

**FOUNDATIONS IN PERL
PROGRAMMING – For
Experienced Programmers**



© September 2011 by
ERIC MATTHEWS

Copyright Notice

Foundations in Perl Programming by Eric Matthews is licensed under a Creative Commons Attribution 3.0 Unported License. This license allows you to

- Copy, distribute and transmit the work
- Adapt the work
- Make commercial use of the work

Under the following conditions:

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work

With the understanding that:

Waiver — any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain — where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights — in no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — for any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to one or more of the following web pages.

www.DeveloperGeekResources.com

www.EducationAnytime.com

Contents

Copyright Notice	2
<i>Preface</i>	8
About these Materials.....	8
A brief introduction to Perl.....	9
Assumptions.....	10
<i>Getting Perl</i>	11
<i>The Bare Minimum</i>	12
Get this one out of the way.....	12
Comments.....	12
Running a Perl program.....	12
Print function.....	13
Getting input from the terminal.....	14
Scalars (what you probably know as a variable).....	14
Operators.....	16
<i>Arithmetic</i>	16
<i>Logical</i>	16
<i>Assignment</i>	16
<i>Backslash</i>	17
<i>Match</i>	17
<i>String Comparison</i>	17
<i>Integer Comparison</i>	18
Parentheses.....	18
Conditional Logic.....	18
IF	18
IF/ELSE	19
IF/ELSIF	19
NESTED IF	20
Loops.....	21
For	21
While	21
Iteration Operators.....	21
File IO.....	22
Reading files	22
Using either the standard error handling, and/or return your own error text	24

Enter the file you want to open at program runtime	24
Copying the contents of a file to a new file, or recreate an existing file	24
Appending a file with the contents of another file	25
Other techniques	25
Using "die" in file IO	25
Arrays	25
Determining the length of an array	26
Accessing an array item by subscript	27
Finding an array element	27
Assigning an array to an array	27
Sorting an array	27
Pattern Matching	27
The =~ and !~ operators	28
The + and * Character	28
[] (Character class)	29
Match any letter of the alphabet, case insensitive	30
Match any number	30
Match a special character	30
Using SPLIT	30
Subroutines	31
Getting input from the terminal	32
CHOP and CHOMP	32
CHOMP	33
<i>More on Scalars</i>	35
Legal names.....	35
Assignment	35
Assigning a character or string to a scalar	36
Assigning a scalar to a scalar	36
Assigning two scalars to a scalar using the concatenation operator	36
Assigning a number to a scalar	36
Assigning the results of an expression to a scalar	36
Indirection	36
Misc. Scalar Stuff	36
UNDEF/DEFINED or Perl's equivalent to NULL	36
Testing the state of a scalar	37
Converting a character to ASCII and Binary	38
Octal/Hex/Decimal	38
Rounding Numbers	38
Scope/Persistence	38
<i>More on Arrays</i>	41
Pushing an array	41
Popping an array.....	41
Reversing an array	41
Shifting and Unshifting	42

Shift.....	42
Unshift	42
<i>More on Subroutines</i>	<i>44</i>
Subroutine declaration.....	44
Calling subroutines.....	44
Passing parameters	45
Using a variable number of arguments.....	46
Variable scope in a subroutine	46
Static variables in subroutines	48
Returning values.....	48
Scalars.....	48
Arrays.....	50
Hashes.....	50
<i>Regular Expressions.....</i>	<i>51</i>
So how does a regex match?	51
Framework to match within.....	52
Quantifiers	53
Character class.....	54
Basic use of Parenthesis and Alternation	55
Putting some of what we have covered together	55
The dot operator	58
Being able to read and articulate a regular expression.....	59
Character and Non Character anchors.....	59
Word and Non Word anchors	60
Position at start or at end of line	60
Negated character class.....	61
Greed	61
A practical application of greed.....	62
Gimme back my bullet (with apologies to Lynyrd Skynyrd)	63
Just when you were starting to feel good (when greed follows greed).....	64
A wee bit more theory that you could apply (Backtracking).....	65
Making quantifiers less greedy.....	67
More Goodies	68
Does not match operator.....	68
Substitution operator	68
Making your regex case insensitive.....	68
Compiling your regex.....	69
Some Regexercises (pun intended).....	69
<i>Hashes</i>	<i>74</i>
Creating hashes.....	75
Adding an element to a hash	75

Deleting a hash element.....76

Get hash keys.....76

Get hash values.....77

Get both hash keys and values.....77

Finding a particular hash key77

Reversing values and keys in a hash78

Sorting a hash79

By Key79

By Value79

Merging two hashes.....80

Basic Debugging81

 Debugging81

 Starting the debugger.....81

 List source code.....81

 Stepping.....82

Execute next statement.....82

 Listing lines of code82

Display next window of code82

Display n lines from line#.....82

Display lines n through n83

Display a specific line#83

Display lines around the current line.....83

 Variables.....84

Display variables.....84

Modify variables84

 Commands85

Display last n commands.....85

Recall a command.....85

 Breakpoints.....85

On line#85

On Subroutine.....85

 Resume86

 Return from a Subroutine86

 Delete all breakpoints86

 Writing code while you are debugging.....86

 Exit debugger86

 Tracing a call87

 Watching a variable throughout program execution87

Misc Stuff.....88

Entering values in hex88

Bit manipulation88

Random Numbers88

Truth Tables88

Time.....88

Appendix A – Recommended Reading and Sites89

 Reading.....89

Perl Black Book by Steven Holzner89

Mastering Regular Expressions by Jeffrey E. Freidl89

Programming Perl by Larry Wall, Tom Christiansen, Jon Orwant90

 Websites90

Appendix B – Useful Reference Information91

 Operators91

Arithmetic91

Logical91

Assignment.....91

Backslash.....92

Match.....92

String Comparison92

Integer Comparison.....93

 Regular Expressions93

Greedy Quantifiers.....93

Non-Greedy Quantifiers93

Pattern Matching.....93

Atoms.....94

Special Characters.....94

Assertions (Anchors)95

Modifiers for m// and s//95

 Debugging96

PREFACE

ABOUT THESE MATERIALS

These materials were originally written in 1998. It is very cool to think over a decade later that they are still relevant today. Perl is a great language. While it is rare to find a job in the marketplace for Perl only, having Perl as a second (or third) language that you are proficient in is a big plus to a prospective employer. While Perl can be used in many venues I use Perl to develop command line tools. I also reach for Perl when I need to write a quick hack to solve a problem. Perl has many uses across the software development spectrum.

The point of these materials are to...

[Get YOU programming in Perl as quickly as possible.](#)

I have tried to cut the narrative down to the bare minimum. The real story here is to present you with syntax and a number of small working examples in code for you to examine and get ideas from. Again, this work assumes that you are an experienced programmer. If you are not, then you may not find this of much help.

I also did not make the assumption that everyone reading this was Einstein. I certainly am not. So many books on the market are either geared toward the novice programmer or the rocket scientist programmer.

For me a good book on programming

- Assumes I am a programmer, but does not assume that I enjoy reading pages upon pages of someone else's code.
- Contains lots of small, working code examples to clarify and exemplify the author's point about a key concept pertaining to the language.
- Provides me with the author's point of view regarding the discipline of software engineering so I can gain a perspective other than my own.
- Provides me with some short practical examples of how to best use the language.
- Is not written like my daughter's Social Studies book (Note: my daughter is now a school teacher, how time flies), so dry and empty that I can barely read a paragraph before becoming comatose.

I have tried to keep the preaching down to a minimum. Having been in this business for the past 14 years I have come to realize how many passionate people we have in the discipline of software engineering (myself being one of them). Put a group of programmers in a room and try to get them to agree on the practical implementation of subjects like error handling and reuse and you generally end up with the equivalent of a holy war. I think that there is a fine line between shoving a point of view down someone else's throat and offering your perspective on a matter that you feel very passionate about. Nothing hacks me off more than a programmer that thinks his/her way is the only way. My perspective is nothing more than my perspective. It is formed (and is continually being reformed) based on what works for me, and what I see working for others.

A “Perl” claim to fame is the freedom it allows you as the programmer to solve a problem. On the cover of the acclaimed book, Programming Perl are the words “There’s more than one way to do it”. In fact, the inventor of Perl, Larry Wall has these two very important things to say.

“How does Perl put the focus onto the creativity of the programmer? Very simple. Perl is humble. It doesn’t try to tell the programmer how to program. It lets the programmer decide what rules today, and what sucks. It doesn’t have any theoretical axes to grind. And where it has theoretical axes, it doesn’t grind them. Perl doesn’t have any agenda at all, other than to be maximally useful to the maximal number of people.”

“True greatness is measured by how much freedom you give to others, not by how much you can coerce others to do what you want.”

Throughout this book I will do my best try to offer you different approaches to logic. But, you must understand at the outset (I believe most of you already do) that I am going to be naturally biased to showing you toy code in the way I think it should be written. You should and must be free to think and solve problems in the way that works best for you. I am not at all advocating writing sloppy code, or code that is so obfuscated that only you can read and understand it. I am saying that brother Larry got it right by realizing that logic is not something one can or should dictate.

In reading this please feel free to disagree with me. I will offer my perspective at times to provoke thought and offer a point of view that you may have never considered. In the end you must work within the limits and boundaries of your own imagination and thought processes.

Important note about the chapters I have written

These chapters were put together over a very long period of time. They were also put together the moment I started learning the language. You will find that I have used many different ways to do the same thing. This was not done on purpose, rather as an evolution to my learning the language. I really like the attitude of “there is more than one way of doing it”! Most languages immediately force you to learn and practice their dogma. Most languages are quite blatant in the axes that they grind. The thing I admire about Perl is that you can come in and program in the manner that you are already accustomed to and be quite productive. Over time you will hopefully learn new ways of doing things and even new ways of thinking.

I debated high and low about going back and totally rewriting the earlier modules. But I came to the realization that this would not be the best approach. In some of the early modules I have gone back and added new things that I have learned, where I felt their introduction to be appropriate. Also, I believe in hindsight that some of my initial approaches and struggles with the language are similar to yours. This perspective is inherent in my level of experience at the time it was written, and one I could duplicate today even if I wanted to.

A BRIEF INTRODUCTION TO PERL

Perl is a strange bird. It looks on the surface like an interpreted language. Yet under the hood Perl is really a compiler that thinks it is an interpreter. I am generally not a cheerleader for any particular language, yet I find that the more and more I use Perl the more “into” the language I get. For me this is a language that I can see using for many years to come (and here it is 2011, fourteen years after I penned those words). In fact, this language will probably be around in some incantation as long as text files exist.

Perl (Practical Extraction Reporting Language) is a general purpose language that can be used for a variety of applications including...

- Developing programs and tools for maintaining and administering operating systems
- Creating dynamic HTML pages (Authors note: I have moved on to Php, which I find to be a stronger language for web development than Perl and CGI)
- Creating server side WEB scripts
- Writing complex parsers that read, interpret, and modify text files
- And many more applications.

Additionally,

- Perl can be obtained for the right price. It is free.
- Perl has been ported to most major platforms (Unix, AIX, NT, Linux, even H-P)
- Perl programs run fast.
- Perl is a fast development tool.

What a great language that allows you to make a change to your source file, switch to another window and run the program to see your changes without having to close your source file

While Perl syntax looks C like the similarities soon end. Also, some programmers seem to go out of their way to write Perl code that is unreadable. There is even an annual event to see who can write the most unreadable code. This certainly detracts from people wanting to learn the language. Do not let this deter you. You can write readable, optimized code in Perl.

Perl is in my opinion the best language I have used for working with text files.

Perl is not a hard language to learn. The first week I starting learning the language I wrote a very useful script. I have never been able to apply another language in so short a time to do some useful work.

The sample code presented was originally written in Windows NT4. Today I run it can be run on any version of windows. You should also have no problem applying these materials to any platform that Perl is on. If you are using this as part of a class you are taking, keep in mind that the teacher may not be able to help you beyond Perl if you choose to use a different platform other than what is being used to teach on.

ASSUMPTIONS

As the title implies, I am assuming that you are already an experienced programmer. No attempt will be made to define basic programming terminology. I will attempt to bridge terminology and concepts of software engineering where I feel it is applicable.

Have fun!

GETTING PERL

I tend to like the ActiveState version of Perl. Frankly it does not matter which build you get.

Here is a url to get your hands on Perl.

<http://www.perl.org/get.html>

Installing Perl is pretty easy.

THE BARE MINIMUM

This section is all about providing the bare minimum of Perl syntax to get you started. In later sections we will get into some of the finer points of these topics, and we will be using these topics in a more applied manner. You may also find some of this information repeated in later sections.

You should be able to put all of this information together and begin writing some small Perl programs to get a feel for the language. I would recommend you run each script then spend some time analyzing the code.

GET THIS ONE OUT OF THE WAY

Time to get tradition out of the way.

PROGRAM: hello.pl

```
print "Hello World";
```

This single line program besides being lame does introduce a few introductory items in perl. Note that just like in C/C++ , Perl statements are terminated with a (;) semicolon. It also introduces us to the print function and to literals.

COMMENTS

denotes the beginning of a comment. A comment can start a line or be placed at the end.

EXAMPLES

```
# this is a comment
```

```
$x = (100 * 7); # a comment can be placed after code
```

As Perl code can get cryptic it is a good idea to add comment's as you may not have a clue what the code does weeks after you write it.

RUNNING A PERL PROGRAM

You run a Perl program from the directory where the executable perl.exe lives. Unless you specify otherwise this will live in the directory perl5. Please note that my examples assume you are using Perl in some version of Windows.

