Appendix B. Beyond the Llama

We've covered a lot in this book, but there's even more. In this appendix, we'll tell you a little more about what Perl can do, and give some references on where to learn the details. Some of what we mention here is on the bleeding edge and may have changed by the time you're reading this book, which is one reason we frequently send you to the documentation for the full story. We don't expect many readers to read every word of this appendix, but we hope you'll at least skim the headings so that you'll be prepared to fight back when someone tells you, "You just can't use Perl for project X because Perl can't do Y."

The most important thing to keep in mind (so that we're not repeating it in every paragraph) is that the most important part of what we're *not* covering here is covered in *Intermediate Perl*, also known as "the Alpaca." You should definitely read the Alpaca, especially if you'll be writing programs that are longer than 100 lines (either alone or with other people). Especially if you're tired of hearing about Fred and Barney and want to move on to another fictional universe featuring seven people who got to spend a lot of time on an isolated island after a cruise!

After *Intermediate Perl*, you'll be ready to move on to *Mastering Perl*, also known as "the Vicuña." It covers the everyday tasks that you'll want to do while programming Perl, such as benchmarking and profiling, program configuration, and logging. It also goes through the work you'll need to do to deal with code written by other people and how to integrate that into your own applications.

In *Perl New Features*, brian covers the features added to Perl starting with v5.10 up to the current version. As an ebook, it's easy to update this book for new versions of Perl.

There are many other good books to explore. Depending on your version of Perl, look in either perlfaq2 or perlbook for many recommendations,

especially before you spend your money on a book that might be rubbish or out of date.

Further Documentation

The documentation that comes with Perl may seem overwhelming at first. Fortunately, you can use your computer to search for keywords in the documentation. When searching for a particular topic, it's often good to start with the perltoc (table of contents) and perlfaq (frequently asked questions) sections. On most systems, the *perldoc* command should be able to track down the documentation for Perl, installed modules, and related programs (including *perldoc* itself). You can read the same documentation online, although that is always for the latest version of Perl.

Regular Expressions

Yes, there's even more about regular expressions than we mentioned. *Mastering Regular Expressions* by Jeffrey Friedl is one of the best technical books we've ever read. It's half about regular expressions in general, and half about Perl's regular expressions, which many other languages incorporate as Perl-Compatible Regular Expressions (PCRE). It goes into great detail about how the regular expression engine works internally, and why one way of writing a pattern may be much more efficient than another. Anyone who is serious about Perl should read this book. Also see the perlre documentation (and its companion perlretut and perlrequick in newer versions of Perl). And there's more about regular expressions in *Intermediate Perl* and *Mastering Perl* as well.

Packages

Packages allow you to compartmentalize namespaces. Imagine that you have 10 programmers all working on one big project. If someone uses the global names `$fred`, `@barney`, `%betty`, and `&wilma` in their part of the project, what happens when you accidentally use one of those same

names in your part? Packages let you keep them separate; I can access your `$fred`, and you can access mine, but not by accident. You need packages to make Perl scalable so that you can manage large programs. We cover packages in great detail in *Intermediate Perl*.

Extending Perl's Functionality

One of the most common pieces of good advice heard in the Perl discussion forums is that you shouldn't reinvent the wheel. Other folks have written code that you can put to use. The most frequent way to add to what Perl can do is by using a library or module. Many of these come with Perl, while others are available from CPAN. Of course, you can even write your own libraries and modules.

Modules such as `Inline::C` allow you to easily hook up C code to Perl.

Writing Your Own Modules

In the rare case that there's no module to do what you need, you can write a new one, either in Perl or in another language (often C). *Intermediate Perl* covers how to write, test, and distribute modules.

Databases

If you've got a database, Perl can work with it. We've already seen the `DBI` module briefly in Chapter 11.

Perl can directly access some system databases, sometimes with the help of a module. These are databases like the Windows Registry (which holds machine-level settings), or the Unix password database (which lists which username corresponds to which number, and related information), as well as the domain-name database (which lets you translate an IP number into a machine name, and vice versa).

Mathematics

Perl can do just about any kind of mathematics you can dream up. The `PDL` module (for Perl Data Language) provides high-powered ways to do tricky math.

All of the basic mathematical functions (square root, cosine, logarithm, absolute value, and many others) are available as built-in functions; see the perlfunc documentation for details. Some others (like tangent or base-10 logarithm) are omitted, but those may be easily created from the basic ones, or loaded from a simple module that does so. (See the `POSIX` module for many common math functions.)

Although the core of Perl doesn't directly support complex numbers, there are modules available for working with them. These overload the normal operators and functions so that you can still multiply with `*` and get a square root with `sqrt`, even when using complex numbers. See the `Math::Complex` module.

You can do math with arbitrarily large numbers with an arbitrary number of digits of accuracy. For example, you could calculate the factorial of two thousand, or determine π to ten-thousand digits. See the `Math::BigInt` and `Math::BigFloat` modules.

