INTRODUCTION:

Perl is short for **Practical Extraction and Report Language**. It's a language that is available free over the Web, and it's used for a variety of things, from writing CGI scripts to assisting administrators in maintaining their systems. Perl was created, and is still maintained, by Larry Wall. It's slower than C, but faster than a normal interpreted language. Instead, it's compiled when executed and then interpreted. A Perl compiler does exist but it's still under development. For more information on compiled Perl go to the Perl Home Page.

As I mentioned earlier, one of the nice things about Perl is that it's free. It's distributed under the GNU license, and the source is available from the Perl Home Page. This, along with the fact that it's flexible and has very few constraints, helps make Perl a popular language. It's also highly platform-independent, and has been ported to many different platforms including Unix, Windows, and even DOS. Code modifications can be minor or major depending on any system specific functions that are used.

Perl and CGI

This is a brief explanation of Perl and CGI. If you have, or want to, put up your own web site, you'll have heard of HTML, Javascript, Java, and CGI. CGI stands for the Common Gateway Interface. CGI is unlike Javascript, HTML, and Client Side Java. CGI provides a standard method for programs written in any language to run on the server side, and to communicate with the Web server software in response to requests for Web pages. In other words, CGI is the part of your web site that communicates with the other programs running on your server.

The main purpose of writing CGI programs or scripts is to afford more interactivity on your web site. Many web sites are static, and don't allow much user interaction. Others have guestbooks, image counters, and allow you to place orders, access databases, and other useful applications.

This is where CGI comes in. In order to implement these types of interactivity, you must have an application running on the server that, for example, knows how to process certain types of user input. As I mentioned, CGI scripts can be written in any language, but the most common are Perl and C. Perl is the preferred language for writing CGI scripts because of its strength in string manipulation. And now other programming languages such as Java and PHP are also gaining popularity, as they provide interactivity without the use of CGI.

CGI programming is nothing more than programming with some special types of input, and a few very strict rules for program output. When a user fills out a FORM (a collection of HTML tags that allow them to submit input) the server sends the form input to the specified CGI program, which in turn parses the entered data and uses it in a specified manner, and returns HTML with an answer to the user's request.

When you're choosing a language in which to write CGI scripts, ensure that the language you select:

- **Makes text manipulation easy**
- **Is able to interface with other software utilities and libraries**
- **Is able to access your operating system's environment variables**

Perl satisfies all the above requirements, making it a very good language for CGI programming. If you want to write your own CGI scripts, I'd suggest that you use Perl -- you'll find that it will make your life a lot easier. If you want to get more detail on CGI and how it works, try the CGI specification on the NCSA site (be aware, this is a pretty technical description aimed at experienced programmers).

Backgroung of Perl

In this appendix, we introduce a brief historical, sociological and psychological reasons for why Perl works the way it does. We also include a brief history of Perl.

When learning a new language, it is often helpful to learn the history, motivations, and origins of that language. In natural languages such as English, this helps us understand the culture and heritage of the language. Such understanding leads to insight into the minds of those who speak the language. This newly found insight, obtained through learning culture and heritage, assists us in learning the new language.

This philosophy of language instruction can often be applied to programming languages as well. Although programming languages grow from a logical or mathematical basis, they are rarely purely mathematical. Often, the people who design, implement and use the language influence the language, based on their own backgrounds. Because of the influence the community has upon programming languages, it is useful, before learning a programming language, to understand its history, motivations, and culture. To that end, this chapter examines the history, culture, and heritage of the Perl language.

Brief History of Perl

Larry Wall, the creator of Perl, first posted Perl to the `comp.sources` Usenet newsgroup in late 1987. Larry had created Perl as a text processing language for Unix-like operating systems. Before Perl, almost all text processing on Unix-like systems was done with a conglomeration of tools that included AWK, `sed`, the various shell programming languages, and C programs. Larry wanted to fill the void between "manipulexity" (the ability of languages like C to "get into the innards of things") and "whipuptitude" (the property of programming languages like AWK or `sh` that allows programmers to quickly write useful programs).

Thus, Perl, the Practical Extraction and Report Language, was born. Perl filled a niche that no other tool before that date had. For this reason, users flocked to Perl.