The syntax for running a Perl program is:

```
perl <path to Perl program>
```

for example:

```
perl g:\myperl\hello.pl
```

Perl programs in Windows will have the extension '.pl', though Perl could really care less what the extension is or if there is an extension at all. The only requirement Perl has is that the file is a text file that contains valid Perl syntax.

It is a good idea to use the file extension '.pl' as a standard for being able to easily spotting your Perl programs.

You may also want to set up your path so that you can run Perl from anywhere. Since this material is about Perl I will let you figure that one out on your own.

In writing this manual I struggled with when and how to introduce certain aspects of the language that come in handy but whose use is not required. One of these is the `-w` argument when running your scripts. This argument is benign if your script is warning free. It is possible to write a Perl script that does exactly what you want but contains warnings. An “age old” question in computer science is when should I be concerned about a warning and when does it not matter. Perl's answer to this question is to only give you the warnings if you explicitly ask for them. I do not care to extend this argument any further in this book. All of the Perl scripts I will offer you should run without warning when using this argument. Obviously it is a good practice to write code that is devoid of errors and warnings.

Here is an example of a Perl script that generates a warning. Run it with, and then without the `-w` argument.

PROGRAM: warning.pl

```
perl c:\myperl\warning.pl
```

```
perl -w c:\myperl\warning.pl
```

As you can see the script works either way, but does have a warning. My opinion is that you should use the `-w` argument in writing and running your scripts. It does not take all that much extra effort to do this. If it did I might have a different opinion. I will practice what I preach and do this for all the scripts I write. The ultimate decision is yours or your organizations. Our organization is pretty passionate that we account for warnings in our code. In fact, Guardian 90 Perl automatically includes `-w` when you run a script.

PRINT FUNCTION

The print function is used to print to your terminal (this is the default), or to a file.

SYNTAX

```
print <<variable and/or string and or escape character>>;
```

EXAMPLES

```
#Print a string
print "Print this string";
```

```
#Print a couple of strings with a line break in between
print "Print this string,\n Print another string";

#Print a string and a variable
print "Print this string: $myvar";
```

This should be enough for now to get you going.

PROGRAM: print.pl

GETTING INPUT FROM THE TERMINAL

You will ultimately write scripts that need input from the user. Here is a quickie on how to do this.

SYNTAX

```
$<<some variable>> = <STDIN>;
```

EXAMPLES

```
my $myvar = <STDIN>;
```

Since STDIN is assumed by Perl you can write the line above.

```
my $myvar = <>;
```

The equal sign is not an equal sign in Perl it is an assignment operator. In the example above, user input is captured via STDIN and assigned to the variable named \$myvar.

PROGRAM: getinput.pl

SCALARS (WHAT YOU PROBABLY KNOW AS A VARIABLE)

I am caught in a quandry here as to whether to be a Perl purist or to use language that is more generally accepted and understood by the programming community. Since this course is designed for programmers new to Perl I am going to, in the spirit of Perl, use the term scalar but you feel free to think of it as a variable. Before we get started on this subject I want to state for the record that Perl considers the following to be variables:

- Scalars**
- Arrays**
- Hashes**
- Subroutines**
- Typeglobs**

As part of the class this text is used for we will be looking at the first **four**.

Larry Wall defines a variable in Programming Perl 3rd Edition as “a handy place to keep something”. Sounds like an Object Oriented Programming attitude to me. I want you to be aware that technically Perl considers anything in the above list to be a variable. You may very well need to think of scalars as variables. If you do, no problem. I just wanted you to be aware of this distinction. Here are some attributes about scalars.

- In Perl you do not need to worry about data types. Scalars can hold numbers or characters.
- Scalars can be scoped local or global
- Scalar names begin with a \$ when declaring or referring to in code
- Scalars can hold large text strings, words, characters, or numbers
- Scalars cannot begin with an underscore or dash
- Legal characters are a-z,A-Z,0-9,-,_
- Scalar names can be as long as you want (but please do not write a novel)
- Scalar name must contain a character
- The second letter of a scalar must be a character (i.e. – first position after \$)
- Scalars can be declared 'on the fly' unless you use USE STRICT (which I recommend you do). It really does save you time in debugging problems if you are the type of programmer that writes a great deal of code before testing your efforts.

EXAMPLES

```
$myvar;
$my_var;
$my-var;
$myvar = 100;
$myvar = 10.765;
$myvar = "Hello Dolly";
$myvar = "x";
$myvay = $yourvar
```

ILLEGAL SCALAR NAMES:

```
$-X;
$_X;
$34X;
```

PROGRAM: scalar.pl

This program shows use of a scalar. Note that a scalar can be used to store text in the way it is formatted.

Before we continue I want to introduce the use of a directive called STRICT. This is not the place or time to get into a long theoretical discussion on this subject. I would highly recommend using this in your code. Declaring variables on the fly is okay but it can present some interesting debugging issues in complex programs. Use of “USE STRICT” will force you to declare your variables before you use them. Please note that I switched from using the word scalar to using the word variable. I did this because arrays, hashes and subroutines are also considered variables. “USE STRICT” applies to all variables, including scalars.

Please analyze the following two programs as they will give you an idea of working with variables, and with using STRICT in your code with respect to variables.

PROGRAM: scalar 1.pl
 PROGRAM: scalar 2.pl

OPERATORS

There are a number of operators that you can use to work with variables. Run and examine the following programs. They will give you a good introduction to operators in Perl. Note that the examples show use of these operators with scalars.

ARITHMETIC

*	Multiplication
/	Division
+	Addition
-	Subtraction
**	Exponentiation
%	Remainder

PROGRAM: miscoper.pl

LOGICAL

Operator	Or operator	Example
&&	And	\$a && \$b
	Or	\$a \$b
!	Not	!\$a

PROGRAM: and-or.pl

ASSIGNMENT

		Example	Equivalent
=	Assignment	\$a = 5	n/a
+=	Addition and assignment	\$a += 5	\$a = \$a + 5
--=	Subtraction and assignment	\$a -= 5	\$a = \$a - 5
*=	Multiplication and assignment	\$a *= 5	\$a = \$a * 5
/=	Division and assignment	\$a /= 5	\$a = \$a / 5
%=	Remainder and assignment	\$a %= 5	\$a = \$a % 5
**=	Exponentiation and assignment	\$a **= 5	\$a = \$a ** 5
&=	Bitwise AND and assignment	\$a &= 5	\$a = \$a & 5
=	Bitwise OR and assignment	\$a = 5	\$a = \$a 5
^=	Bitwise Exclusive OR and assignment	\$a ^= 5	\$a = \$a ^ 5

.=	Concatenation and assignment	\$a = "can" \$a .= "not" \$a results in "cannot"	
----	------------------------------	--	--

PROGRAM: oper-assign.pl
PROGRAM: assignmentequals.pl

BACKSLASH

\cn	CTRL+n charactor
\e	Escape
\E	Ends the effect of \L, \U, \Q
\f	Form feed
\l	Forces next letter to lowercase
\L	Forces all subsequent letters to lowercase
\n	Newline
\r	Carriage return
\Q	Do not look for special pattern characters
\t	Tab
\u	Forces next letter to uppercase
\U	Forces all subsequent letters to uppercase
\v	Vertical tab

PROGRAM: case.pl

MATCH

=~	Binds a pattern to a string to see if pattern is matched. Used with regular expressions.	\$y =~ /^[a-zA-Z]/
!~	Binds a pattern to a string to see if pattern is not matched. Used with regular expressions.	\$y !~ /^[a-zA-Z]/

STRING COMPARISON

I always loathed the operators below when using them in various report writers over the years. I still run into difficulty remembering to use these instead of the traditional operators (<, <=, ==, etc...).

lt	Less than
gt	Greater than
le	Less than or equal
ge	Greater than or equal
eq	Equal

ne	Not equal
cmp	Compare - returns 1,0,-1

PROGRAM: oper-stringcompare.pl

INTEGER COMPARISON

<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal
!=	Not equal
<=>	Compare - returns 1,0,-1

PROGRAM: oper-integercompare.pl

PARENTHESES

Parentheses are evaluated inner most to outer most.

PROGRAM: parenth.pl

CONDITIONAL LOGIC

The following is for your review. This is pretty basic stuff so, except for the topic of blocks, no other narrative is included. The syntax and toy code examples get you through.

{ } is a block. If you have never worked with a language that uses blocks then you should still be reading. A block allows us to hold and process more than one statement as a singular unit of logic. Blocks should always be used when coding conditional logic (something that is not the case for C or Java programs). Another important thing to know about a block is that variable scope within a block is only for the duration of the block.

IF

PROGRAM: if.pl

SYNTAX

```
if (<<condition>>) {  
    <<outcome1>>;  
    [[<<outcome2>>;]]  
    [[<<outcomeN>>;]]  
}
```

EXAMPLE

```
if ($x eq "Hello") {
    $y=$x;
    print ($y);
}
```

IF/ELSE

Spend some time looking this over, as the syntax is a bit different than what you might expect.

PROGRAM: if-else.pl

SYNTAX

```
if (<<condition>>) {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
}
else {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
}
```

EXAMPLE

```
$x eq "Hello";
$y eq "Goodbye";
if ($x ne "Hello") {
    $y=$x;
    print ($y);
}
else {
    $x=$y;
    print ($y);
}
```

IF/ELSIF

Ditto!

PROGRAM: if-elsif.pl

SYNTAX

```
if (<<condition>>) {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
}
elsif (<<condition>>) {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
}
#as many other elsif as you need
```

```
[[elsif (<<condition>>) {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
}]]
[[ else {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
}]]
```

EXAMPLE

```
if ($x eq "Celtics") {
    print ("Bob Cousey");
}
elsif ($x eq "Bulls") {
    print ("Michael Jordan");
}
elsif ($x eq "Lakers") {
    print ("Jerry West");
}
else {
    print ("Don't know that team");
}
```

NESTED IF

PROGRAM: if-nested.pl

SYNTAX

```
if (<<condition>>) {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
[[if (<<condition>>) {
    <<outcomel>>;
    [[<<outcome2>>;]]
    [[<<outcomeN>>;]]
}]]
}
```

EXAMPLE

```
use strict;
my $str = " ";
my $v;
if ($str) { # because i initialized $str to a space it evaluates true here.
# If not initialized or initialled to "" it would evaluate false. This is
# called scalar truth.
    if ($v eq "T") {
        $str = "Hello";
    }
    my $flg = "Y";
    if ($flg eq "Y") {
        $flg = "N";
    }
}
```

```

        }
    print ($flg, " ", $str);
}

```

LOOPS

Perl offers a number of different types of loops for you to use. Presented are some examples for each type.

For

PROGRAM: forloop.pl
 PROGRAM: forloop2.pl
 PROGRAM: forloop3.pl

SYNTAX

```

for (<<loop initializer>>; <<loop terminator>>; <<iterator>>) {
    <<statement block>>
}

```

While

PROGRAM: whileloop.pl
 PROGRAM: whileloop2.pl

SYNTAX

```

while (<<loop condition>>) {
    <<statement block>>
}

```

Until

PROGRAM: untilloop.pl

SYNTAX

```

until (<<loop condition>>) {
    <<statement block>>
}

```

Note: Loops can be nested. I have met programmers that believe one should never write nested loops. I raise this subject because one tends to write a lot of loops in Perl. I prefer to code nested loops instead of breaking them out. This is how I think. I have no axe to grind on this subject.

ITERATION OPERATORS

PROGRAM: oper-inc-dec.pl
 PROGRAM: oper-inc-string.pl

FILE IO

As I said earlier, Perl is a great language for working with text. It is time to examine how to do some file IO.

Reading files

SYNTAX

```
open (<<"FILEHANDLE">> , <<"filespec">>);
```

PROGRAM: filio-open.pl

Note that in the example above we dump the file into an array and print the array to the display. Also, this program opens itself and prints its code to the display. Even if we have this file open our program will still work (this is not the case in Guardian 90).

I want to point out to you in case you did not notice the sheer power of this program. We opened a text file and dumped each line into an array with 2 lines of code! Whoa! This is just beyond cool. Of course you need to be careful doing this in Perl scripts that will be widely used on systems that will have a lot of folks using your program on files that could get quite large. For example, I have a delimited text file that is 28mb. I do not want to dump that file using this method! You get my drift. Incredibly powerful but be careful and use some thought when using this technique. If you are a novice programmer your inclination may be to use this as a technique. DO NOT.

The following example's accomplish the same thing as the first one, but do so via a loop. This gives us a bit more control as we parse each line. Also we have added a count of the total number of lines in the program. After analyzing the first program make sure you run and examine the second program. Use the -w argument when running these programs. Note they accomplish the same thing, though the second one returns a warning when we run it.

PROGRAM: filio-open2.pl
PROGRAM: filio-open2a.pl

I want to point out what some may consider a Perlism. In the second program we receive the warning "Use of uninitialized value at ... line 36, <fil> line 14." The programs are identical except for the exit mechanism of our loop.

PROGRAM: filio-open2.pl
while (!eof) {

PROGRAM: filio-open2a.pl
while (\$ln ne "") {

This works the same as filio-open2.pl but when we run it with the -w argument we receive the warning "Use of uninitialized value at...". The difference between this script and the previous one is that we are executing the loop based on the condition of the line we are parsing in the file. We receive the warning because once Perl is at eof it closes the file and with the file closed we hit the loop where we are passing the point that <fil> is at to our variable \$ln. Problem is once we hit eof Perl has no reason to keep our filehandle variable and so it is toast, hence the warning, as we have no value for \$ln because it is no longer in scope.

