

my $Base = $ENV{ATTACHMENT_ROOT} || $ENV{HOME};
my $message = do { local $/; <> };
my $from = from( \$message );

my $parser = MIME::Parser->new();
$from = $parser->filer->exorcise_filename($from)
    || 'malformed-sender';

my $path = catfile( $Base, $from );
```

```
do { local @ARGV = ( $path ); mkpath; } unless -d $path;

$parser->output_dir( $path );

my $entity = $parser->parse_data( $message );

sub from
{
    my $message = shift;

    my( $from ) = $message =~ m/^From:\s+(.*)/mg;

    $from =~ s/\s* \(.?\)\ \s*//x;
    $from =~ s/. * < (. * .) > .*/$1/x;

    return $from;
}
```

TPJ

Subscribe now to

Dr. Dobb's E-mail Newsletters

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

- ✓ **AI Expert Newsletter.** Edited by Dennis Merritt; the AI Expert Newsletter is all about artificial intelligence in practice.
- ✓ **Dr. Dobb's Linux Digest.** Edited by Steven Gibson, a monthly compendium that highlights the most important Linux newsgroup discussions.
- ✓ **Dr. Dobb's Software Tools Newsletter.** Having a hard time keeping up with new developer tools and version updates? If so, Dr. Dobb's Software Tools e-mail newsletter is just the deal for you.
- ✓ **Dr. Dobb's Math Power Newsletter.** Join Homer B. Tilton and expand your base of math knowledge.
- ✓ **Dr. Dobb's Active Scripting Newsletter.** Find out the most clever Active Scripting techniques from Mark Baker.

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



Unicode in Perl

Simon Cozens

Two hundred and fifty five characters really ought to be enough for anyone. I've lost count of how many times I've heard this statement or similar sentiments expressed when it comes to dealing with Unicode and the more general question of character encodings.

However, this kind of ASCII-centric thinking is becoming a liability. As Harald Tveit Alvestrand put it in RFC1766, "There are a number of languages spoken by human beings in this world," and the Unicode Standard was designed to be a way to make it easy for data from all kinds of environments, languages, and scripts to play nice together.

Until Unicode came along, the world was in a mess (at least in terms of data processing). Anyone who wanted to represent any kind of nonLatin character had to cobble together their own set of important characters to live in the top 127 character codepoints generated when we all moved from 7-bit ASCII to 8 bits. Unfortunately, when everyone has their own idea of what character 160 means, depending on whether they're coming from ASCII extensions to support Hebrew, Cyrillic, or the plain old European accents defined in ISO 8859-1—data interchange is impossible.

To make things worse, the Chinese, Japanese, and Koreans got involved with data processing and soon realized that the 127 spare codepoints just weren't enough to put a dent in their data processing needs. With over 2000 kanji characters in general use in Japan, plus two alphabets of about 85 characters, 255 characters start to look a bit piffling.

If you need more than 255 characters, you're not going to be able to store each character in a single 8-bit byte. Going to 16-bit bytes

was not an option, so they devised any number of encoding mechanisms to shoehorn huge numbers of codepoints into 8-bit bytes—EUC, JIS, Shift-JIS, Big-5, and many others. Many of these try to maintain compability with ASCII by keeping the semantics of the bottom 127 characters and using the top half as "shift" characters, which introduce a wider character. Now we not only have many incompatible assignments of codepoints to characters, we have multiple incompatible ways of representing "wide" characters (more than a single byte) on disk or in memory.

Unicode came along to sort all this out. It introduced a single mapping between codepoint and character for every written script on Earth—the Unicode character set. It also proposed a number of standard ways to lay out these characters when they get bigger than a single byte—the UTF-8, UTF-16, and UTF-32 Standards (ss well as some extra ones like UTF-7 that nobody seriously uses).

Perl caught the Unicode bandwagon pretty early, thanks in part to Larry Wall's foresight (not to mention his love of Japan and its language), but many of Perl's programmers aren't on board yet. This month, I'm going to try to turn you from an ASCII-phile to a Unicode-aware programmer.

Generating and Munging Unicode Data

First, though, how do we create and deal with Unicode characters? Perl tries to make this as natural as possible. For instance, where *chr* and *ord* could previously deal only with values up to 255, they can now deal with values up to 4,294,967,295, at least on my poor old 32-bit computer.

Similarly, string escapes have been extended to deal with characters higher than `\xFF`. However, to keep Perl compatible with old programs, which may say `"\x0dabc"` if you want an escape sequence longer than two characters, you must surround the character code with curly braces, like so:

```
print "\x{263a}\n"; # Prints a white smiling face
```

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

Now it may not be immediately apparent that codepoint 263a is a white smiling face, so Perl provides the *chardnames* pragma, which allows you to specify characters by name, using the *\N* escape:

```
use chardnames ':full';
print "\N{WHITE SMILING FACE}\n";
```

Since the names themselves may not be that easy to find unless you have a copy of the Unicode Standard on hand—and may be a little unwieldy even if you do—you can also specify a short name consisting of the script name and the character name. For instance:

```
use chardnames ':short';
print "\N{katakana:sa}\N{katakana:i}\N{katakana:mo}\N{katakana:n}";
```

This will print out my name in Japanese. Of course, if I'm handling lots of Japanese, it gets rather tedious to type *katakana*: every time, so we can also say:

```
use chardnames qw(katakana);
print "\N{sa}\N{i}\N{mo}\N{n}\n";
```

Now we have a bunch of Unicode data to deal with. What can we do with it? Well, the first thing to note is that we can do anything we usually do with Perl. Nothing has changed now that Unicode data appears on the scene.

True, we're dealing with characters that are now wider than a single byte, but that's OK. Perl does the right thing with them:

```
print length("\N{sa}\N{i}\N{mo}\N{n}"); # prints 4
```

One neat extra thing that we can do with Unicode data is to use extended regular expressions. For instance, the Unicode Standard defines a set of properties that each character may have, and we can use regular expressions to match these properties. I deal with a kanji dictionary, which contains kanji headwords, followed by a mixture of codes and indexes that mean very little to me, and phonetic readings in the katakana and hiragana scripts. We'll see later how I read in the dictionary, but I can extract the hiragana readings like this:

```
while (<KANJI.DIC) {
    my @readings= /\p{Hiragana}+/g;
    /\p{Han}+/ and print "$1: @readings";
}
```

"Han" is the property descriptor for a Chinese kanxi or Japanese kanji character. For a full list of Unicode properties, see the Unicode Standard.

Perl's Unicode Support

Perl's own support for Unicode has developed and matured over the years, after a pretty shaky start. Not only that, but the nature of the support and what Perl has offered in terms of Unicode support has changed. Writing with the benefit of hindsight, I can now tell you about what Perl can do at the moment—regardless of what it was supposed to be all along. We can mainly ignore all of the motivations and all of the little hacks along the way, and talk about the real world.

But first, a bit of history so we are clear on what's possible with particular Perl versions. The first Perl release to support any kind of Unicode data was Perl 5.6.0. You could generate Unicode characters as we've previously discussed, and you could print that data out, more or less, but there was no other way of getting Unicode data from files or from other sources into your application as Unicode. This was a bit useless, really. It also didn't help that

Perl didn't have a clear strategy for what happened when Unicode data hit nonUnicode data, and it's here that an important distinction arises, which we'll look at in a second.

These problems were mostly sorted out through 5.6.1 and gone by Perl 5.6.2, but the problem of getting Unicode data into Perl still remained. Work began in 5.6.1 or so to fix this using the *Encode* module, and this has only been usable since around Perl 5.8.2. So while it is possible to do some Unicode-related work in 5.6.2 if you're careful, real Unicode applications ought to be based on 5.8.2 and above.

*With over 2000 kanji
characters in general use in
Japan, plus two alphabets
of about 85 characters,
255 characters start to
look a bit piffling*

The Big Lie

I've been claiming that Perl now supports Unicode, but to be honest, that's a bit of a lie. Perl supports data encoded in the UTF-8 representation and knows what to do with it if that data is Unicode. It doesn't ever know whether that data really does represent Unicode or not.

Let's suppose we're dealing with a string of Japanese data (as I reasonably often do) and let's further suppose we know nothing about Unicode at all. We're just an ordinary Perl 5 application merrily handling Japanese text, which is encoded in the EUC encoding often used for UNIX-based Japanese data processing:

```
my $hello = "\272\243\306\374\244\317\241\242\300\244\263\246";
print $hello, "\n"; # Prints "Hello world" on an EUC terminal
print length($hello) # 12 bytes
```

Now we want to play in the Unicode world and add our familiar smiley face to the end of our "hello world" greeting:

```
my $smiley = "\N{WHITE SMILING FACE}";
print $hello . $smiley;
```

At this point, we have a problem. Perl has absolutely no idea that this data is Japanese EUC. It could be in any legacy encoding under the sun. And now we want to append a Unicode string to the end. What's Perl going to do?

Well, there's very little it can do. It knows that the string on the right is Unicode data, but it can't assume very much about the string on the left. What it does do is rely on a flag that marks a string as being represented internally as UTF-8. It further assumes that when it sees a string that isn't represented as UTF-8, this should be treated as ISO-8859-1. Since our Japanese data isn't ISO-8859-1, madness will soon ensue.

Perl will "upgrade" the string to UTF-8, but it doesn't know how to convert it to Unicode. What we end up with is some -

UTF-8-encoded Japanese EUC data, not UTF-8-encoded Unicode data, and this is no good to man or beast.

Encode—Dealing with Legacy Data

So what can we do about legacy data that isn't ISO 8859-1? At this point, the *Encode* module becomes useful. We can't tell Perl what encoding we're dealing with, but we can ask Perl to translate everything to Unicode for us, and use that as a lingua franca—one of the things it was precisely designed to do.

*While it is possible to do some
Unicode-related work in 5.6.2 if
you're careful, real Unicode
applications ought to be based on
5.8.2 and above*

Let's take that same EUC string—the Japanese for “Hello, world”:

```
my $hello = "\272\243\306\374\244\317\241\242\300\244\263\246";
```

and use *Encode* to translate it from Japanese-EUC into Unicode:

```
my $hello_uni = decode("euc-jp", $hello);
```

Where before we were dealing with the string as a binary sequence of bytes, we're now dealing with it as Unicode characters. This is not just our useless EUC-UTF-8 mix, but real, honest-to-goodness Unicode.

```
print length($hello);  
print length($hello_uni);
```

At this point, all of our Unicode slicing-and-dicing, including Unicode-aware regular expressions, will work properly on *\$hello_uni*.

Once we've finished munging our data, of course, we might want to put it back into the EUC format we began with. Once again, *Encode* helps out with the predictably named *encode* routine:

```
open OUT, ">sliced-hello.euc" or die $!;  
print OUT encode("euc-jp", $hello_uni);
```

To find out what encodings *Encode* supports, you can say:

```
use Encode;  
print Encode->encodings("all");
```

So, for instance, we might want to create ourselves a Unicode transcoder—that is, something that takes data in one format and spits it out in another encoding. This is something I end up doing rather often, so I came up with the following program:

```
#!/usr/bin/perl  
use Encode;  
my ($from, $to) = splice(@ARGV,0,2); ($from && $to) or usage();  
while (<>) {  
    my $unicode;  
    eval { $unicode = decode($from, $_) };  
    if ($@ =~ /unknown encoding/i) { usage() }  
    eval { print encode($to, $unicode) };  
    if ($@ =~ /unknown encoding/i) { usage() }  
}  
  
sub usage {  
    die qq{  
$0 - $0 <to> <from> [<file> ...]  
}
```

This acts as a filter, encoding data from the first character set to the second. Available character sets are:

```
}, map { sprintf("%t%s\n", $_) } Encode->encodings("all");  
}
```

However, there's yet a neater way to do things. If you have *Encode* available, you also have the *PerlIO* module, which hooks into Perl's IO streams to control how file access is done. *PerlIO* is a mechanism that can be used to add filters onto a filehandle: One that automatically strips the newlines, for instance, or reads files that are gzipped, or even bypasses standard IO altogether and reads files directly into memory with *mmap*. *Encode* hooks into this *PerlIO* framework to read and write files through character set encoding or decoding. For instance, to read a Russian file from a Windows computer, using the koi-8 encoding, you can say:

```
open IN, "<:encoding(koi8-r)", "russian.txt" or die $!;
```

And to write it out again for use on a Russian Mac running System 9, you would say:

```
open OUT, ">:encoding(MacCyrillic)", "russian.mac" or die $!;  
while (<IN>) { print OUT $_ }
```

So, if you're content with a little simplicity, you can slim your transcoder down to:

```
#!/usr/bin/perl -p  
BEGIN {  
    binmode(STDIN, ":encoding(.shift(@ARGV).)");  
    binmode(STDOUT, ":encoding(.shift(@ARGV).)");  
}
```

Dealing with the Outside World

The final piece of the Unicode puzzle comes when you need to send or receive UTF-8 data from files, or send to other applications that may or may not know anything about Unicode themselves—such as databases, which just store the data, not caring about its semantics.

To store data as Unicode is easy enough—you just do it. If you write to a filehandle with data that Perl thinks contains Unicode characters—that is, has the UTF-8 flag set—Perl will write the UTF-8 representation of the string to the file:

```
open OUT, ">smiley.txt" or die $!;  
use charnames ':full';  
print OUT "\N{WHITE SMILING FACE}\n";
```

This will work just fine, but Perl will issue a warning when any multibyte characters are emitted:

```
Wide character in print at smile.pl line 3.
```

In order to tell Perl that it's OK to send the output as UTF-8, you can set a flag on the filehandle:

```
binmode(OUT, ":utf8");
```

Similarly, if you have a file that contains UTF-8 data that you want to recognize as such, you can set the same flag using *binmode* again on the input filehandle:

```
open IN, "smiley.txt" or die $!;
$a = <IN>; chomp $a;
print length $a # 3 bytes - not marked as Unicode
close IN;
```

```
open IN, "smiley.txt" or die $!;
binmode IN, ":utf8";
$a = <IN>; chomp $a;
print length $a # 1 character - marked as Unicode
```

Indeed, this is the usual and best way of getting Unicode data into a Perl application. Unfortunately, files are not the only places where you might receive UTF-8-encoded strings. We might read data from a socket, or receive it via DBI from a database, or as I had to do recently, read it from the middle of another binary file.

This last case is particularly interesting—you can't read the whole binary file as though it were UTF-8 because you really want to treat it as a stream of bytes; however, when you get to the part representing a string, you need Perl to treat it as a UTF-8 string, and work character-wise.

The way to do this is to use *utf8* as just another encoding. You have data that you know is in UTF-8 and you want Perl to turn it into Unicode data, so you say:

```
# String is packed with the length first
my $len;
read(BIN, $len, 4);
my $len_bytes = unpack("N", $len);

# Now read the string
my $str;
read(BIN, $str, $len_bytes);

# Make a UTF8-aware copy
my $utf8 = decode("utf8", $str);
```

There is another way to do this, which is a little messier, but I recommend it nonetheless. *Encode* can optionally export a subroutine called *_utf8_on*. As its name implies, this is an internal routine in that it directly messes with Perl's internal representation of the string, turning on the bit that says this data is UTF-8. I prefer this, however, because it is efficient, self-documenting, and easier to understand than trying to work out from what *decode("utf8", \$str)* is decoding into what.

Finally, you may have to deal with situations where you don't want to end up with your Unicode data as Unicode. For instance, you have a bunch of database records about your company's contacts in Eastern Europe that you need to have inserted into your master contacts database. Unfortunately, even though you are an educated and progressive programmer, and have stored everything correctly in Unicode, Headquarters is full of people for whom ISO-8859-1 is a recent advance over 7-bit ASCII. What will you do for your friend in Čžrný?

Here is a problem where you are guaranteed to lose information. You want to represent a character that simply can't be represented in the character set you have to deal with. Your choice is how much information you want to lose. If you take the obvious approach, and say:

```
print decode("iso-8859-1", "C\u{17e}rny");
```

Encode will helpfully substitute in a "substitute character" for the letter that cannot be represented, and you'll end up with "C?rny." This is acceptable so far, but you should be thankful that you're not dealing with completely nonLatin alphabets, as mail to the Korean city of ??? is guaranteed not to arrive.

If you need to lose less information, you could try the wonderful *Text::Unidecode* module, which tries to turn Unicode strings into "plain text." For example:

```
use Text::Unidecode;
print unidecode("\x{d478}\x{c0b0}"); # pusan
```

It's not perfect, but it's certainly better than a stream of question marks. When you still need to communicate with ASCII dinosaurs, *Text::Unidecode* will give them pretty much what they deserve.

Unicode for All!

Thankfully, though, the world is getting more and more Unicode aware. As we move into global business, working with more countries, languages, and scripts, the importance of Unicode will continue to grow. Most of the time, it takes very few changes to make an application aware of the possibility of Unicode text or to deal with that text when it arises, so there's really no excuse for not making your code Unicode compliant—do it now, and it'll save time and effort later.

TPJ

**Fame & Fortune
Await You!**

**Become a
TPJ
author!**

The Perl Journal is on the hunt for articles about interesting and unique applications of Perl (and other lightweight languages), updates on the Perl community, book reviews, programming tips, and more.

If you'd like share your Perl coding tips and techniques with your fellow programmers – *not to mention becoming rich and famous in the process* – contact Kevin Carlson at kcarlson@tpj.com.



XML Publishing With AxKit

Jack J. Woehr

XML Publishing with AxKit covers a new Perl CPAN tool that manages, collates, and transforms XML to output specs; for example, HTML, PDF, and the like. It's conceptually a generalization or factor of something like DocBook. It's useful and useable to the Perl CGI programmer, to the technical writer, and to the staff programmer at a publishing house.

Author Kip Hampton does a reasonably good job of identifying the niche and a better job of taxonomizing AxKit itself. Rather than burble 1990's-style about the wonders of XML, I would have started the book, "If you need XML transforms and you're surfeited with Java..." because that is the case here. Nothing that AxKit does in the field of XML transforms is left undone by Java, and that which is done in Java is done by more programmers using more mature tools than AxKit. But since many, if not most, programmers are surfeited with Java, they would have read on anyway.

XML Publishing with AxKit hyperbolically asserts that AxKit "turns Apache into an XML publishing and application server." There's a lot more to application serving than publishing transforms. The management of transactional data objects is the crux of most or all real-world web applications and the subject matter of most application-server XML. AxKit provides no data object facilities of the sort that Enterprise Java does. The AxKit enthusiast, having built the cart first, may turn to Perl DBI and discover that there is insufficient horsepower for high-volume, real-time transaction processing. Not that such things aren't done in Perl, but the Perl programmer doing transaction processing will usually spend a higher percentage of his/her time writing functions as opposed to describing data objects than will an Enterprise Java programmer. Perl programmers are largely those who prefer functional to declarative programming, so I'm probably worrying about nothing.

AxKit, like all modern tools—especially Perl CPAN tools—is primarily a pipeline of other tools. As with other such pipelines,

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. He can be contacted at <http://www.softwoehr.com/>.

XML Publishing with AxKit

Kip Hampton

O'Reilly Media, 2004

216 pp., \$29.95

ISBN 0596002165

extensibility and integration with tools such as XPathScript is its strength. However, AxKit is also very much a work in progress. AxKit doesn't support Apache 2, nor does it support recent versions of libxml2. Nonetheless, if you're bound and committed to doing it in Perl (whatever "it" is), then AxKit, or what AxKit becomes when it is more fully developed by its creators and maintainers, is the way to do it. It's a timely and clever idea that brings sophisticated programmatic XML transformations within reach of the scripter.