Over the next four years or so, Perl began to evolve. By 1992, Perl version 4 had become very stable and was a "standard" Unix programming language. However, Perl was beginning to show its limitations. Various aspects of the language were confusing at best, and problematic at worst. Perl worked well for writing small programs, but writing large software applications in Perl was unwieldy.

The designers of the Perl language, now a group, but still under Larry's guidance, took a look around at the other languages that people were using. They seemed to ask themselves: "Why are people choosing other languages over Perl?" The outcome of this self-inspection was Perl, version 5.

The first release of version 5 came in late 1994. Many believed that version 5 made Perl "complete". Gone were the impediments and much of the confusion that were prevalent in version 4. With version 5, Perl was truly a viable, general purpose programming language and no longer just a convenient tool for system administrators.

Perl as a Natural Language

Natural languages, languages (such as English) that people use on a daily basis to communicate with each other, are rich and complete. Most natural languages allow the speaker to express themselves succinctly and clearly. However, most natural languages are also full of arcane constructs that carry over from the language's past. In addition, for a given natural language, it is impossible to fully master the vocabulary and grammar because they are very large, extremely complex, and always changing.

You may wonder what these facts about natural languages have to do with a programming language like Perl. Surprising to most newcomers to Perl, the parallels between Perl and a natural language like English are striking. Larry Wall, the father of Perl, has extensive scholastic training as a linguist. Larry applied his linguistic knowledge to the creation of Perl, and thus, to the new student of Perl, a digression into these language parallels will give the student insight into the fundamentals of Perl.

Natural languages have the magnificent ability to provide a clear communication system for people of all skill levels and backgrounds. The same natural language can allow a linguistic neophyte (like a three-year-old child) to communicate herself nearly completely to others, while having only a minimal vocabulary. The same language also provides enough flexibility and clarity for the greatest of philosophers to write their works.

Perl is very much the same. Small Perl programs are easy to write and can perform many tasks easily. Even the newest student of Perl can write useful Perl programs. However, Perl is a rich language with many features. This allows the creation of simple programs that use a "limited" Perl vocabulary, and the creation of large, complicated programs that seem to work magic.

When studying Perl, it is helpful to keep the "richness" of Perl in mind. Newcomers find Perl frustrating because subtle changes in syntax can produce deep changes in semantics. It can even be helpful to think of Perl as another natural language rather than another programming language. Like in a natural language, you should feel comfortable writing Perl programs that use only the parts of Perl you know. However, you should be prepared to have a reference manual in hand when you are reading code written by someone else.

The fact that one cannot read someone else's code without a manual handy and the general "natural language" nature of Perl have been frequently criticized. These arguments are well taken, and Perl's rich syntax and semantics can be confusing to the newcomer. However, once the initial "information overload" subsides, most programmers find Perl exciting and challenging. Discovering new ways to get things done in Perl can be both fun and challenging! Hopefully, you will find this to be the case as well.

Lets the Game Begin

In this tutorial , a variety of conventions are used to explain the material. Certain typographical and display elements are used for didactic purposes.

Any Perl code that is included directly in flowing text appears like this: `$x = 5`. Any operating system commands discussed in flowing text appear like this: `program`. Operating system files that are discussed directly in the flowing text appear like this: `file`. When a technical term of particular importance is first introduced and explained, it appears in emphasized text, like this: *an important term*.

When Perl code examples or operating system commands need to be separated away from the flowing text for emphasis, or because the code is long, it appears like this:

```
my $x = "foo";      # This is a Perl assignment
print $x, "\n";     # Print out "foo" and newline
```

All Perl code shown in this manner will be valid in Perl, version `5.6.0`. In most cases, you can paste code from one of these sections into a Perl program, and the code should work, even under `use strict` and `use warnings`.

Sometimes, it will be necessary to include code that is not valid Perl. In this case, a comment will appear right after the invalid statement indicating that it is not valid, like this:

```
$x = "foo;     # INVALID: a " character is missing
```

When code that we set aside forms an entire Perl program that is self-contained, and not simply a long example code section, it will appear like this:

```
#!/usr/bin/perl
use warnings;
use strict;
print "Hello World\n";
```

Finally, when text is given as possible output that might be given as error messages when `perl` is run, they will appear like this:

```
Semicolon seems to be missing
syntax error
```

Keep these standards in mind as you read this tutorial.