We can easily remedy this situation by making a minor change in the program below

```
PROGRAM: filio-open2b.pl
while ($ln) {
```

By changing the loop condition to “\$ln” we no longer receive the warning. This is because our test is for the existence of the scalar itself and not the contents of the scalar. Finally I want to show you one final means of coding this example.

```
PROGRAM: filio-open2c.pl
while (<FIL>)
{
    print $_;
    $cnt++;
}
```

This method is the one that seems to be preferred by most seasoned Perl programmers. In fact, there are some folks that will make you feel less than adequate if you do not code using this convention. I refer you back to what the inventor of Perl has to say on this very subject (covered in the preface).

This is rather lean (if not also obtuse to a new Perl programmer). In the loop our test is against the filehandle. As long as we are reading the file the filehandle will have scope that we can test. Also notice the strange use of \$_. \$_ is the input line variable. This variable is available for your use for any type of IO. Very nice to have a predefined scalar to use.

What is even more interesting is that we do not even need to reference \$_ in the program. Just saying “print” by itself will do the trick. Perl knows that you must be referring to \$_. And actually, by default the output part of our I/O is assigned by default to STDOUT. The default for STDOUT is the console. Perl does not make us explicitly add the reference in our code. It just so happens that as we read each line of the input from our loop (in any of the programs I have cited about as examples) Perl stashes what it reads to the scalar \$_. So the contents of \$_ (what we are getting with each line we read from our filehandle) is written to SDTOUT (which by default is the console). Yes, this is quite a lot to absorb, but I think it is good to know how things really work.

```
PROGRAM: filio-open2c1.pl
while (<FIL>)
{
    print;
    $cnt++;
}
```

One last Perlism before moving on. It is also possible to redirect STDIN. Consider the next example.

```
PROGRAM: filio-open2c2.pl
open (STDIN, "filio-open.pl") || die ("can't open file");
my $cnt=0;

while (<>)
{
    print;
    $cnt++;
}
```

In this example I have redirected STDIN to read from a file. As you have already seen STDIN by default reads from the console. By naming my filehandle STDIN, it actual becomes the STDIN. Notice also that since my handle has been assigned to STDIN I can just use the angle brackets (<>) by their lonesome. This is certainly a Perlism you are likely to encounter in your journey.

It is interesting to add at this point that all these programs work the same and deliver the same goods. I generally am not the type to show multiple methods of accomplishing the same thing. I deviated from this general philosophy on this occasion as I thought there were a number of hidden truths about Perl I wanted to expose you to. Also as you look at other programs I wanted you to be familiar with different coding techniques.

Moving on to a few quick odds and ends.

Using either the standard error handling, and/or return your own error text.

PROGRAM: filio-die.pl

Enter the file you want to open at program runtime

PROGRAM: filio-read.pl

Now let us take a look at how we write to files.

Copying the contents of a file to a new file, or recreate an existing file

PROGRAM: filio-io1a.pl

```
use strict;

open ("FILL", "FILE-for-filio-write.txt") || die ("can't open file");

open (STDOUT, ">Newfile.txt") || die ("can't open file");
```

Note, in the line above we are specifying the file we want to either create, or blow away and re-populate. This is done by using ">" before our reference to the file. Also notice that I have chosen to use STDOUT for the filehandle. STDOUT can be used in conjunction with the print function.

```
while (<FILL>)
{
    print STDOUT;
}
```

Since our filehandle is STDOUT, the contents of \$_ for each execution of our loop will go to the file named "Newfile.txt".

```
close(STDOUT);
```

We want to be sure and close STDOUT. This will reset its default, which is to output to the console.

Now that I have shown you this... **WARNING, WARNING, BE CAREFUL WHEN DOING THIS!!!**

Appending a file with the contents of another file

PROGRAM: filio-io1b.pl

```
open (STDOUT, ">>FILE-for-filio-write.txt") || die ("can't open file");
```

To append to an existing file use ">>".

Other techniques...

I am not going to offer any examples here. I have shown you how to pass data from one file to another in the above example. You can also populate data in a file from input from the console, or by dumping the contents of an array to the file. A word of caution. I am not advocating or suggesting that you open a file, assign the contents of a file to an array, and then dump the array into another file. While simple to achieve in code this is a very, very inefficient means of doing business. I went to the pain (well not really pain) of showing you the technique in the examples above so you would not do this! There certainly may be times when it is preferable to IO a file, extract from the file to an array that you will use for further processing and manipulation. My point has been that while Perl is a most powerful and accessible language to mere mortals like you and I, it can be used in a way that consumes (no hogs) system resources. And, coupled with the fact that there are many ways to do the same thing you really need to give some thought to what you are doing and how it might impact the system and others using the system.

Using "die" in file IO.

PROGRAM: filio-die.pl

"die" should be used when opening files as it allows you to gracefully do something if the file open operation fails. Here are two examples of coding "die" in you file IO statement.

```
if (open ("FIL", "/myperl/filio-open.pl"))
{
    die ("file was opened");
}
```

or

```
open ("FIL", "/myperl/filio-open.pl") || die ("can't open file");
```

ARRAYS

This is an area where Perl really shines. Implementation of arrays in Perl is very robust. For this 101 section we will only be covering a portion of the functionality of arrays. Later on we will go into more details on arrays, and associative arrays, or hashes as they are referred to in Perl.

SYNTAX

```
@<<name_of_array>> [[ = ( [[<<elem1, elem2, elemn>>]] )]];
```

EXAMPLE

Program: array.pl

```
@x;          #an empty array
@y = ();     #also an empty array
@z = (a,b,c,1,2,"3",1+3,"1"+"3");
```

Since there is only one data type in Perl you can mix numbers and text. You can even use expressions in arrays. In the above example we see that both the expression `1+3` and `"1"+"3"` evaluate to 4. "1" and "3" are stored in the array as text. Perl knows to convert them to numbers when it sees them as part of a mathematical expression. (Note: There is more overhead in dealing with "4" + "1" than in dealing with `4 + 1`)

When dealing with arrays some topics come to mind.

- How can I determine the length of an array?
- Can I assign an array to an array?
- Can I dump the contents of a file into an array?
- How do I access an element in an array?
- When I print an array, how can I put a space or other separator between each element?
- How can I sort an array?

PROGRAM: array1.pl
PROGRAM: arraysort.pl

Determining the length of an array

You do not have to write a loop to count the items in an array. Just assign a scalar to the array and this will give you the count.

```
$length = @arr;
```

Do not put parentheses around the variable or the first item in array will be returned.

```
($length1) = @arr;
```

Another way of determining the length of an array is...

SYNTAX:

```
 $#<arr_name> + 1
```

EXAMPLE:

```
print $#arr + 1;
```

This will return the last subscript of the array. Since arrays are zero based you need to add one to get the correct length.

Accessing an array item by subscript

```
#accessing an element in an array, note 0 based
print ($arr2[2], "\n");
```

Finding an array element

```
for ($i=0; $i<@arr2; $i++) {
    if ($arr2[$i] eq "three") {
        print ($arr2[$i], "\n");
        print ("element ", $i, " in the array", "\n");
    }
}
```

Assigning an array to an array

```
@arr3 = @arr2;
```

Sorting an array

```
#sorting an array and rebuilding an array in sorted order
#note that sort is alphabetical not numeric
@x = (1, 5, 2, 30, a, A, Z, x, This, That);
print("@x\n");
@y = sort(@x);
print("@y\n");
@x = @y;
print("@x\n");
@y = reverse (@y);
print("@y\n");
```

These examples should offer you a good foundation on arrays for now.

PATTERN MATCHING

If you are already familiar with regular expressions you will really be able to take off and use Perl. If you are not then you will have some learning ahead of you. Regular expressions is a language unto itself. It is also a topic that will take you a great deal of effort to understand and even longer too fully master. Consider the following regular expression.

```
$str =~ \^[/#.*a-z +]*\?\io;
```

Now that I have shocked you let me say that you do not have to be a guru at regular expressions to use them. I once heard these referred to as line noise and thought that to be an accurate description. I am going to provide the drop dead minimum to get you started and to keep you engaged in the language. Understanding regular expressions will take your programming to a new level.

You no doubt have used wildcard operators (*,?) before. What we will be doing here is not that much more involved than that. Here we will be covering the drop dead essentials of regular expressions and

their association to pattern matching constructs in Perl, hence the title “Pattern Matching”. Later on we have a whole mind bending (not really) chapter devoted to regular expressions.

The =~ and !~ operators

In a pattern matching statement you can use the =~ or !~ operators. Consider the following.

```
$str =~ /a/;
```

=~ reads “**matches**” and the about statement translates into...
"content of the variable \$str matches the character 'a'"

```
$str !~ /a/;
```

!~ reads “**does not match**”

You will notice that as I cover this subject that I am keen on translating the regular expression into something that resembles verbage that a human can readily understand. You too should get into this habit.

The + and * Character

Program: patt1.pl

The + operator matches one or more preceeding characters. The * operator matches zero or more preceeding characters. Consider the following code.

```
@arr = (me, meet, meat, meaning, melvin);
print ("Match - mea: with the asterisk\n");

for ($i=0;$i<5;$i++) {
    if (@arr[$i] =~ /mea*/) {
        print ("@arr[$i]\n");
    }
}

print ("\n");
print ("Now the plus\n");
print ("\n");

for ($i=0;$i<5;$i++) {
    if (@arr[$i] =~ /mea+/) {
        print ("@arr[$i]\n");
    }
}
```

In the first loop we are looking to match the pattern mea*. This pattern will match.

```
me
meet
meat
melvin
meaning
```

You probably are not sure at this point why this matched anything other than "meat" and "meaning". You are thinking this because of your understanding of how the asterisk operator is typically

implemented. You are going to have to break this notion when you see this pattern operator in regular expressions. Based on our definition of the * operator (matches zero or more preceding characters) this pattern (mea*) reads...

Match any string that begins with "me" and is followed by zero or more occurrences of the letter "a".

The + operator will only match meat and meaning. This is because the requirement now reads...

Match any string that begins with "me" and is followed by one or more occurrences of the letter "a".

Do not think that this operator works the same as the traditional asterisk operator. It does not. The traditional asterisk operator matches "mea" as a string. In our use, the asterisk after the "a" means we still will match even if there is not an "a". Quite different!

Just to make sure you get this. What will the following code match?

Program: patt1b.pl

```
@arr = ("b", "bb", "bbb", "bba", "bab");

for ($i=0;$i<5;$i++) {
    if (@arr[$i] =~ /bb*/) {
        print ("@arr[$i]\n");}
}
```

Matches: _____

```
@arr = ("b", "bb", "bbb", "bba", "bab");

for ($i=0;$i<5;$i++) {
    if (@arr[$i] =~ /bb+/) {
        print ("@arr[$i]\n");}
}
```

Matches: _____

[] (Character class)

PROGRAM: brackets.pl

Brackets are used to group a set of characters you want to match. These are actually called character classes. Consider the following code fragment.

```
$str = "abcdefghij";

if ($str =~ /[fik]/) {
    print ("$str\n");}
```

The pattern [fik] reads, *match any pattern that contains the letters "f" or "i" or "k".*

Match any letter of the alphabet, case insensitive

```
/[a-zA-Z]/
```

or

```
/[a-z]/i
```

Match any number

```
/[0-9]/
```

Match a special character

"\" is used for special characters within a pattern. Without this you will receive a syntax error.

SYNTAX

```
\<special_character>
```

EXAMPLE

```
/\?/
```

```
/\.\?abc/
```

```
/?/ #results in a syntax error
```

Using SPLIT

Program: split.pl

SPLIT comes in handy when you need to tokenize a string. In the code example below we want to dump a string into an array. The assumption is that each element stored will be a word or punctuation. But if we assume that each word is separated by only a space, how do we deal with situations where there are multiple spaces?

```
$str = "this is the way to Ed and Fred and Teds place";  
@arr = split (/ /, $str);  
$cnt = @arr;  
print "$cnt\n";
```

In the next example we use split to store sentences in the array

```
$str1 = "this is the one? This is two! This is three."  
@arr1 = split (/[\.!\?]/, $str1);  
$cnt1 = @arr1;  
print "@arr1\n";  
print "$cnt1\n";
```

Notice that the array count is three. What is stored in our array is...

```
@arr[0]=this is one  
@arr[1]=this is two
```

```
@arr[2]=this is three
```

This is enough for now. We will be spending more time with regular expressions later in the book. We have only scratched the surface of this subject.

SUBROUTINES

I would be remiss if I did not cover this topic in the 101 section. We will cover this subject in more detail later. For now here are some of the basics.

Calling a subroutine

SYNTAX

```
&<<subroutine_name>>;
```

EXAMPLE

```
&my_sub;
```

Subroutine example

PROGRAM: sub1.pl

```
use strict;
# global vars
my $a=1;
my $b=2;
my $c=3;

# program flow
&mysub;
&mysub2 ($a,$b,$c);
&mysub3 ($a,$b,$c);
print ($a,"\n");
print ($b,"\n");
print ($c,"\n");

# subroutines
sub mysub {
# note our sub has scope to the global variables
    print "$a$b$c\n";
}

sub mysub2 {
# pass call args to our local vars
    my ($a,$b,$c) = @_;

# so which $a are we referring to?
    my $a=4;
    print "$a\n";
# ...and how can we access the global vars
    my $a = $c;
    my $b = $c;
    my $c = $c;
    print "$a$b$c\n";
}
```

```
}  
  
sub mysub3 {  
    my $a=10;  
    my $b=20;  
    my $c=30;  
    print ("$a ", "$b ", "$c ", "\n");  
}
```

GETTING INPUT FROM THE TERMINAL

This is a very simplistic program that accepts a line of input from the console and displays it back.