Lists and Arrays

Perl has a number of features that make it easy to manipulate an entire list or array.

In Chapter 16, we mentioned the `map` and `grep` list-processing operators. They can do more than we could include here; see the perlfunc documentation for more information and examples. And check out *Intermediate Perl* for more ways to use `map` and `grep`.

Bits and Pieces

You can work with an array of bits (a `bitstring`) with the `vec` operator, setting bit number 123, clearing bit number 456, and checking to see the

state of bit 789. Bitstrings may be of arbitrary size. The `vec` operator can also work with chunks of other sizes, as long as the size is a small power of two, so it's useful if you need to view a string as a compact array of nybbles, say. See the perlfunc documentation or *Mastering Perl*.

Formats

Perl's formats are an easy way to make fixed-format, template-driven reports with automatic page headers. In fact, they are one of the main reasons Larry developed Perl in the first place: as a Practical Extraction and *Report* Language. But, alas, they're limited. The heartbreak of formats happens when someone discovers that they need a little more than what formats provide. This usually means ripping out the program's entire output section and replacing it with code that doesn't use formats. Still, if you're sure that formats do what you need, *all* that you'll need, and all that you'll *ever* need, they are pretty cool. See the perlform documentation.

Networking and IPC

If there's a way that programs on your machine can talk with other programs, Perl can probably do it. This section shows some common ways.

System V IPC

All of the standard functions for System V IPC (interprocess communication) are supported by Perl, so you can use message queues, semaphores, and shared memory. Of course, an array in Perl isn't stored in a chunk of memory in the same way that an array is stored in C, so shared memory can't share Perl data as is. But there are modules that will translate data so that you can pretend your Perl data is in shared memory. See the perlfunc and the perlipc documentation.

Sockets

Perl has full support for TCP/IP sockets, which means you could write a web server in Perl, or a web browser, Usenet news server or client, finger daemon or client, FTP daemon or client, SMTP or POP or SOAP server or client, or either end of pretty much any other kind of protocol in use on the internet. You'll find low-level modules for these in the `Net::` name-space, and many of them come with Perl.

Of course, there's no need to get into the low-level details yourself; there are modules available for all of the common protocols. For example, you can make a web server or client with the `LWP`, `WWW::Mechanize`, or `Mojo::UserAgent` modules.

Security

Perl has a number of strong, security-related features that can make a program written in Perl more secure than the corresponding program written in C. Probably the most important of these is data-flow analysis, better known as *taint checking*. When this is enabled, Perl keeps track of which pieces of data seem to have come from the user or environment (and are therefore untrustworthy). Generally, if any such piece of so-called "tainted" data is used to affect another process, file, or directory, Perl will prohibit the operation and abort the program. It's not perfect, but it's a powerful way to prevent some security-related mistakes. There's more to the story; see the perlsec documentation or *Mastering Perl*.

Debugging

There's a very good debugger that comes with Perl and supports break-points, watchpoints, single-stepping, and generally everything you'd want in a command-line Perl debugger. It's actually written in Perl (so if there are bugs in the debugger, we're not sure how they get those out). But that means that, in addition to all the usual debugger commands, you can actually run Perl code from the debugger—calling your subroutines, changing variables, even redefining subroutines—while your program is run-

ning. See the perldebug documentation for the latest details. *Intermediate Perl* gives a detailed walkthrough of the debugger.

Another debugging tactic is to use the `B::Lint` module, which can warn you about potential problems that even the `-w` switch misses.

Command-Line Options

There are many different command-line options available in Perl; many let you write useful programs directly from the command line. See the perlrun documentation.

Built-in Variables

Perl has dozens of built-in variables (like `@ARGV` and `$0`), which provide useful information or control the operation of Perl itself. See the perlvar documentation.

References

Perl's references are similar to C's pointers, but in operation, they're more like what you have in Pascal or Ada. A reference "points" to a memory location, but because there's no pointer arithmetic or direct memory allocation and deallocation, you can be sure that any reference you have is a valid one. References allow object-oriented programming and complex data structures, among other nifty tricks. See the perlreftut and perlref documentation. *Intermediate Perl* covers references in great detail.

Complex Data Structures

References allow you to make complex data structures in Perl. For example, suppose you want a two-dimensional array. You can do that, or you can do something much more interesting, like have an array of hashes, a hash of hashes, or a hash of arrays of hashes. See the perldsc (data-struc-

tures cookbook) and perllol (lists of lists) documentation. Again, *Intermediate Perl* covers this quite thoroughly, including techniques for complex data manipulation, like sorting and summarizing.

Object-Oriented Programming

Yes, Perl has objects; it's buzzword compatible with all of those other languages. Object-oriented (OO) programming lets you create your own user-defined datatypes with associated abilities, using inheritance, overriding, and dynamic method lookup. Unlike some object-oriented languages, though, Perl doesn't force you to use objects.