Basics of Perl

First code of line, And Other Details

Grossly oversimplifying: A Perl program belongs in a file (which you have made executable with some command like

```
  chmod +x filename
) in your bin directory, where the first line of the file is
  #!/usr/bin/perl
```

Everything after that first line is Perl, except for lines beginning with #, which are comments. Every command in Perl should end in a semi-colon; missing semi-colons account for 90% of novice user errors, and 80% of expert user errors. So Perl programs look like this:

```
  #!/usr/bin/perl
  command;
  command;
  # this is a comment, you can say whatever you want here
  command;
```

Once you've written your Perl program, you run it by typing the name of the file. If you've made some mistake in your "code" (that's geek for "program," as if "program" weren't geeky enough) Perl will refuse to run it, spitting out some usually-informative error message instead.

Enough abstraction; on to some examples.

The Print Statement

The most basic thing you're going to want a perl script to do is tell you things that is, usually, send output to the terminal. Here's a fully functioning one-line Perl script:

```
  #!/usr/bin/perl
  print "1 + 2 = 3\n";
```

Run it, and here's what it prints on your screen:

```
  1 + 2 = 3
```

The \n is an end-of-line statement; you'll spend a lot of time in Perl sticking those in and taking them out, unless you want all of your output to come all mushed together on one line and generally misbehaving.

Variables

You can store data in "variables", so that you can tinker with them and use them at will. Here's the above program using variables:

```
#!/usr/bin/perl
$x = 1;
$y = 2;
$z = 3;
print "$x + $y = $z\n";
```

Any variables in quotes will be interpolated--that is, they'll be translated into the data they contain. So when you run that program, you still get

```
1 + 2 = 3
```

Assigning to Variables

The print statement above cheated, though; it didn't really add the numbers. You can add $x and $y with a line like this:

```
$z = $x + $y;
```

That line really performs the math, and stores the result in $z. So a program like this

```
#!/usr/bin/perl
$x = 1;
$y = 2;
$z = $x + $y;

print "$x + $y = $z\n";
```

will once again print out

```
1 + 2 = 3
```

Getting Input From the User

Variables give your program flexibility. You can now change that program so it adds any two numbers you type in. The way to get input from the keyboard into a running program is with a line like this:

```
$x = <>;
```

When your program encounters a line like that, it will stop and wait for numbers (or letters, or anything else) to be typed in at the keyboard, until RETURN is hit. Then it will assign everything you just typed into $x, and the program can then play with it. So this program

```
#!/usr/bin/perl
print "Type in a number: ";
$x = <>;
chop($x);
print "Type in a number: ";
$y = <>;
chop($y);
$z = $x + $y;
print "$x + $y = $z\n";
```

will wait for two numbers, add them up, and then print the result.

The one strange thing in that program is the *chop* command. When the value of $x comes in from the keyboard, it's stored *along with the* \n *character from hitting RETURN*. So if you typed 35 and then hit RETURN, the value of $x became

```
35\n
```

Chop just chops the last character off of a variable. Its most common use is for getting rid of that nasty "\n"--which, believe it or not, counts as one character. (Even more to the point: it counts as a charcter, which means "35\n" is not anything like "35". You can't add with it, you can't print it without getting an unwanted line break, and it won't do anything else you might expect. Chopping is good Perl practice.)

Subroutines

See how, in the above program, the first six lines basically do the same thing twice? Computer programs are designed to perform repetitive tasks for you; if you find yourself typing the same thing over and over again in a program, it's a good clue that you should condense your program with a *subroutine*.

A subroutine is like a command you define yourself. So let's make a version of the program that uses a subroutine called *getnumber* that gets a number from the keyboard:

```
#!/usr/bin/perl
$x = &getnumber;
$y = &getnumber;
$z = $x + $y;
print "$x + $y = $z\n";
# The program ends there...
# the subroutine is just tacked on at the end:
sub getnumber {
    print "Type in a number: ";
    $number = <>;
    chop($number);
    $number;
}
```

Everything inside the curly brackets is the subroutine. Now that section of the program will be run every time you use the command

```
&getnumber;
```

in your program. In this case, the subroutine "getnumber" *returns a variable* which is then assigned to $x and $y. Not all subroutines are designed to return variables. This one returns the contents of $number because of the line

```
$number;
```

So the command

```
$x = &getnumber;
```

first runs the subroutine, and then, when it sees the line *$number;* it exits the subroutine, and gives $x the value of $number.