PROGRAM: getuserinput.pl

```
$mystr = <STDIN>;  
print ($mystr);
```

CHOP AND CHOMP

These two are frequently used so these examples are offered for your inspection.

PROGRAM: chop-chomp.pl

CHOP

CHOP removes the last character from a scalar value

```
my $word1 = "Goodbye ";  
  
print $word1;  
print "\n\n";  
  
print "Using Chop on a word";  
print "\n";  
print "-----";  
print "\n";  
  
## Using chop on a word  
chop $word1;  
print $word1;  
print "\n";  
  
chop $word1;  
print $word1;  
print "\n";  
print "\n";  
  
$word1 = "Goodbye";
```

CHOMP

Checks to see if the last character of a string or list of strings matches the input line separator. If it does `chomp` removes them.

```
## Using chomp

my $word3 = "Goodbye";
if ($word1 eq $word3) {
    print "match\n";
} else { print "no match\n"; }

my $word2 = " ";
chomp ($word2);
if ($word1 eq $word3) {
    print "match\n";
} else { print "no match\n"; }

print "type Goodbye and press enter: ";
$word2 = <stdin>;

if ($word1 eq $word2) {
    print "match\n";
} else { print "no match\n"; }

chomp ($word2);
if ($word1 eq $word2) {
    print "match\n";
} else { print "no match\n"; }

$word1 = "Goodbye "; # we add a space
if ($word1 eq $word2) {
    print "match\n";
} else { print "no match\n"; }

chomp ($word2);
if ($word1 eq $word2) {
    print "match\n";
} else { print "no match\n"; }
```

PROGRAM: pattern2b.pl

Provides a practical example of using `chomp`. Run the program then analyze the code.

```
#get a search string
print "Enter a string \n";
my $str = <stdin>;
print ("Search string entered: $str","\n");

#get a pattern
print "Enter a pattern \n";
my $pat = <stdin>;

#do not forget chop or chomp with this or it will not work like the
#previous hardcoded version
```

```
chomp ($pat);
print ("Search string entered: $pat","\n");

print ($pat,"\n");
my $rslt = $str =~ /$pat/;
print ("String to search: $str","\n","Did we get a match?: $rslt");
$rslt = $str !~ /$pat/;
print ("\n","Did we NOT get a match?: $rslt");

my @arr = split (/ $pat/, $str);
my $cnt = @arr;
print ("\n","Whats left in array: @arr","\n","number of hits: ", ($cnt-1) );
```

EXERCISES TO APPLY WHAT YOU HAVE LEARNED

You have now covered enough turf that it is time for some hands-on.

1. Open up exe1_1a.pl and complete the code to the requirements specified.
2. Open up exe1_2a.pl and complete the code to the requirements specified.
3. Open up exe1_2b.pl and complete the requirements specified.
4. Open up exe1_2c.pl and complete the requirements specified.

MORE ON SCALARS

In this chapter we will explore in detail how to work with scalars. The assumption is that you are already familiar with these terms as well as other terms like scope and persistence.

I will present a series of working trivial code examples to demonstrate these subjects.

Perl has one data type. Data is not typed as it is in C, C++, Cobol, and Java. It is more like Mumps (M).

One final note: The examples presented in this manual will be done using `USE STRICT`. This convention will not allow us to declare variables "on the fly". Also we will use the "my" keyword when declaring our variables. "my" gives us lexical scoping of our variables. More on this subject will be covered later.

LEGAL NAMES

SYNTAX:

`$<name_of_scalar>`

- Can contain -, _, a-z, A-Z, 0-9
- First character of name cannot be a - or _ or a number
- Second character must be a letter
- Cannot contain . or :

EXAMPLES:

```
$x;  
$my-var;  
$my_var;  
$str1;
```

Scalar names can be as long as you want but one should exercise some reason.

```
$my_scalar_that_is_long_but_in_english_so_anyone_can_understand_what_I_mean;
```

Yeah, right!

Names are case sensitive. `$SCALAR <> $Scalar <> $scalar`.

ASSIGNMENT

PROGRAM: assignment.pl

SYNTAX:

```
$<name_of_scalar>=<assignment_value>;
```

Note: It is not necessary to put white space between the operator and operands. This author believes that using white space makes your program more readable.

```
$<name_of_scalar> = <assignment_value>;
```

An assignment value can be a string, a character, another scalar, a number, or an expression.

Assigning a character or string to a scalar

```
$str = "this is a string";
```

Assigning a scalar to a scalar

```
$str2 = $str;
```

Assigning two scalars to a scalar using the concatenation operator

```
$str3 = $str .= $str2;
```

The concatenation operator offers some strange results when attempting to concatenate the same scalar as in the following example:

```
$str3 = $str .= $str;
```

Assigning a number to a scalar

```
$str4 = 10;
```

Assigning the results of an expression to a scalar

```
$str5 = (5 * ($str4 + 4));
```

Parentheses are resolved inner to outer.

Indirection

```
$str6 = $str7 = "hello"
```

MISC. SCALAR STUFF

UNDEF/DEFINED or Perl's equivalent to NULL

PROGRAM: undef.pl

When you declare a scalar in Perl and do not initialize it, its state is "not defined". You can test the state of a scalar using the keyword `DEFINED` as in the code fragment below.

```
my $str;
if (defined $str) {
    print "scalar \$str is undefined\n";
}if (!defined $str) {
    print "scalar \$str is undefined\n";
}
```

It is also possible to reset a scalar back to an uninitialized state as illustrated in the code below. The UNDEF keyword is the equivalent of NULL in SQL.

```
my $str2 = "foo";
undef $str2;
if (defined $str2) {
    print "scalar \$str2 is undefined";
}if (!defined $str2) {
    print "scalar \$str2 is undefined";
}
```

Testing the state of a scalar

PROGRAM: tf-varitest.pl

It is possible to check the state of a scalar in a couple of ways. First we can create a scalar and set it to 1 (True) or 0 (False).

```
my $tf=1; # 0 is false 1 is true
my $i=0;

while ($tf) {
    if ($i == 10) {
        $tf = 0;
    } else {
        $i++;
        print ($i," ");
    }
}
```

We can also check to see if a scalar has a value or has not been initialized.

```
my $var;
if ($var) {
    (print "initialized","\n");
} else {
    (print "not initialized","\n");
}

my $var = "some value";
if ($var) {
    (print "initialized","\n");
} else {
    (print "not initialized","\n");
}
```

Now that you have seen these programs you may be a bit puzzled. A scalar can be tested for truth. A scalar that has not been initialized will be false. A scalar with text will be true. A scalar with a number greater or less than 0 (not "0") will be true. A scalar set to 0 will be false.

Converting a character to ASCII and Binary

PROGRAM: `ascii_binary.pl`

```
my $alpha = "A";
my $alpha2 = "0";

my $bin = unpack("B8", pack("A", $alpha));
print ("$alpha: ", $bin, "\n");

my $bin2 = unpack("B8", pack("A", $alpha2));
print ("$alpha2: ", $bin2, "\n");

print "\n";

my $a="A";
my $i;
my $b;
my $c;
for ($i=0;$i<26;$i++) {
    $b = unpack("B8", pack("a", $a));
    $c = unpack("C", pack("B8", $b));
    print ($a, " ", $b, " ", "$c", "\n");
    $a++;
}
}
```

There are a wealth of conversion routines for working with the pack/unpack functions. "B8" formats the data into a byte (8 bit) format. "C" format the data as the ascii value for a character.

The following program provides a list of the ascii character set.

PROGRAM: `pack.pl`

Octal/Hex/Decimal

PROGRAM: `octhex.pl`

Rounding Numbers

PROGRAM: `rounding.pl`

SCOPE/PERSISTENCE

Please note that the principles of scope apply to all types of Perl variables. I do not plan on addressing this subject again so I have purposely used the word variable instead of scalar.

The subject of scope is important and oftentimes overlooked. This is an area that tends to be very different from language to language. Variables that are declared outside of subroutines are global in scope to the Perl program. They are persistent for the life of the program.

PROGRAM: scope.pl

```
my $i=7;
&mysub;
sub mysub {
    my $i=9;
    print ("sub \${i}: ",$i,"\n");
}
print ("global \${i}: ",$i);
```

This piece of toy code demonstrates that variables inside of subroutines are isolated from global variables and variables in other subroutines. It is important to use “my” in our declaration as it actually serves to limit scope. Consider the following code.

PROGRAM: scope2.pl
 PROGRAM: scope2b.pl
 PROGRAM: scope2c.pl

```
&mysub;
sub mysub {
    $i=9;
    print ("sub \${i}: ",$i,"\n");
}
print ("\${i} outside the sub: ",$i," hmmmmm");
```

Because we do not use “my” in declaring the variable \$i we expose it to areas outside of the subroutine. If this is something you desire then bag the “my”. It is also possible, and quite practical to pass variables among subroutines. Consider the following two programs.

PROGRAM: scope2d.pl

```
&mysub;

sub mysub {
    my $i=9;
    print ("sub \${i}: ",$i,"\n");
    &mysub2($i);
}
sub mysub2 {
    my $x = shift(@_);
    print ("sub \${x}: ",$x,"\n");
}
```

PROGRAM: scope2e.pl

```
print "Enter num1: ";
my $a = <STDIN>;
print "Enter num2: ";
my $b = <STDIN>;
&callsub;
sub callsub {
```

```
        print &multiply($a,$b);
    }
sub multiply {
    my $x;
    my $y;
    ($x, $y) = @_;
    return ($x * $y);
}
```

You can see that I have just exposed you to the `@_` (a Perlism, depending on who you ask). In any event `@_` is the list (array) of arguments passed into a subroutine.

The final topic for this discussion is the persistence or life of a variable within a subroutine. While global variables persist for the life of the programs, a variable within a subroutine hangs around only for the duration of the call. Consider the following code.

PROGRAM: scope2f.pl

```
&mysub;
sub mysub {
    my $i=9;
    print ("sub \ $i: ", $i, "\n");
    &mysub2($i);
}
sub mysub2 {
    my $x = shift(@_);
    print ("sub \ $x: ", $x, "\n");
}
&mysub2;
```

MORE ON ARRAYS

PUSHING AN ARRAY

push.pl

```
use strict;

# create empty array
my @arr = ();

push (@arr, 1);
push (@arr, 2);
push (@arr, 3);
push (@arr, 4);
push (@arr, 5);

# print 1st element in array
print $arr[0];
```

POPPING AN ARRAY

pop.pl

```
use strict;

# create empty array
my @arr = ();

push (@arr, 1);
push (@arr, 2);
push (@arr, 3);
push (@arr, 4);
push (@arr, 5);

my $Aval = pop(@arr);

# print 1st element in array
print $Aval;

print "\n";

print @arr; # NOTE! element we popped is no longer in the array
```

REVERSING AN ARRAY

Program: reverse_array.pl

```
use strict;
```

```
# create empty array
my @arr = ("two", "one", "ten");

@arr = reverse @arr;

print @arr;
```

SHIFTING AND UNSHIFTING

Shift

This is similar to pop except that it works from the front of the array.

Program: shift.pl

```
use strict;

# create empty array
my @arr = (1,2,3,4,5,6);
my $arr = @arr;

my $v = shift(@arr);
print $v;

print "\n";

print @arr;
```

Unshift

This is similar to how push works, except that it can be used to add an element to the beginning of an array. Consider these two programs.

Program: unshift.pl

```
use strict;

# create empty array
my @arr = (1,2,3,4,5,6);
my $arr = @arr;

unshift (@arr, 'where am I?');

print @arr[0];
```

Program: push2.pl

```
use strict;

# create empty array
my @arr = (1,2,3,4,5);
```

```
push (@arr, 'where am I?');  
  
# print 1st element in array  
print $arr[5];
```

MORE ON SUBROUTINES

I am from the school that feels that writing structured programs is very important. I believe that if at all possible (and usually it is) that code should be broken out such that a subroutine performs a specific function. I also believe that such code should be as generalized as time allows so that it can be reused.

I am waiting this late in the game to address this subject for a couple of reasons. I have not introduced you to any code of a scope or complexity at this point that would make you pause to consider this subject. Also, some pretty quick and highly functional scripts can be written without broaching this subject.

I do want you to think about this subject. As you begin to find a myriad of uses for Perl, you will want to develop your scripts using subroutines. Why? It will make it easier to organize your code so that you can reuse it! And, while Perl can accomplish a lot in a few lines of code you can also write programs that are hundreds and thousands of lines of code.

SUBROUTINE DECLARATION

SYNTAX:

```
sub <<subroutine_name>>
```

EXAMPLE:

```
sub mysub
```

CALLING SUBROUTINES

It is possible to call a subroutine without using the subroutine dereferencer. I will not be showing you this way, but thought I would reference it as you will likely encounter this convention in others code.

SYNTAX:

With subroutine dereferencer

```
& <<subroutine_name>>
```

With subroutine dereferencer

```
<<subroutine_name>>
```

EXAMPLE:

```
&mysub
```

or

```
mysub
```

I would encourage you to use the dereferencer when calling a subroutine. It makes the code easier to read and spot a call, and it could come in handy if you wanted to write a Perl program that reads through Perl programs and finds all the subroutine calls.

Below is a barebones Perl program that demonstrates a subroutine call.

```
suba.pl

use strict;
&callsub;

sub callsub
{
    print "subroutine callsub was called\n";
}
```

PASSING PARAMETERS

If you are a C or C++ programmer you will see that a subroutine in Perl is really the equivalent of a function (granted, the technical details are quite different). It is possible to pass parameters to a subroutine as part of the call. You can pass a single value or an array of values.

```
subb.pl

use strict;
# global vars
my $a="variable a";
my $b="variable b";
my $c="variable c";

&mysub ($a,$b,$c);

sub mysub
{
    print "$a\n$b\n$c\n";
}
```

What is really neat is that the arguments you pass get stored in a special array `@_`. In reality this is the construct that Perl uses to deal with values passed to subroutines. You have access to this (as you have access to many things like this throughout Perl).