The book has the potential to baffle the less experienced user at installation time, to baffle them to the point of never completing the installation of AxKit. It took me about six hours of fiddling around to install AxKit completely and get it running. Among the problems were:

- Difficulty building prerequisites via the AxKit install.
- Version problems with libxml2.
- Having to manually modify *XML::LibXML* Version 1.58's perl-libxml-mm.c (creating a { block } to enclose the declaring and accessing of the local variable *xmlChar **, decoded starting at line 968) to get it to compile: Classic C doesn't allow bare variable declarations except at the top of a function.

- The book incorrectly (or outdatedly) indicates that the sample configuration file provided with AxKit is `examples/example.conf`, which does not exist. Currently, there is only the file `demo/axkit.conf` to be included from Apache's `httpd.conf`, and an `.htaccess` file provided by AxKit that provides other hints to the server.

It's true that I'm not sitting in front of the latest-and-greatest canned Linux release: I'm maintaining my own rich, variegated, and mature open-source software toolchain on Solaris 9. It's not an exotic environment, nor is it in any way hostile to open-source development or Perl. One surmises that the author, while writing this book sitting in front of an ideal platform optimized for AxKit, either neglected to mention the sorts of problems that ordinary users might encounter or shrugged them off as endemic to the CPAN approach, where modules can change asynchronously with respect to both underlying compiled libraries and companion modules.

The best procedure for building a minimally useful AxKit is as follows:

1. Download, configure, build, and install (or make sure you already have) Perl 5.8.x.
2. Download, configure, build, and install libxml2 Version 2.6.8 (<http://xmlsoft.org/>).

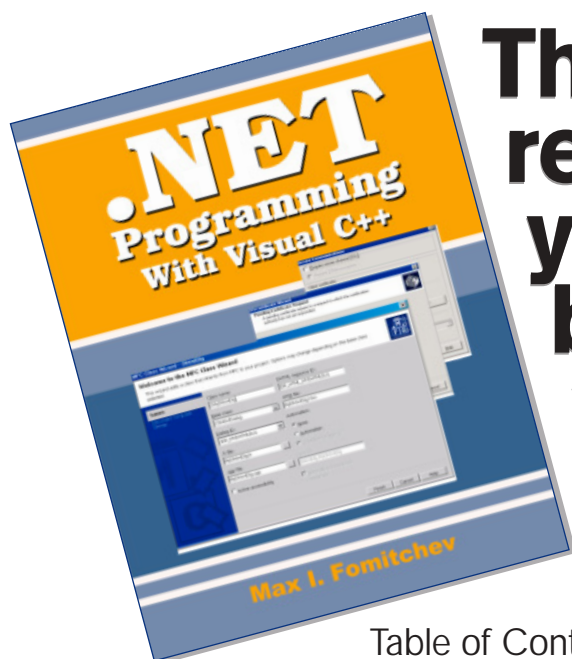
3. Download, configure, build, and install libxslt (<http://xmlsoft.org/>), which is apparently not version sensitive.
4. Use the Perl `cpm` tool to install `Apache::Test` and `Error` (if you have not already done so).
5. Use the Perl `cpm` tool to install `XML::LibXML` and `XML::LibXSLT`.
6. Use the Perl `cpm` tool to install AxKit.

If, in the course of the build, you discover that mandatory prerequisites not mentioned above are missing, the easiest and most reliable procedure is to control-C out of `cpm`, and restart and install the prerequisite directly, rather than allowing the `cpm` tool to chain multiple prerequisite builds. Build chaining is a nice feature of the `cpm` tool, only it doesn't always work well, and it works spectacularly poorly in the AxKit build.

The web site for the book is <http://www.oreilly.com/catalog/xmlaxkit/> where the usual table of contents, sample chapter, and errata can be found. As of this writing, none of the example code from the book is provided.

TPJ

XML Publishing with AxKit
*hyperbolically asserts that AxKit
"turns Apache into an XML
publishing and application server"*



The .NET resource you've been waiting for!



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

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

Get your copy now!

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

Source Code Appendix

brian d foy “Detaching Attachments”

Listing 1

```
#!/usr/local/bin/perl5.8.0
use strict;
use warnings;

use ExtUtils::Command qw(mkpath);
use File::Spec::Functions qw(catfile);
use MIME::Parser;

my $Base      = $ENV{ATTACHMENT_ROOT} || $ENV{HOME};
my $message = do { local $/; <> };
my $from      = from( \ $message );

my $parser = MIME::Parser->new();
$from = $parser->filer->exorcise_filename($from)
    || 'malformed-sender';

my $path = catfile( $Base, $from );
do { local @ARGV = ( $path ); mkpath; } unless -d $path;

$parser->output_dir( $path );

my $entity = $parser->parse_data( $message );

sub from
{
    my $message = shift;

    my( $from ) = $$message =~ m/^From:\s+(.*)/mg;

    $from =~ s/\s* \(.?\)\s*//x;
    $from =~ s/.* <(.?@.?)> .*/$1/x;

    return $from;
}
```

TPJ