But like I said, not all subroutines are designed to return a value. For instance, you could have a subroutine that just printed out a standard warning:

```
sub warning {
   print "WARNING: The program is about to ask
         you to type in a number!\n";
}
#So the line
  &warning;
```

will now print out:

WARNING: The program is about to ask you to type in a number!

every time you use it.

You can stick the code for subroutines anywhere you want in your program, but the traditional place is to group them all together at the end of the program, after the main section.

While

Right now the program just executes once and then kicks you out. Many times you'll want your program to do something over and over again; you can do this with a *control structure* that creates a "loop". There are many types of control structures; we'll start you off with a simple one called while. It basically means: *do this set of commands while this statement is true.* For instance, here's a code fragment that loops over and over as long as you type in the number "15":

```
$x = 15;
while($x == 15) {
    $x = &getnumber;
}
```

Two things to note: 1) *$x == 15* means "$x is equal to 15". It's a comparative statement; it should never be confused with *$x = 15*, which *assigns* the value 15 to $x. Confusing == and = is one of the most common novice errors; in this case, it would create an infinite loop.

2) The first line of the code fragment sets the value of $x to 15. If you didn't do that, $x wouldn't equal 15 the first time around, so the program would never enter the loop at all, and would never ask for a number.

Here's our adding program, with a loop so you can use it over and over:

```
  #!/usr/bin/perl
  $doagain = "yes";
  while($doagain eq "yes") {
      $x = &getnumber;
```

```
        $y = &getnumber;
        $z = $x + $y;
        print "$x + $y = $z\n";
        print "Do it again? (yes or no) ";
        $doagain = <>;
        chop($doagain);
    }
```

(I left out the subroutine to save space here, but you'd have to include it if you wanted that program to run.) Note that the comparative statement this time uses "eq" rather than "==". == compares for numerical values; eq is for variables made up of letters, numbers, and other characters (called "string" variables.)

Pattern Matching and Regular Expressions

If you were paying attention, you noticed a huge loophole in the programs above: there's nothing to prevent you from typing in a string variable when you're supposed to be typing in a number. You can type in "dog" and "cat", and the program will try to add "dog" and "cat" (which, if you're curious, gives a result of zero.) You need some way to check to make sure that the person actually typed in numbers; then, if they didn't, you can ask them again (with a looping control structure), until they get it right.

Welcome to the concepts of pattern matching and regular expressions, two of Perl's powerful text-processing tools. Let's start with a simple pattern first: one letter. If you want to test a variable to see if it contains the (lower-case) letter "z", use this syntax:

```
     if ($x =~ /z/) {
          print "$x has a z in it!\n";
     }
```

Let's take that apart: *if* is just like while, except it only checks once (that is, it won't loop around again and again.) Like while, it will execute every command inside the curly brackets if the statement inside the parentheses is true.

The statement inside the parentheses works like this: =~ makes a comparison between $x and whatever's inbetween the two slashes; in this case, if there's a z anywhere inside $x, then the statement is true.

Let's up the ante, and match only if $x begins with the letter z:

```
    if ($x =~ /^z/) {
        print "$x begins with a z!\n";
    }
```

^z is a *regular expression*; the carat (^) stands for the beginning of the string. Thus, the matching statement has to find a z immediately following the beginning of the string in order to be true.

How about words that begin with z and end with e? Use the regexp

```
 /^z.*e$/
```

The $ stands for the end of the string; the period stands for "any character whatsoever"; combined with the asterisk, it means "zero or more characters." Without the asterisk,

```
/^z.e$/
```

would mean "z followed by one character followed by e."

There's a lot of different regular expressions. For instance,

```
/^z.+e$/
```

means "z followed by *at least one* character, followed by e."

```
/^z\w*e$/
```

means "z followed by zero or more *word characters* followed by e"--that is, "z!e" wouldn't match.