For example, consider the following code. It does the same thing as the program above, but does so using `@_` and `$_`.

```
subb2.pl

use strict;
# global vars
my $a="variable a";
my $b="variable b";
my $c="variable c";

&mysub ($a,$b,$c);

sub mysub
{
    my $i=0;
```

```
my $sm=@_;
  for ($i=0; $i<$sm; $i++)
  {
    print "$_[ $i]\n"
  }
}
```

`$_` is the default variable. In the context of our program it represents the argument being passed to the subroutine. We could have coded...

```
print "@_[ $i]\n"
```

...but we would have received the warning "Scalar value `@_[$i]` better written as `$_[$i]` at subb2.pl... if we run our program using the `-w` argument.

USING A VARIABLE NUMBER OF ARGUMENTS

It is also possible to pass an array into a subroutine. You can again see how `@_` and `$_` come in handy. For one thing you can code a subroutine so it does not have to be concerned about the name of the thing or things you are passing into it!

subd.pl

```
use strict;
my @arr = ("a",5,"c");
&mysub (@arr);

sub mysub
{
  my $i=0;
  my $sm=@_;
  for ($i=0; $i<$sm; $i++)
  {
    print "$_[ $i]\n"
  }
}
```

VARIABLE SCOPE IN A SUBROUTINE

Variables within a subroutine have scope for the duration that the subroutine executes.

PROGRAM: scope.pl

```
my $i=7;
&mysub;
sub mysub {
  my $i=9;
  print ("sub \ $i: ", $i, "\n");
}
print ("global \ $i: ", $i);
```

This piece of toy code demonstrates that variables inside of subroutines are isolated from global variables and variables in other subroutines. It is important to use “my” in our declaration as it actually serves to limit scope. Consider the following code.

PROGRAM: scope2.pl
 PROGRAM: scope2b.pl
 PROGRAM: scope2c.pl

```
&mysub;
sub mysub {
    $i=9;
    print ("sub \${i}: ", $i, "\n");
}
print ("\${i} outside the sub: ", $i, " hmmmmm");
```

Because we do not use "my" in declaring the variable \$i we expose it to areas outside of the subroutine. If this is something you desire then bag the "my". It is also possible, and quite practical to pass variables among subroutines. Consider the following two programs.

PROGRAM: scope2d.pl

```
&mysub;

sub mysub {
    my $i=9;
    print ("sub \${i}: ", $i, "\n");
    &mysub2($i);
}
sub mysub2 {
    my $x = shift(@_);
    print ("sub \${x}: ", $x, "\n");
}
```

PROGRAM: scope2e.pl

```
print "Enter num1: ";
my $a = <STDIN>;
print "Enter num2: ";
my $b = <STDIN>;
&callsub;
sub callsub {
    print &multiply($a,$b);
}
sub multiply {
    my $x;
    my $y;
    ($x, $y) = @_;
    return ($x * $y);
}
```

Finally for this discussion is the persistence or life of a variable within a subroutine. While global variables persist for the life of the programs, a variable within a subroutine hangs around only for the duration of the call. Consider the following code.

PROGRAM: scope2f.pl

```
&mysub;
sub mysub {
    my $i=9;
    print ("sub \ $i: ", $i, "\n");
    &mysub2($i);
}
sub mysub2 {
    my $x = shift(@_);
    print ("sub \ $x: ", $x, "\n");
}
&mysub2;
```

STATIC VARIABLES IN SUBROUTINES

It is sometimes desired to keep variable state within a subroutine. Perl does not offer a direct means of making a variable within a subroutine static. A solution or workaround to this would be to declare the variable as global and refer to it within the subroutine. Not all that elegant but it will get the job done.

Suppose you want a subroutine that increments a counter. The following program demonstrates how to accomplish this.

Program: sube.pl

```
use strict;

my $i=0;

sub mysub
{
    ++$i;
    print "$i\n";
}

&mysub ($i);
&mysub ($i);
&mysub ($i);
&mysub ($i);
```

Note that if we coded the variable within the subroutine we would always return 1.

RETURNING VALUES

Scalars

We can return scalar values or arrays from a subroutine. In fact we can even return hashes. The following code demonstrates how to return a variable.

subf.pl

```
use strict;

my $i=10;

sub mysub
{
    $i = "ten";
    return $i;
}

PRINT $i = &MYSUB ($i);
```

As you can see we can call a subroutine as part of an expression. That statement could also have been written:

```
&MYSUB ($i);
PRINT $i;
```

Either way will yield the same result. Below is another example.

subg.pl

```
use strict;

my $a=0;
my $b=10;
my $c=8;

sub mysub
{
    $a = ($b * $c);
    return $a;
}
#call mysub
$a = &mysub ($b,$c);
print "$a\n";

# the above example works fine but does defeat the
# containment principle of a subroutine by acting on
# a global variable.

sub mysub2
{
    my $z=0;
    $z = ($b * $c);
    return $z;
}
#call mysub2
$a = &mysub2 ($b,$c);
print $a;
```

If your intent is to reuse a subroutine you need to strongly consider the use of global variables. Even rewritten, `mysub2` still relies on the scalar variables `$b` and `$c`. How could you rewrite this routine to act independent of these variables?

Arrays

```
subi.pl

use strict;
my @i = (1, 5, "foo", "end");

&mysub(@i);

sub mysub
{
    foreach (@_)
    {
        print "$_\n";
    }
}
```

Refer to the "Passing Parameters" topic in this section for a discussion on the use of `@_` and `$_`.

Hashes

You can also pass a hash (associative array) into a subroutine.

```
subg.pl

use strict;

my %hsh =
(
    'batting_avg' , '306' ,
    'hits'        , '83'  ,
    'homeruns'    , '9'   ,
);

&mysub(%hsh);

sub mysub
{
    my $key="";
    %hsh = @_;
    foreach $key (keys %hsh)
    {
        print "$key = $hsh{$key}\n";
    }
}
```

REGULAR EXPRESSIONS

Having been a student of regular expressions for awhile I have pondered for some time how to cover this subject. Why doesn't my regular expression do what I think it should be doing?" is a common lament. I have said this (truth be told, still do). I am also from the school that believes one does not need to know how a compiler works to be a programmer (though it is not a bad thing to know). However there are times when an understanding of what happens under the hood is instrumental in helping us mere mortals get the job done.

You might think this coming section is not for you. Please read it. I am going to introduce some of the basics of regular expressions (which you definitely need). But, interspersed with this I am going to discuss at a high level what is happening under the hood.

SO HOW DOES A REGEX MATCH?

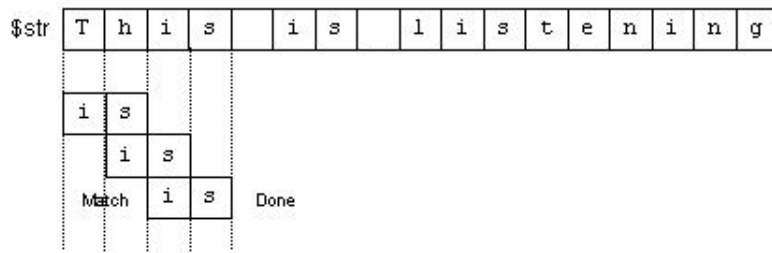
Usually a frightening regular expression is first presented (as I did in an earlier section). I am going to take a different tact and present a simple one. Consider the following code.

```
$str = "This is listening";
$str =~ m/is/
```

You have already been exposed to the matching operator (`=~`) from a previous module so we shall only take a look at the regular expression `"m/is/"`. `/` is used to hold our regular expression and the `"m"` means that we our matching. Since `"m"` is the default we do not have to include it (and it usually is not in most Perl programs). So our regular expression could be written

```
/is/
```

This regular expression as written will match the literal string `"i"` and `"s"`. The question becomes, which `"is"` gets matched in the string `$str`? There are three occurances of the string `"is"`. Now lets look at how the regex engine goes about matching.



The regex engine attempts to match the literal string `"is"` starting at the beginning of the string, moving down the string until it finds a match or has nothing left to match. I chose a literal for our first regular expression as it is easier to see how matching is attempted. Also, it is very easy to think that our regex is actually matching the word `'is'`. In fact, the match occurs on the `"is"` substring in the word `"This"`.

QUANTIFIERS

Quantifiers are metacharacters.

The “?”, “+”, and “*” quantifiers are used in conjunction with characters (or character classes, which will be covered later).

Each of these quantifiers has the following meaning.

?	means that the character preceding is optional, though one is allowed.
*	means match zero or more of the preceding character, though none are required, any amount allowed.
+	means match one or more of the preceding character, one is required, any amount allowed.

Suppose we had a document where we needed to match words that could be spelled using the American or the English convention (example: Flavor or Flavour). We could write the following type of regular expression.

```
Flavou?r
```

This reads, “match all the literal characters “f”, “l”, “a”, “v”, “o”, and may or may not include a “u”, “r”.

```
regex1a.pl
```

Note: *To better focus on the regular expression itself I have chosen to only include what I believe to be relevant code fragments.*

```
$str =~ /flavou?r/)
```

Now we take a look at the “*” and “+” quantifiers. While the two examples presented are not all that useful in a practical sense, they will give us a working literal definition for these two quantifiers.

```
regex1b.pl
```

```
@arr=qw(feel felt fertile feat);
```

```
@arr[$i] =~ /fea*/
```

In the preceding example we will return every element in the array as our condition reads “an “f”, followed by an “e” followed by zero or more of “a”. I promise you as we explore this subject deeper you will get into some practical uses of these. For now I want you to absolutely understand what they do!

Sidebar note: *The use of qw makes it so I do not have to put commas to delineate array members, and do not require quotes around array members.*

```
regex1b2.pl
```

```
@arr=qw(feel felt fertile feat);
```

```
@arr[$i] =~ /fea+/
```

In this example we must match an “f”, followed by an “e” followed by one or more “a”s, hence all we return “feat” from the array

Now, before I trudge merrily along as if all was well in Dodge, I want to point out a subtlety for you that you may not have picked up (actually there are many of these you will find the more you use regular expressions).

Going back to our first example (regex1a.pl) ...

```
$str =~ /flavou?r/)
```

If we changed this to read ...

```
$str =~ /flavou*r/)
```

We would still match both words “flavour” and “flavor”. But, there is a subtle difference you need to be aware of. If we tried to match “flavouur” the regex using “?” would fail, while the one using “*” would succeed. Our criteria for “?” is that the preceding character is optional, up to a max of one. Whereas, “*” also means the preceding character is optional, but any amount is allowed.

These three quantifiers are also considered greedy. We will not be addressing this subject now (I do not feel it is the place), but we will cover this later down the road. For now what this means is that the regex will continue to match until it reaches the maximum number of matches governed by the quantifier.

It is also very important to remember at this point that these quantifiers are only for the character (or collection of characters) preceding it.

CHARACTER CLASS

It is very difficult to cover each element of regular expressions in-vitro. This is because there is such a dynamic relationship between these entities. I recall the first time I read Jeffrey Friedl’s “Mastering Regular Expressions” I thought he seemed to bounce around a bit. Now, in writing this I can see the challenge he faced. There are many subjects in computer science that fit the “chicken and egg” model. This is certainly one of them.

I want to cover here what I believe to be the essentials of the character class. We will then stroll off into some application of these things.

A character class allows us to match any one of a number of characters. Character classes use the [] construct. Look at the following regex.

```
Gre[ea]t
```

This reads, match “G” followed by “r”, followed by “e”, followed by either an “e” or an “a”, followed by a “t”. This expression will match both “Greet” and “Great”. If we wanted to also match “great” and “greet” we can write.

```
[Gg]re[ea]t
```

A character class will only match a single character (by themselves). A character class is a language by itself. I say this as metacharacters have different meaning when used inside a character class. For example, if I wanted to match a number zero to nine, I could write.

```
[0123456789]
```

This is a bit much and can be streamlined by saying.

```
[0-9]
```

Both mean the same thing. If we said 0-9 outside of a character class this would no longer mean zero to nine, but “0” followed by a “-”, followed by a “9”.

We will look at some more of the details of using metacharacters inside a character class later on.

BASIC USE OF PARENTHESIS AND ALTERNATION

Parentheses have a number of uses in regular expressions. Here I am just looking to explore their use with alternation. For example, suppose I wanted to match either “T” or “True” or “Y” or “Yes”. From what you know thus far how might you do that? Using parentheses and alternation we can say.

```
(T|True|Y|Yes)
```

This reads match “T” or “True” or “Y” or “Yes”.

PUTTING SOME OF WHAT WE HAVE COVERED TOGETHER

Now that you have been exposed to some of the basic elements of a regex, lets troll off and see how these might be used in concert with one another. We will also look to explore some of the subtleties of crafting regular expressions. Consider the following list of items.

```
Operand1=Operand2;
Operand1 =Operand2;
Operand1 = Operand2;
Operand1 =      Operand2;
Operand1 = Operand2;
```

If we wrote a Perl statement similar to the ones above we already know that each of these is syntactically correct (though one might question style) for many computer languages. For example, in Perl we can write.

```
$a=$b;
$a = $b;
$a  = $b;
```

...And so on and so forth.