If your program is going to be larger than N lines of code, it may be more efficient (if a tiny bit slower at runtime) for the programmer to make it object oriented. No one knows the precise value of N, but we estimate it's around a few thousand or so. See the perlobj and perlootut documentations for a start, and Damian Conway's excellent *Object Oriented Perl* (Manning Press) for more advanced information. *Intermediate Perl* covers objects thoroughly as well.

As we write this, the `Moose` meta-object system is very popular in Perl. It sits atop the bare-metal Perl objects to provide a much nicer interface.

Anonymous Subroutines and Closures

Odd as it may sound at first, it can be useful to have a subroutine without a name. Such subroutines can be passed as parameters to other subroutines, or they can be accessed via arrays or hashes to make jump tables. Closures are a powerful concept that comes to Perl from the world of Lisp. A closure is (roughly speaking) an anonymous subroutine with its own private data. Again, we cover these in *Intermediate Perl* and in *Mastering Perl*.

Tied Variables

A tied variable may be accessed like any other, but uses your own code behind the scenes. So you could make a scalar that is really stored on a remote machine, or an array that always stays sorted. See the perltie documentation or *Mastering Perl*.

Operator Overloading

You can redefine operators like addition, concatenation, comparison, or even the implicit string-to-number conversion with the `overload` module. This is how a module implementing complex numbers (for example) can let you multiply a complex number by `8` to get a complex number as a result.

Using Other Languages Inside Perl

Through the `Inline` modules, you can embed C and other languages inside a Perl program. The module takes care of connecting the external language to your Perl program in a seamless way that you won't notice. This is especially handy when a vendor provides a library in another language but you want to use Perl.

Embedding

The reverse of dynamic loading (in a sense) is embedding.

Suppose you want to make a really cool word processor, and you start writing it in (say) C++. Now you decide you want your users to be able to use Perl's regular expressions for an extra-powerful search-and-replace feature, so you embed Perl into your program. Then you realize that you could open up some of the power of Perl to your users. A power user could write a subroutine in Perl that could become a menu item in your program. Users can customize the operation of your word processor by writing a little Perl. Now you open up a little space on your website where users can share and exchange these Perl snippets, and you've got

thousands of new programmers extending what your program can do at no extra cost to your company. And how much do you have to pay Larry for all this? Nothing—see the licenses that came with Perl. Larry is a really nice guy. You should at least send him a thank-you note.

Although we don't know of such a word processor, some folks have already used this technique to make other powerful programs. One such example is Apache's `mod_perl` , which embeds Perl into an already-powerful web server. If you're thinking about embedding Perl, you should check out `mod_perl` ; since it's all open source, you can see how it works.

Converting find Command Lines to Perl

A common task for a system administrator is to recursively search the directory tree for certain items. On Unix, this is typically done with the *find* command. We can do that directly from Perl too.

The *find2perl* command, which comes with Perl up to v5.20 (and in `App::find2perl` now), takes the same arguments that *find* does. Instead of finding the requested items, however, the output of *find2perl* is a Perl program that finds them. Since it's a program, you can edit it for your own needs.

One useful argument that's available in *find2perl* but not in the standard *find* is the `-eval` option. This says that what follows it is actual Perl code that should be run each time that a file is found. When it's run, the current directory will be the directory in which some item is found, and `$_` will contain the item's name.

Here's an example of how you might use *find2perl*. Suppose that you're a system administrator on a Unix machine, and you want to find and remove all the old files in the */tmp* directory. Here's the command that writes the program to do that:

```
$ find2perl /tmp -atime +14 -eval unlink >Perl-program
```

That command says to search in */tmp* (and recursively in subdirectories) for items whose `atime` (last access time) is at least 14 days ago. For each item, the program should run the Perl code `unlink`, which will use `$_` by default as the name of a file to remove. The output (redirected to go into the file *Perl-program*) is the program that does all of this. Now you merely need to arrange for it to be run as needed.

Command-Line Options in Your Programs

If you'd like to make programs that take command-line options (like Perl's own `-w` for warnings, for example), there are modules that let you do this in a standard way. See the documentation for the `Getopt::Long` and `Getopt::Std` modules.

Embedded Documentation

Perl's own documentation is written in *pod* (plain-old documentation) format. You can embed this documentation in your own programs, and it can then be translated to text, HTML, or many other formats as needed. See the perlpod documentation. *Intermediate Perl* covers this too.

More Ways to Open Filehandles

There are other modes to use in opening a filehandle; see the perlopentut documentation. The `open` built-in is so feature-full that it gets its own documentation page.

Graphical User Interfaces (GUIs)

There are several GUI toolkits with Perl interfaces. See CPAN for `Tk`, `Wx`, and others.

And More...

If you check out the module list on CPAN, you'll find modules for even more purposes, from generating graphs and other images to downloading email, from figuring the amortization of a loan to figuring the time of sunset. New modules are added all the time, so Perl is even more powerful today than it was when we wrote this book. We can't keep up with it all, so we'll stop here.