So to make sure that somebody's typing in numbers in our adding program, and not words, make the subroutine getnumber look like this:

```
sub getnumber {
    $number = "blah";
    while($number =~ /\D/){
       print "Enter a number ";
       $number = <>;
       chop($number);
    }
    $number;
}
```

"\D" is the regular expression for non-digits; if any character in $number is not 0-9, the expression won't match, and you'll get asked to enter a number again.

Note how we had to set $number to include a non-digit (`$number = "blah"`) to get inside the loop the first time around.

Substitution

You can transform variables at will or whim using regular expressions, by use of the substitution command. This command will change the letters "dog" to "cat" if they occur in the variable $x:

```
$x =~ s/dog/cat/;
```

Actually, that will only change the last occurence (so "dogdog" would become "dogcat".) To make the change "global", add a g at the end:

```
  $x =~ s/dog/cat/g;
```

This will change "dig" and "dog" (but not "doug") to "cat":

```
  $x =~ s/d.g/cat/g;
```

A Little About Arrays

A single number or string can be assigned to a scalar variable, in the form

```
  $x = 45;
```

If you have a bunch of variables and want to store them together, that's an array. Here's how you assign the numbers 3, 5, 7, and 9 into an array called @dog:

```
  @dog = (3, 5, 7, 9);
```

Although the entire array is referred to as @dog, an individual element--let's say, the first one--is $dog[0]. So $dog[0] equals 3, and $dog[1] equals 5. (Throughout history, programmers have begun their lists with the number zero, and throughout history, this has caused nothing but grief and turmoil.)

Note that the variable $dog and the array elements $dog[0]...$dog[3] are totally unrelated. $dog could be "zebra" or 87000, and it would never affect any of the elements of @dog.

That's just the barest hint of what arrays can do. I'm only bringing them up now so you won't freak out if I use them in an example.

if/then/elsif/else

You can use `if` all by itself, in lines like

```
   print "You're 28!\n" if ($age == 28);
```

That's really shorthand for

```
  if ($age == 28) {
      print "You're 28!\n";
  }
```

which means, "if $age equals 28, then do everything inside the curly brackets." The if/then control structure can be extended with `elsif` and `else`:

```
  if ($age == 28) {
      print "You're 28!\n";
  } elsif ($sex eq "f") {
      print "You're a female!\n";
  } elsif ($sex eq "m") {
      print "You're a male!\n";
  } else {
      print "You're a mystery to me...";
  }
```

Only the first true statement will match. So if ($age == 28) and ($sex eq "f"), you'll get the output

```
  You're 28!
```

`else` is the default; its commands will be executed only if none of the preceding statements in the control structure were true

The while/until Structures

The `while` structure is equivalent to the `while` structures in Java, C, or C++. The code executes while the expression remains true.

```
use strict;
while (expression) {
    While_Statement;
    While_Statement;
    While_Statement;
}
```

The `until (expression)` structure is functionally equivalent `while (! expression)`.

The do while/until Structures

The `do/while` structure works similar to the `while` structure, except that the code is executed at least once before the condition is checked.

```
use strict;
do {
    DoWhile_Statement;
    DoWhile_Statement;
    DoWhile_Statement;
} while (expression);
```

Again, using `until (expression)` is the same as using `while (! expression)`.

The for Structure

The `for` structure works similarly to the `for` structure found in C, C++ or Java. It is really syntactic sugar for the `while` statement.

Thus:

```
use strict;
for(Initial_Statement; expression; Increment_Statement) {
    For_Statement;
    For_Statement;
    For_Statement;
}
```

is equivalent to:

```
use strict;
Initial_Statement;
while (expression) {
    For_Statement;
    For_Statement;
    For_Statement;
    Increment_Statement;
}
```

The foreach Structure

The `foreach` control structure is the most interesting in this chapter. It is specifically designed for processing of Perl's native data types.

The `foreach` structure takes a scalar, a list and a block, and executes the block of code, setting the scalar to each value in the list, one at a time. Consider an example:

```
use strict;
my @collection = qw/hat shoes shirts shorts/;
foreach my $item (@collection) {
    print "$item\n";
}
```

This will print out each item in collection on a line by itself. Note that you are permitted to declare the scalar variable right with the `foreach`. When you do this, the variable lives only as long as the `foreach` does.