If we were tasked with writing a parser/tokenizer (not that we will even be coming close to doing such a thing), how would we determine that each of these statements were indeed correct and the same? To limit how abstract this discussion will get, we also will say that the operand will be the word "Operand" followed by some number.

```
Operand[0-9]* *= *[Operand[0-9]*;
```

Question: Will this statement match the five items above? _____

Lets take a moment and break apart, piece by piece, the regular expression.

```
Operand[0-9]* *= *Operand[0-9]*;
```

We are looking to match the literal string "Operand". Since this is a literal string match it must appear as written in order to match. In other words, "operand" or "OPERAND" will not fly.

```
Operand[0-9]* *= *[Operand[0-9]*;
```

Immediately following the literal string "Operand" we have a character class. A character class allows us to specify more than one character to match. You can recognize a character class by the opening and closing brackets. The character class as it is written above reads, "any number from zero to nine". The "-" allows us to specify a range. For example, [A-Za-z] would read, "match any character a to z that is either uppercase or lowercase. The "*" following the character class is for the character class and when applied means (just as it would when applied to a single character), "match zero or more of the preceding character (or in this case the character class), none are required, any amount allowed".

Since "*" is for the character class that precedes we could re-state [0-9]* to read, "match zero or more, up to any quantity of any number that is zero to nine".

What this means with respect to our expression is that...

```
Operand  
Operand4  
Operand23238787
```

...would all constitute and match.

Now, if we had said...

```
Operand[0-9]+
```

...we would not match Operand, but would still match the other two in our list because the "+" quantifier requires at least one.

Hopefully you are beginning to see the power of the character class when used with quantifiers (+, ?, *). Next,

```
Operand[0-9]* *= *Operand[0-9]*;
```

We have a space, an asterisk, an equals sign, another space, and an asterisk (sorry I could not figure out how to bold a space :-). Within a regular expression it is important to realize that the space is not whitespace, rather it is referring to a space as a character. So this part of our expression reads, "match zero or more spaces followed by an equals sign, followed by zero or more spaces".

Next,

```
Operand[0-9]* *= *Operand[0-9]*;
```

"Operand[0-9]*" does the same thing as I previously described.

Finally,

```
Operand[0-9]* *= *Operand[0-9]*;
```

We end the expression with a semicolon as a literal character. So, I will re-ask my question. Will this statement match the five items in the list I gave you? Another question. Will the expression as it is written match the two additional items? (the second list item has 2 spaces before the "O")

```
Operand1 = Operand2 ;
  Operand1 = Operand2;
```

Regex regex2.pl will provide you the answer to the first question.

EXERCISE: Re-write "regex2.pl" so that it matches the last two list items.

Time to cover some new turf.

Suppose we wanted to write a regular expression that matches a date that could be expressed in the following formats.

```
11/28/47
11-28-47
```

How would we accomplish this? Why don't you give it a try if you want or read on.

Consider the following regular expression. It contains a few new items. I will only cover the new parts. I will ask you to interpret the stuff that we have already covered.

```
[0-9]* (\/|-) [0-9]* (\/|-) [0-9]*
```

You tell me,

[0-9]* Reads: _____

In our matching expression for dates we are saying, "match a '/' or a '-' ". If you notice, I needed to add the escape character "\" in front of the "/".

regex3.pl

Items we want to try to match with out regex.

```
"11/28/47"      ,
"11-28-47"      ,
"1/2/01"        ,
"-12-98"        ,
"11//54"        ,
"//"            ,
"--"           ,
```

Our regex

```
/[0-9]*(\|/|-)[0-9]*(\|/|-)[0-9]*-/
```

Well, you no doubt saw this coming. Besides matching our two dates we are also matching...

```
"-12-98"      ,
"11//54"      ,
"//"           ,
"--"          ,
```

These are not dates.

EXERCISE: Re-write "regex3.pl" so that it does not match those items.

I want to revisit my first example where we matched "Operand=Operand" and all it's incantations. This served our purpose well for opening a discussion regarding character classes. However from a practical perspective the example was not all that practical. Suppose we were wanting to write a parser/lexer for a language. This example kind of intimated such a project, but our literal use of "Operand" leaves us far short of a desired outcome.

How could we rewrite that regular expression to accept an argument string on either side of our operator? Enter the dot match operator.

The dot operator

If we change the expression from...

```
/ *Operand[0-9]* *= *Operand[0-9]* *;/
```

To...

```
/ *.* *= *.* *;/
```

We will be able to match any string on either side of the operator.

Note: 1) I have enclosed the regular expression inside of the "/" so the spaces would be apparent to you. 2) The regular expression may look slightly different than the one in regex2.pl. This is because the regular expression is the solution to the exercise I gave you after showing you regex2.pl.

The dot operator says to match any character. Used in conjunction with the "+" quantifier our expression reads in English:

“Match one or more occurrences of any character”.

More appropriately stated,

“The dot operator matches any byte except newline.”

If the second, more technically accurate definition does not sit well with you then revert to the first.”

This now makes our example more practical. Of course we still have not accounted for issues like supporting all the available operators, checking our operand for legality, and a whole host of other things one has to do in writing lexical parsers. But hey, this is an introduction course in Perl and my last name is not Knuth.

Oh, and we just introduced you to the dot operator.

regex2_1.pl is a program demonstrating everything we just covered.

Being able to read and articulate a regular expression

I want to stop and point out that I am trying to get you to be able to break down a regular expression and articulate what it does in English, or some other language we humans can understand. I truly believe this is paramount to being able to write regular expressions. When you arrive at the point where you can consciously write these without having to translate them then you will know that you have become one sick puppy. Pressing onward.

Character and Non Character anchors

“\w” represents the character anchor, and “\W” represents the non character anchor.

First lets define what a character is and is not.

\w will match: A-Z, a-z, 0-9 and the underscore

\W will match: anything not included above (example, +, -, %, etc...)

For instance, the following regular expressions will match.

```
A =~ /\w\  
a =~ /\w\  
8 =~ /\w\  
_ =~ /\w\  
  
+ =~ /\W\  
- =~ /\W\  
@ =~ /\W\  

```

\w and \W are powerful metacharactors that can be used in conjunction with quantifiers.

Word and Non Word anchors

“\b” represents the word boundary anchor, and “\B” represents the non word boundary anchor.

We can define a word anchor as any position where the character matches “\w” on one side and “\W” on the other. This is very important to remember. Lets look at three examples and see how well I have explained this.

```
“This was on.” =~ /is/
```

Match? _____

```
“This was on.” =~ /\bis/
```

Match? _____

```
“This was on.” =~ /is\b/
```

Match? _____

```
“This was on.” =~ /\bis\b/
```

Match? _____

The answers are match, no match, match, no match. In the first regex we will match any time there is an occurrence of the string “is” regardless of where it occurs. The second regex fails to match because we are asking it to match any occurrence of the string “is” that begins a word boundary. This string would match the word “is” and any words that begin with “is” (isn’t, isocoles). The third regex matches. It will also match the word “is” or any other word that ends in “is”. Finally, the last match fails. This regex will match the word “is” or any instance of the string “is” where a non-character anchor is on either side. For instance, +is+ will match.

I do not know if you caught it earlier or not, as it is a very subtle point, but one that you need to understand. I said earlier that our second regex (\bis) would match “isn’t”. While indeed it will, it is only really matching “isn”. Remember, we match on a word boundary up until a non character character. The “” represents a non character, so we never get to the “t”. This is precisely why we could match a hyphenated word like “non-fat” with either the regex “\bfat” or “fat\b”.

Position at start or at end of line

There may be times when we only want to search within the beginning or end of a line. Certainly this is a means of making your search for efficient and it can also prevent from undesired matches (or non matches).

The `^` is used to match at the beginning of a line. The `$` is used to match at the end of a line. Note that the carrot has a different meaning when used within a character class. You will find a script that demonstrates use of the examples below. It is named `eolbol.pl`.

```
/^Subject/
```

This regex will match the pattern “Subject is topic” but will not match the pattern “Topic is Subject”.

```
/Subject$/
```

This regex will match the pattern “Topic is Subject” but will not match the pattern “Subject is topic”.

Negated character class

As you have already seen, we can write an expression `/A-Za-z/` that will match any character `a-z` in either upper or lowercase. But how can we exclude characters? We do so by using the `^`.

For example, given the following array elements, suppose we want to only return the items that do not contain numbers.

```
regex5.pl
```

```
"HELLO"      ,
"57"         ,
"Goodbye"    ,
"Turk82"     ,
"82Turk"     ,
"Whatever"   ,
"xyz12xyz"   ,
```

```
/\b[^\d]+\b/ )
```

The code above will return the following members of the array.

```
HELLO
Goodbye
Whatever
```

GREED

Besides being a short lived game show, this is a subject related to quantifiers. I choose to cover this subject now, removed some distance from the quantifiers as I find this topic to be very abstract and I did not want to “confuse the issue with facts” at the time that I was initially discussing quantifiers.

The quantifiers `+`, `*`, and `?` are greedy (there are other greedy quantifiers as well, though I have not covered these in this text).

Remember back to the beginning of this module (those of you that are still coherent). I stated a fundamental truth about regular expressions. That is that the **first match wins**. Well there is another

fundamental truth. It is, a **quantifier settles for the minimum matches allowed, but will continue matching as many times as it can**. Consider the following.

```
\b[aeiou]+[a-z]*
```

We are looking to match one or more occurrences of “a” or “e” or “i” or “o” or “u” (our character class).

This as applied to the word “earlier” would successful match on the “e”. But since + is a greedy little bugger that says one or more, with respect to our character class it will also match the “a” and the “ie” as well. In fact, one important difference between the way I wrote it, or writing it `\b[aeiou][a-z]*` is on when the engine with finish the match. In the way I just restated the regex, it will match the first character class up until the point that it encounters the first vowel.

We must keep in mind that the maximum number of matches is with respect to the entire regular expression. For example, suppose I wanted to match all words ending in “ing”. I could write something like.

```
\b\w+ing\b
```

This regex would match all words ending in “ing” but as it is written would not match words like “mink” or “ingot”.

Oh yeah, you were probably thinking you got off “scot free” here. No, no, no. I want you to state in English the regular expression.

```
\b[aeiou]+[a-z]*
```

Reads: _____

A practical application of greed

You have already seen some practical uses of greed when we covered the quantifiers earlier in this module. There are some more very practical applications that are not nearly as apparent. I am in the business of education. Each year we publish a course catalog. This catalog contains course descriptions. These descriptions follow the standard layout.

Course
Length
Who Should Attend
Prerequisites
Purpose of the Course
What You Will Learn

Suppose I wanted to produce a list of course names from the catalog. Further, I want the list to appear like...

Foundations in Perl Programming - for experienced programmers
Foundations in OOP Programming using Java - for experienced programmers

Instead of...

Course: Foundations in Perl Programming - for experienced programmers

Course: Foundations in OOP Programming using Java - for experienced programmers

To solve this problem we could write the following regex.

```
/^Course: (.*)/
```

Before I explain this regex. Examine and run the program below.

re1.pl

```
my $str="Course: Foundations of Perl Programming - for experienced programmers";
print ("$str\n");

if ($str =~ /^Course: (.*)/)
{
    print $1;
}
```

This program will print “Foundations of Perl Programming - for experienced programmers”.

We know that the heading “Course:” is standard. And since it is possible for the word “Course:” to appear elsewhere in the text we craft our regex to match at the beginning of the line.

```
^Course:
```

Next we use the dot operator followed by the * quantifier. This reads match zero or more of any character, as many as you can. Also notice that we placed this in parentheses. This is what is termed “parenthesis memory”. By wrapping “.” in parentheses the pattern matched gets stored in a variable \$1 that we have access to. I don’t know about you, but I think this is really cool. To further clarify what I am talking about look at and run re2.pl.

re2.pl

```
my $str="Course: Foundations of Perl Programming - for experienced programmers";

if ($str =~ /(^Course:) (.*)/)
{
    print $1;
    print "\n";
    print $2;
}
```

For this script the value of \$1 is “Course:” and the value of \$2 is “Foundations of Perl Programming - for experienced programmers”.

Gimme back my bullet (with apologies to Lynyrd Skynyrd)

This is a neat subject not only for what you can accomplish, but also because using parentheses can serve as a sort of debugging tool for your regular expressions. For example, in the regex

```
\ (.*) (.*) \
```

Who gets the data? (Note: code for the following regex examples covered can be found in re3.pl)

It so happens that \$1 has the data and \$2 is empty. This is because \$1 has gobbled everything up and is not willing to give it up to the second "(.*)". Does this mean that once we use this we can no longer match anything else? Not at all. Consider the following string applied to the following regular expression.

```
Course: Perl Programming 201 - for experienced programmers
```

```
(.*) ([0-9][0-9][0-9])
```

In \$1 we will find "Course: Perl Programming ". \$2 will contain "201". What happens is that initially, since "(*)" is the ultimate greed monger, it gobbles everything. Then it is forced to give up something in order to match "[0-9]". It gives up the "s" in programmers. Since this does not match it is then forced to give up the "r". I am quite sure about this time it is getting a little irritated at having to give away it's stash. You know, this is how greedy and selfish things are. Little does it know at this point that it will be giving it up until the "1". Of course it is not done as we hit the second [0-9] which forces it to give up the "0", and so on and so forth.

The rest of our string (" – for experienced programmers") at this time is unavailable based on how we wrote our regex. However, we could get it by saying.

```
(.*) ([0-9][0-9][0-9]) (.*)
```

Now we have \$3 that holds " – for experienced programmers".

Just when you were starting to feel good (when greed follows greed)

The script re3a.pl is discussed here.