You will find `foreach` to be one of the most useful looping structures in Perl. Any time you need to do something to each element in the list, chances are, using a `foreach` is the best choice.

File Manipulation

Filehandles

You can write to a file as easily as you can write to the terminal. The first step (almost the only step) is to open the file with the open command. Here's an example:

```
   open(DOG,">/home/scotty/data/dogs");
```

DOG is the "filehandle"--the name by which you'll refer to the open file from now on. It's customary to use all caps for filehandles. The other thing inside the parens is the full pathname of the file; it's prefixed with a > so you can write to it. (Without the > you could only read from it--we'll talk about that in a second.)

Now to write a line of text to the file, just do it like this:

```
print DOG "This line goes into the
             file and not to the screen.\n";
```

Pretty easy, huh? I love Perl.

Since failure to successfully open a file can cause your program to go batty, it's a good idea to have the program exit gracefully if it fails to open the file. To do that, use this syntax for the open command:

```
open(DOG,">/home/scotty/data/dogs")
          || die "Couldn't open DOG.\n";
```

Now it will either open the file, or quit the program with an explanation why.

To read from a file, open it without the >:

```
open(DOG,"/home/scotty/data/dogs")
           || die "Couldn't open DOG.\n";
```

(If the file doesn't exist, the program will quit.) Once the file is open, there are two common ways to get the information out. You can do it one line at a time, with lines like this:

```
   $x = <DOG>;
```

That copies the first line from DOG (or the next line, if you've already taken some lines out) and assigns it to $x. The <DOG> syntax is just like the <> we used earlier to get input from the keyboard; sticking DOG in there just tells Perl to get the input from the open file instead.

You can also do it in a loop. This program prints out all the contents of DOG:

```
  #!/usr/bin/perl
  open(DOG,"/home/scotty/data/dogs") ||
                  die "Couldn't open DOG.\n";
  while(<DOG>) {
       print;
  }
```

Notice how we didn't specify a variable to store each line from DOG in, and we didn't specify anything for the print command to print? This is a really important concept I should have introduced earlier: Perl features a default variable called `$_`. Basically, if you don't specify which variable you want to use, or if you use a command like print by itself, Perl assumes you want to use $_.

Some other common commands that can assume you're talking about $_:

```
  s/dog/cat/g;
```

That's a valid line all by itself; it means "substitute all occurences of 'dog' in the variable $_ with 'cat'." Another popular type of construction is

```
  print if (/dog/);
```

That means "print $_ if $_ has 'dog' in it."

$_ shows up everywhere in Perl, just to make your life easier. For instance, foreach will store its elements in $_ if you're too lazy to name a variable yourself:

```
    foreach (@array) {
        print if (/Tonight/);
    }
```

You can assign from $_ or manipulate it just like any other variable, with commands like

```
   $_++;
   $x = $_;
```

You just have to do it before the next time you overwrite $_ with a command like

```
   <DOG>
```

--it's a very temporary storage space.

When you're done with a file, don't forget to close it with the close command:

```
   close(DOG);
```

Filename Globbing

Perl can read all the filenames in a directory (/home/scotty/bin in the following example) with this syntax:

```
    while($x = </home/scotty/bin/*>) {
        ...
    }
```

One obvious and powerful use of this "filename globbing" is a loop like this:

```
while($x = </home/scotty/bin/*>) {
    open(FILE,"$x") ||
        die "Couldn't open $x for reading.\n";
    ...
}
```

Thus, the following simple program will print all lines containing the word "dog" (along with the names of the files they came from) in the /home/scotty/bin directory:

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

```
while($x = </home/scotty/bin/*>) {
    open(FILE,"$x") || die "Couldn't open $x for reading.\n";
    while(<FILE>){
        if(/dog/) {
            print "$x: $_";
        }
    }
}
```

Opendir

Here's a key point of Perl philosophy: Perl tries to never limit you. Arrays can be as big as you want, strings as long as you want, and strings can contain anything. If you try to open a file, and the file doesn't open, the program hums right along; if a line is expecting three variables back from a subroutine, as in

```
  ($one, $two, $three) = & some_routine;
```

and it only gets back one, well, it won't care. It'll just pad the other variables and move along.