Consider the following string and the three regex's.

```
Course: Perl Programming 201 - for experienced programmers
```

```
(.*) ([0-9]+)  
(.*) ([0-9]*)  
(.*) ([0-9]?)
```

How might you think these three will match? What would you expect to find in \$2 for each of these? Chances are your guess is wrong unless you are already well versed in regular expressions (at which point I might question why you are reading this).

For (.*)([0-9]+), \$2 holds "1". For the other two \$2 is empty. It would appear from this that their might be some higher form of intelligent life on the planet regex. Now this logic may appear bizarre. It may even offend your sense of justice and good sportmanship. I know that my eyebrow raised when I first saw it. I actually even considered burying my head in the sand and ignoring this. But alas, as an educator


```

if ($str =~ /(z*)/)
{
    print("\$1 is $1\n");
}

print("\nRegex: (z(z(z(z(z?)?)?)?)?)\n");

if ($str =~ /(z(z(z(z(z?)?)?)?)?)?)/)
{
    print("\$1 is $1\n");
    print("\$2 is $2\n");
    print("\$3 is $3\n");
    print("\$4 is $4\n");
    print("\$5 is $5\n");
}
    
```

```

D:\regex>perl -w re4a.pl
Regex: (.*)
$1 is zz

Regex: (z(z(z(z(z?)?)?)?)?)
$1 is zz
$2 is z
$3 is
$4 is
$5 is
    
```

As you can see by the output (which reflects the saved state), we match up to the y. This becomes our first saved state. The “y” fails to match and we backtrack to “zz”. Since none are required, one is allowed, and we have two the engine can conclude the match. The last “z” never gets looked at.

If you are still scratching your head, play with this code and modify \$str to be...

```

zzyzzzz
yzzz
    
```

...See what results you get.

MAKING QUANTIFIERS LESS GREEDY

You may have gathered already that greedy quantifiers can do an awful lot of work to match. There will no doubt be times when you want to turn greed off. In other words, you want to use the quantifiers but you want them to stop after they have found a match.

*?	– Matches zero or more times
+?	– Matches one or more times
??	– Matches zero or one time

<code>{n}?</code>	– Matches n times
<code>{n,}?</code>	– Matches at least n times
<code>{n,m}?</code>	– Matches at least n times but not more than m times

MORE GOODIES

I think by now you should have the beginning of a good foundation. Of course, the concrete may not be fully hardened. This will take time. Here we will cover a poupourri of other items (in no particular order) to round out your knowledge of regular expressions.

Does not match operator

```
<what to match goes here> !~ /<regex goes here>/
```

`!~` is the does not match operator. You can use it in place of the matches (`=~`) operator.

Substitution operator

Up until now I have shown you the modify operator `m/ /`. The `m` can be assumed when writing a regex. Here is the syntax for the substitution operator and an example.

```
s///
```

Assume the string “This is it, is it not?”. We want to change each occurrence of “is” with “was”. Examine the following regex.

```
s/\bis\b/was/
```

We need to be sensitive to word boundaries or we could also end up changing “This” to “Thwas”. If we run this our string would now be, “This was it, is it not?”. In order to change each occurrence of “is” we use the global modifier.

```
s/\bis\b/was/g
```

Making your regex case insensitive

```
/<some reg ex goes here>/i
```

Just add an “i” after your regex to make it case insensitive. Keep in mind this adds a lot more work for the engine. Only use this where you truly need to make your regex case insensitive or where it is impractical to localize case.

```
/hello/i
```

Will match

```
Hello
```

```
hello  
HELLO  
Hello
```

Or, any other combination of case.

Compiling your regex

When using regular expressions in a loop, for each iteration of the loop, your regex needs to be compiled. This is certainly necessary if you are using variables in your regex. But, if you are not you can use the “o” modifier. This will compile your regex just one time.

```
/<some reg ex goes here>/o
```

Some Regexercises (pun intended)

The following exercises will put to the test what you have learned. In doing these you will need to write more than just a regular expression to solve the problem.

Ideas...

Finding duplicate words

Trimming leading whitespace

Matching multiple lines

Extracting a substring

Changing a word

USEFUL REGEX REFERENCES

Pattern Matching

Operator	
<code>=~</code>	Matches
<code>!~</code>	Does not match

Type	Syntax	Example	
Matching	<code>m//</code> Or <code>//</code>	<code>"Hello" =~ /^[A-Za-z]+/</code>	Will match
Substitution	<code>s///</code>	<code>s/Hello/Goodbye/</code>	Replaces first "Hello" matched with "Goodbye".
Translation	<code>tr///</code>	<code>tr/A-Z/a-z/</code>	This operator is used to make text translations. It will replace characters found in the first list with characters found in the second. In the example, it will change case from upper to lower

Atoms

Symbol	Description
<code>.</code>	Any one character
<code>[]</code>	Character class
<code>[^]</code>	Negated character class
<code>()</code>	Used to limit scope. Used with alternation character. Can capture text via parenthesis memory.
<code> </code>	Or logic. Used in conjunction with <code>()</code> .

Special Characters

\077	Octal char
\a	Alarm/Bell
\c[Control Char
\d	Match a digit
\D	Match a non-digit
\E	End case modification
\e	Escape
\f	Form Feed
\l	Next character lowercase
\L	Lowercase until \E is encountered
\n	Newline
\Q	Disables pattern metacharacters until \E is encountered
\r	Return
\S	Match a non-whitespace character
\s	Match a whitespace character
\t	Tab
\u	Next character uppercase
\U	Uppercase until \E is encountered
\w	Match a word character (alphanumeric and underscore)
\W	Match a non-word character
\x1	Hex character

Quantifiers

Greedy

Symbol	
*	Match zero or more of the preceding character, though none are required, any amount allowed.
+	Match one or more of the preceding character, one is required, any amount allowed.
?	Character preceding is optional, though one is allowed.
{n}	Match n times.
{n,}	Match at least n times.
{n,m}	Match at least n times but not more than m times.

Non-Greedy

Symbol
*?
+?
??
{n}?
{n,}?
{n,m}?

Assertions (Anchors)

Symbol	Description
^	Matches at the beginning of the line.
\$	Matches at the end of the line.
\b	Match a word boundary.
\B	Match a non-word boundary.
\A	Matches only at the beginning of a string.
\Z	Match only at the end of a string, or before a newline at the end.
\z	Match only at the end of a string.

Modifiers for m// and s//

Symbol	Description
i	Ignore case
x	Ignore whitespace in pattern
g	Global, performs all possible operations
gc	Does not reset the search position for all possible operations
s	Lets the . character match newlines
m	Lets ^ and \$ match embedded \n characters
o	Compiles the pattern only once
e	Indicates the right hand side of s/// is code to evaluate
ee	Indicates the right hand side of s/// is a string to evaluate and run as code; then evaluates its return value again

FINAL OBSERVATION

While we spent twenty some pages we have only nicked the subject. You may feel that your head is ready to explode. If this is the case do not feel bad. This is a subject that takes a great deal of time to really understand and a lifetime to master. You will only get good at this subject through trial and error. Hopefully I have given you some of the essentials for you to get started on this journey.

There are many ways to write a regular expression that does the same thing. I have run across people that have looked at my regular expressions and said, "why didn't you write it this way, it would have been easier." I smile and think to myself "easier for you maybe" and generally reply "because, I do not think that way". I do not want to impose my way of writing expressions on you. You need to write them in the way that you think. I try to be open to new ideas and techniques, but in the end I must operate in a system of logic that works best for me. I would encourage you to do the same. As Larry Wall said, *"One of the characteristics of a postmodern computer language is that it puts the focus not so much onto the problem to be solved, but rather onto the person trying to solve the problem."* And, *"True greatness is measured by how much freedom you give to others, not by how much you can coerce others to do what you want."* Amen, brother Larry!

If you are looking for further study on this subject I would recommend getting the book Mastering Regular Expressions by Jeffrey Friedl. I have also run across some good documents and tutorials out on the internet. Check out www.perl.com.

HASHES

A hash is an associative array. An associative array is an array where the index value is meaningful to a human being. Let us look at a conventional array and contrast it with an associative array (or hash as I will refer to it as from here on).

Consider the following array.

```
@arr = (298, 114, 31, 12, 2, 67, 55);
```

The elements of this array are as follows.

```
@arr[0] = 298;  
@arr[1] = 114;  
@arr[2] = 31;  
@arr[3] = 12;  
@arr[4] = 2;  
@arr[5] = 67;  
@arr[6] = 55;
```

It is a stretch (to say the least) to know what item four in the array represents. When we are just using an array to stash a collection of like variables this is not a problem. But when the data we are storing is not self-described and when it becomes important to know what the data is about, an array can be limiting.

For example, suppose that the data above represents a baseball player's statistics. It would become more important to understand what location `@arr[3]` represented. Enter the hash.

```
my %batter_stats =  
(  
  'Batting Average' = 298;  
  'Hits' = 114;  
  'Homeruns' = 31;  
  'Double' = 12;  
  'Triples' = 2;  
  'RBI' = 67;  
  'Runs' = 55;  
)
```

A hash allows us to index values in an array in a meaningful way. This concept is similar to the concept of a column/field in database management systems, and how tables are created and stored in a catalog or data dictionary. If you think long and hard you will see how hashes can open up the world of creating and working with structured data in your Perl programs.

CREATING HASHES

As with most things in Perl there is more than one way to create a hash. These two ways are presented for your consideration.

Program: hash1.pl

```
use strict;

# create an empty hash
my %hsh = ();

#add some elements
$hsh{'firstname'} = 'John';
$hsh{'middle_init'} = 'H.';
$hsh{'lastname'} = 'Smith';

print $hsh{lastname};
```

Program: hash2.pl

```
use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

print $hsh{lastname};
```

ADDING AN ELEMENT TO A HASH

Program: hash3.pl

```
use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

my $title = "Supreme Commander";

$hsh{title} = $title;
```

```
print $hsh{firstname};
print " ";
print $hsh{middle_init};
print " ";
print $hsh{lastname};
print ", ";
print $hsh{title};
```

Note that you can assign variables to a hash element. As you can see adding a value to a hash is a pretty easy undertaking.

DELETING A HASH ELEMENT

Program: hash4.pl

```
use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

delete $hsh{middle_init};

if (exists($hsh{middle_init}))
{
  print "key exists";
}
else
{
  print "key does not exist";
}
```

GET HASH KEYS

Program: hash5.pl

```
use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

my $key="";
foreach $key (keys %hsh)
{
  print $key . "\n";
}
```

```
}
```

GET HASH VALUES

Program: hash6.pl

```
use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

my $val="";
foreach $val (values %hsh)
{
  print $val . "\n";
}
```

GET BOTH HASH KEYS AND VALUES

Program: hash7.pl

```
use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

my $key="";
my $val="";
while (($key, $val) = each(%hsh))
{
  print "$key => $val\n";
}
```

FINDING A PARTICULAR HASH KEY

This is actually pretty neat. You can find a hash key without having to code a loop yourself.

Program: hash8.pl

```
use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

# exists is used to verify key
if (exists($hsh{middle_init}))
{
  print $hsh{middle_init};
  print "\n"
}
else
{
  print "key does not exist";
}

# delete hash key middle_init
delete $hsh{middle_init};

# is it deleted?
if (exists($hsh{middle_init}))
{
  print $hsh{middle_init};
  print "\n"
}
else
{
  print "key does not exist";
}

# re-add hash key middle_init
$hsh{middle_init} = "";

if (exists($hsh{middle_init}))
{
  print $hsh{middle_init};
  print "\n"
}
else
{
  print "key does not exist";
}
```

REVERSING VALUES AND KEYS IN A HASH

This is quite an interesting little function that allows you to make your values the keys and your keys the values. This certainly could be used in defining meta data schemas. Since this is just the essentials we will not be offering anything more than some toy code.

Program: hash9.pl

```

use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

my %swap = reverse %hsh;

my $key="";
foreach $key (keys %swap)
{
  print "$key => $swap{$key}" . "\n";
}

```

SORTING A HASH

A hash can be sorted by key or by value.

By Key

Program: hash10.pl

```

use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',
  lastname => 'Smith',
);

my $key="";
foreach $key (sort keys %hsh)
{
  print $key . "\n";
}

```

By Value

Program: hash11.pl

```

use strict;

my %hsh =
(
  firstname => 'John',
  middle_init => 'H.',

```

```
lastname => 'Smith',
);

my $val="";
foreach $val (sort values %hsh)
{
    print $val . "\n";
}
```

MERGING TWO HASHES

Program: hash12.pl

```
use strict;

my %person =
(
    firstname => 'John',
    middle_init => 'H.',
    lastname => 'Smith',
);

my %phone =
(
    area_cde => '206',
    exchange => '466',
    extension => '1515',
);

my %person_phone = (%person, %phone);

my $key="";
my $val="";
while (($key, $val) = each(%person_phone))
{
    print "$key => $val\n";
}
```

BASIC DEBUGGING

DEBUGGING

You should have a pretty good working knowledge of Perl by this point. The subject taught now should serve to enrich your understanding of Perl and help you in your endeavors. Also keep in mind that I am not doing exhaustive coverage of this subject. You should however get enough from this section to get you up and running.

Please note that I am using the program if-else.pl for these examples except where noted.

STARTING THE DEBUGGER

Starting the debugger is easy. Just run your script the way you normally would, except, add the `-d` argument.

SYNTAX

```
> perl -d <<scriptname>>
```

EXAMPLES

```
> perl -d if-else.pl
```

```
Loading DB routines from $RCSfile: perl5db.pl,v $$Revision: 4.1 $$Date: 92/08/07
18:24:07 $
Emacs support available.
Enter h for help.
main::(if-else.pl:2):   my $x="Hello";
DB<1>
```

Once the debugger is invoked you will get the debugger prompt "DB<1>". Now lets look at the basic things you can do in the debugger.

LIST SOURCE CODE

Use the dash (-) to list your source code.

```
DB<1> -
1:      use strict;
2:      my $x="Hello";
3:      my $y="Goodbye";
4:
5:      if ($x ne "Hello") {
6:          $y=$x;
7:          print ($y);
8:      }
9:          else {
10:             $x=$y;
```

```
11:             print ($y);
12:             }
13:
14:
15:
DB<1>
```

If you have a long script you can use the lines (l) command in lieu of this command. It is covered below.

STEPPING

```
DB<2> s
main::(if-else.pl:3):   my $y="Goodbye";
```

You can use the step command in conjunction with the command below to list code.

Execute next statement

```
DB<2> n
```

Note: This command does not step into subroutines.

LISTING LINES OF CODE

Here are some various examples of this command.

Display next window of code

```
DB<1> l
2:   my $x="Hello";
3:   my $y="Goodbye";
4:
5:   if ($x ne "Hello") {
6:       $y=$x;
7:       print ($y);
8:   }
9:       else {
10:          $x=$y;
11:          print ($y);
DB<1>
```

Display n lines from line#

SYNTAX

```
> l <<start line#>> + <<#lines to display after start line>>
```

EXAMPLE

```
DB<1> l 5+2
5:      if ($x ne "Hello") {
6:          $y=$x;
7:          print ($y);
DB<2>
```

Display lines n through n

SYNTAX

```
> l <<from line#>> - to line#>>
```

EXAMPLE

```
DB<2> l 3-7
3:      my $y="Goodbye";
4:
5:      if ($x ne "Hello") {
6:          $y=$x;
7:          print ($y);
DB<3>
```

Display a specific line#

SYNTAX

```
> l << line#>>
```

EXAMPLE

```
DB<3> l 10
10:          $x=$y;
```

Display lines around the current line

```
DB<8> w 10
7:          print ($y);
8:      }
9:          else {
10:             $x=$y;
11:             print ($y);
12:          }
```

```
13:  
14:  
15:  
  DB<9>
```

VARIABLES

Display variables

Note: If you display a variable before it has been initialized or set to spaces you will not see anything.

SYNTAX

```
> p <<variable_or_array>>
```

EXAMPLE

```
  DB<9> p $x  
Hello
```

You can also display more than one variable at a time.

```
  DB<10> p $x,$y  
HelloGoodbye
```

Modify variables

```
DB<29> p $y  
Goodbye
```

SYNTAX

```
> <<variable>> = <<new value>>
```

EXAMPLE

```
//Modify variable  
DB<30> $y = "Later"
```

```
DB<31> p $y  
Later
```

COMMANDS

Display last n commands

SYNTAX

```
> H -<<#lines>>
```

EXAMPLE

```
DB<4> H -5
4: H -5
3: p $y
2: p $x
1: l 1-5
```

Recall a command

SYNTAX

```
> !<<line#>>
```

EXAMPLE

```
DB<5> !3
p $y
Goodbye
```

BREAKPOINTS

I will show you how to break at both line numbers and subroutines.

On line#

SYNTAX

```
> b <<line#>>
```

EXAMPLE

```
DB<1> b 10
```

On Subroutine

Note I am using sub1.pl to cover this topic.

SYNTAX

```
> b <<subroutine name#>> line>>
```

EXAMPLE

```
DB<1> b mysub2
```

RESUME

Continue to next breakpoint or end of program

```
DB<2> c
```

RETURN FROM A SUBROUTINE

This command will return from a subroutine that you have broken on.

Note: If we use "c" instead of "r" we will not return to the point in the program where we set the breakpoint.

```
DB<2> r
```

DELETE ALL BREAKPOINTS

```
DB<9> D
```

WRITING CODE WHILE YOU ARE DEBUGGING

This is a very interesting and useful feature of the debugger. You can write and execute lines of code while you are involved in the debugging process.

Note: In this case I have created a new variable while debugging if-else.pl. I then created an expression to assign \$y to prior to printing \$y.

```
DB<3> $foo = "See ya later";
DB<6> s
main::(if-else.pl:11):          print ($y);
DB<6> $y = $foo; //assign $y to the variable we just declared
DB<7> c
See ya later
```

EXIT DEBUGGER

```
DB<9> q
```

TRACING A CALL

This example shows the trace command (T) and the events leading up to utilize it. I am using sub1.pl for this example.

```
//set breakpoint on subroutine
DB<2> b mysub2

//resume to breakpoint
DB<3> c
123
main::mysub2(sub1.pl:24):          my ($a,$b,$c) = @_;

//now that we are at the breakpoint we can use the trace command
DB<3> T
$ = main::mysub2(1, 2, 3) from file sub1.pl line 9
//Note that it tells us where the call came from. In this case from line 9 of the same program
```

WATCHING A VARIABLE THROUGHOUT PROGRAM EXECUTION

This is a specific use of a Perl command that can be setup for use within a debugging session.

Let's say that you want to see the state of a variable through the course of your program. You can setup a print statement and use "<" which will execute your Perl code before every debugger prompt.

```
DB<1> < print "\$b = $b\n";
DB<2> b mysub2
DB<3> b mysub3
DB<4> s
main::(sub1.pl:4):          my $b=2;
$b =
DB<4> c
123
main::mysub2(sub1.pl:24):          my ($a,$b,$c) = @_;
$b = 2
DB<4> c
Etc...
```

MISC STUFF

These are some odd and ends that I did not know where else to put.

ENTERING VALUES IN HEX

PROGRAM: hex.pl

BIT MANIPULATION

PROGRAM: bitmanip.pl

RANDOM NUMBERS

PROGRAM: rand-num.pl

TRUTH TABLES

PROGRAM: truthtables.pl

TIME

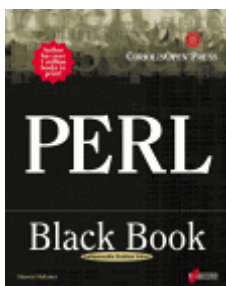
PROGRAM: time.pl

APPENDIX A – RECOMMENDED READING AND SITES

READING

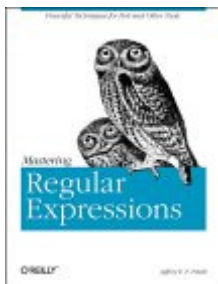
Perl Black Book by Steven Holzner

This is my first book on Perl. This book is centered around solving real problems. It offers a wealth of toy code that works and does a great job of clarifying concepts and ideas. I do find his occupational scenarios (his attempt at humor) a little silly, but this is a minor complaint.



Mastering Regular Expressions by Jeffrey E. Freidl

This is a must have book on a subject that is a language in itself. Jeff does a great job of addressing this subject. He covers regular expressions as implemented in a number of tool (egrep, etc...) as well as Perl. I also think it helps to have worked a little bit already with regular expressions prior to approaching this book. One note: The information is a bit dated, but if you are looking for the nuts and bolts minutia this is the place.



Programming Perl by Larry Wall, Tom Christiansen, Jon Orwant

This book is written by the inventor of the language, and one of it's most regarded authorities. Many programmers like the O'Reilly books, and many Perl devotees claim this is a must have for their library. I like the book a great deal as it real reflects well on the philosophy of the language, provides great historical (and other) context and is written in a very lighthearted style.



WEBSITES

This is the definitive Perl site.
<http://www.perl.com/>

This is another decent Perl site.
<http://www.activestate.com/>

I would also encourage you to read anything you can on Larry Wall (interviews, essays). He is a very interesting individual. Also spend a few hours perusing www.perl.com. There is a wealth of excellent information out there!

APPENDIX B – USEFUL REFERENCE INFORMATION

OPERATORS

There are a number of operators that you can use to work with variables. Run and examine the following programs. They will give you a good introduction to operators in Perl. Note that the examples show use of these operators with scalars.

Arithmetic

*	Multiplication
/	Division
+	Addition
-	Subtraction
**	Exponentiation
%	Remainder

PROGRAM: miscoper.pl

Logical

&&	And	\$a && \$b
	Or	\$a \$b
!	Not	!\$a

PROGRAM: and-or.pl

Assignment

Symbol	Description	Example	Equivalent
=	Assignment	\$a = 5	n/a
+=	Addition and assignment	\$a += 5	\$a = \$a + 5
--=	Subtraction and assignment	\$a -= 5	\$a = \$a - 5
*=	Multiplication and assignment	\$a *= 5	\$a = \$a * 5
/=	Division and assignment	\$a /= 5	\$a = \$a / 5
%=	Remainder and assignment	\$a %= 5	\$a = \$a % 5
**=	Exponentiation and assignment	\$a **= 5	\$a = \$a ** 5
&=	Bitwise AND and assignment	\$a &= 5	\$a = \$a & 5
=	Bitwise OR and assignment	\$a = 5	\$a = \$a 5
^=	Bitwise Exclusive OR and assignment	\$a ^= 5	\$a = \$a ^ 5
.=	Concatenation and assignment	\$a = "can" \$a .= "not"print	

		(\$a)results in"cannot"	
--	--	-------------------------	--

PROGRAM: oper-assign.pl
 PROGRAM: assignmentequals.pl

Backslash

\cn	CTRL+n charactor
\e	Escape
\E	Ends the effect of \L, \U, \Q
\f	Form feed
\l	Forces next letter to lowercase
\L	Forces all subsequent letters to lowercase
\n	Newline
\r	Carriage return
\Q	Do not look for special pattern charactors
\t	Tab
\u	Forces next letter to uppercise
\U	Forces all subsequent letters to uppercise
\v	Vertical tab

PROGRAM: case.pl

Match

=~	Binds a pattern to a string to see if pattern is matched. Used with regular expressions.	\$a = \$y =~ /^[a-zA-Z]/
=!	Binds a pattern to a string to see if pattern is not matched. Used with regular expressions.	\$a = \$y != /^[a-zA-Z]/

String Comparison

I always loathed the operators below when using them in various report writers over the years. I still run into difficulty remembering to use these instead of the traditional operators (<, <=, ==, etc...).

lt	Less than
gt	Greater than
le	Less than or equal
ge	Greater than or equal
ne	Not equal
cmp	Compare - returns 1,0,-1

PROGRAM: oper-stringcompare.pl

Integer Comparison

<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equal
!=	Not equal
<=>	Compare - returns 1,0,-1

PROGRAM: oper-integercompare.pl

REGULAR EXPRESSIONS

Greedy Quantifiers

?	means that the character preceding is optional, though one is allowed.
*	means match zero or more of the preceding character, though none are required, any amount allowed.
+	means match one or more of the preceding character, one is required, any amount allowed.
{n}	Match n times.
{n,}	Match at least n times.
{n,m}	Match at least n times but not more than m times.

Non-Greedy Quantifiers

*?	– Matches zero or more times
+?	– Matches one or more times
??	– Matches zero or one time
{n}?	– Matches n times
{n,}?	– Matches at least n times
{n,m}?	– Matches at least n times but not more than m times

Pattern Matching

Operator	
=~	Matches
!~	Does not match

Type	Syntax	For instance	
Matching	m// Or //	“Hello” =~ /^[A-Za-z]+/	Will match
Substitution	s///	s/Hello/Goodbye/	Replaces first “Hello” matched with “Goodbye”.
Translation	tr///	tr/A-Z/a-z/	This operator is used to make text translations. It will replace characters found in the first list with characters found in the second. In the example, it will change case from upper to lower

Atoms

Symbol	Description
.	Any one character
[]	Character class
[^]	Negated character class
()	Used to limit scope. Used with alternation character. Can capture text via parenthesis memory.
	Or logic. Used in conjunction with ().

Special Characters

\077	Octal char
\a	Alarm/Bell
\c[Control Char
\d	Match a digit
\D	Match a non-digit
\E	End case modification
\e	Escape
\f	Form Feed
\l	Next character lowercase
\L	Lowercase until \E is encountered
\n	Newline
\Q	Disables pattern metacharacters until \E is encountered
\r	Return
\S	Match a non-whitespace character
\s	Match a whitespace character
\t	Tab
\u	Next character uppercase
\U	Uppercase until \E is encountered
\w	Match a word character (alphanumeric and underscore)

\W	Match a non-word character
\x1	Hex character

Assertions (Anchors)

Symbol	Description
^	Matches at the beginning of the line.
\$	Matches at the end of the line.
\b	Match a word boundary.
\B	Match a non-word boundary.
\A	Matches only at the beginning of a string.
\Z	Match only at the end of a string, or before a newline at the end.
\z	Match only at the end of a string.

Modifiers for m// and s//

Symbol	Description
I	Ignore case
X	Ignore whitespace in pattern
G	Global, performs all possible operations
gc	Does not reset the search position for all possible operations
S	Lets the . character match newlines
M	Lets ^ and \$ match embedded \n characters
O	Compiles the pattern only once
E	Indicates the right hand side of s/// is code to evaluate
ee	Indicates the right hand side of s/// is a string to evaluate and run as code; then evaluates its return value again

DEBUGGING

Action	Syntax	Example
Start the debugger	>perl -d <script>	>perl -d hello.pl
Stepping		s
Execute next statement		n
Listing lines of code		l
	l <from> - <to>	l 1-5
	l <line#>	l 23
Display lines around current line		w 10
Display variables		p \$x
Modify variables		\$x = "new string"
Display last <i>n</i> commands	h -<#lines>	h -5
Recall a command	!<line#>	!4
Break on line	b <line#>	b 12
Break on subroutine	b <subroutine name>	b mysub
Return from a subroutine		r
Delete all breakpoints		d
Watching a variable	< print <variable(s)>	< print "\\$x = \$x\n";
Exit debugger		q