Anyways, Perl doesn't have limits, but Unix sometimes does. A key example is the difference between filename globbing, which relies on the Unix shell's built-in functions, and Perl's own `opendir` function.

Try to open a directory with 3000 files using filename globbing, and your program will probably crash. But you can open a directory of 100,000 files with `opendir` (as long as your machine can handle it...)

Anyways, here it is:

```
    opendir(DIR,"/tmp");
    while($file = readdir(DIR)){
          ....
    }
```

Note that opendir will try to open every file, including the enigmatic . and .. files. (I can't imagine why, but I'm sure there's a reason.) So here's a variation that will only attempt to open files whose names don't begin with dots:

```
    opendir(DIR,"/tmp");
    while($file = readdir(DIR)){
        next if ($file =~ /^\./);
          ....
    }
```

Using Unix from Within Perl

Writing to a Process

Watch this: this is so cool:

```
 open(MAIL,"|/usr/lib/sendmail scotty@bluemarble.net");
 print MAIL "Subject: This is a subject line\n\n";
 print MAIL "This is an empty body.\n";
 close(MAIL);
```

What did that do? It wrote to a process rather than a file. That little pipe (|) on the first line tells any output to the filehandle MAIL to go to a sendmail process; when you close up the process, with the close statement--presto, you've sent yourself a mail message.

This is easily one of Perl's most powerful features. For instance, if you have a Perl script that generates a list of old files (at work, say) and the Internet addresses of the co-workers who own the files, you can send a message to every single one of them:

```
while(....){
    ....
    open(MAIL,"|/usr/lib/sendmail $owner");
    print MAIL "Subject: Notification of Ancient Fileness\n\n";
    print MAIL "You own an ancient file named $ancientfile.\n";
    print MAIL "Would you like me to do anything about it?\n";
    close(MAIL);
}
```

Yup: variables.

System Calls

A lot of the time, Unix can do something faster or easier than your Perl scripts can. Or maybe you just don't want to rewrite an entire program that Unix already provides... Perl enables you to execute a shell command from within a Perl program via the system() function: for instance, to allow a user to edit a file directly:

```
    system("/usr/bin/emacs $file")||die;
```

If you don't know what Emacs is, or aren't that familiar with Unix programs, then just forget you ever saw this section. Mostly it would just lead you to do extremely lazy shortcut things anyways.

Backticks

You can also fire up a shell process with the so-called 'backticks':

```
@lines=`/usr/local/bin/lynx -source http://www.website.com`;
```

Now you have the front page of some website in an array, one line neatly stored in each array element. Imagine what you can do with that...

```
@lines=`/usr/local/bin/lynx -source http://www.website.com`;
foreach (@lines) {
    print if (/href/i);
}
```

(And yes, you can write perfectly wonderful spiders, robots and web catalogs with Perl... Just, please, BE RESPONSIBLE. Read up on robot ethics before you even think about it.)

One important difference between a `system()` call and backticks: if you fire up a program with a line like this

```
    `/home/scotty/bin/some_program`;
```

then your original program won't wait until `some_program` is finished before hastening on to the next line. This can be quite powerful, but it's usually unnecessary and kinda scary until you get the hang of it. So to make sure that your original program waits obediently until `some_program` is finished up, do this:

```
    $result = `/home/scotty/bin/some_program`;
```

$result is a throaway value here; its only purpose is to force the original program to wait for `some_program` to exit.

SUMMARY

Strengths

- **Cross-platform scripting language**
- **Easy to write powerful programs with few lines of code**
- **Great language for writing throw-away programs**
- **Good for data parsing/extraction/manipulation tasks**
- **Flexible Language**

Weaknesses

- **Sometimes difficult to maintain large Perl programs**
- **At times, too flexible of a language**
- **Can be made very cryptic, very fast**

Now that you've gotten free know-how on this topic, try to grow your skills even faster with online video training. Then finally, put these skills to the test and make a name for yourself by offering these skills to others by becoming a freelancer. There are literally 2000+ new projects that are posted every single freakin' day, no lie!