Introduction to Perl

by Nancy Blachman
Variable Symbols, Inc.

This workshop provides a practical introduction to Perl with examples of usage. You will learn how to manipulate text, process data, calculate, program, and write CGI scripts in Perl.

Perl ("Practical Extraction and Report Language") is a multi-purpose programming language. It was initially designed for doing text manipulation and report formatting, but it has become increasingly popular for Web-based applications. Many server-based applications, which use the Common Gateway Interface (CGI), are written in Perl, because of Perl's flexibility and portability.

Perl is an easy programming language to learn: beginning Perl programmers can start small, using a simple subset of the language. Since Perl is an interpreted language, small Perl programs can be written and tested rapidly—this makes it easy to experiment with small modifications to existing Perl programs. There are also a large number of standard Perl library modules that help programmers do many tasks.

After this course you will be able to write and execute a Perl script. This workshop includes the following topics:

Prerequisites: This course is for people with no previous experience with Perl, but some experience programming. Because Perl includes aspects from Awk, C, Shell, and Unix, participants who have experience with them will probably pick up Perl more rapidly than others.

1 What this Course Will Cover

Although I designed this course for people new to Perl, it contains information of interest to those who have experience with the program. I've sprinkled quizes and exercises throughout these course notes to test your understanding of the material presented. I strongly urge you to work through the examples and the problem sets.

Though I developed this material on a Micron Pentium running Redhat Linux 5.2, this workshop illustrates general techniques that can be used with other types of computers and operating systems. Perl is available on all sorts of computers including PCs, workstations, and Macintoshes.

This course doesn't cover

- How to install Perl.
- How Perl is different under different operating systems.

1.1 About the Instructor

In 1990 I produced the *Mathematica Quick Reference* by constructing an ascii database and using awk[1] to grab information from the database and append formatting commands in LaTeX[2] so that I could typeset the book. Around that time, I took a Perl workshop where I learned of Perl's versatility. In 1999, I wrote risk analysis programs, and financial reports for a firm on Wall Street where I was a mathematical programmer. At VA Linux Systems, I use Perl to write CGI scripts to produce forms on the web and I write Perl scripts to summarize the results.

2 Why Use Perl?

- Huge and growing library for Perl resources
- Great for manipulating text and generating reports
- Interpreted—short development time
- Easier to learn than C, Java, and other compiled languages
- Portable across platforms
- Free

Here are draw backs to using Perl.

- Slower than compiled languages
- Has some weird constructs

Perl was created by Larry Wall in 1987 to help him do system administration and generate reports. He borrowed features from

- awk
- Shell
- Unix
- C

3 Getting Started with Perl

In these notes, any input commands appear in `Courier`. Output appears in *`italic Courier`*. Usually when I mention a function, I write its name followed by parentheses, e.g., `print()`. In these notes I listed several Perl programs, which I will sometimes refer to as scripts. Following a program, I often include the output it generates, but I don't usually include the statement I used to invoke the program.

[1] Awk is a pattern scanning and processing language that is available on Unix systems.
[2] LaTeX is a language for typesetting text. These course notes were prepared using LaTeX.

3.1 Writing a Perl Program or Script

In a file named `hello`, enter the follow Perl commands with any text editor. Save the file as text (ascii).

```
print "Hello class!\n";
```

Remember to include the semicolon (`;`) at the end of the line. Nearly every Perl statement ends with a semicolon. That's how Perl can tell where one statement ends and the next begins. So if you leave out a semicolon, Perl may complain or interpret your program in a different way than you intended.

Run the Perl script by typing

```
perl hello
```

If you want the Perl script `hello` to be executable directly from the Unix command line, so you can type `hello` instead of `perl hello`, add a line to the beginning of your program

```
#!/usr/bin/perl
print "Hello class!\n";  # prints Hello class! followed by a newline (\n)
```

On all lines except the first, a hash sign (`#`) is used to begin a comment.

On Unix systems, use the `chmod` program to make your program executable.

```
chmod +x hello
```

Now you can run the program by typing its name, providing it is in a directory that is in your path.

```
hello
```

Hello class!

3.2 Command-Line Flags

Command-line flags or switches affect how Perl runs your program. To find out what version of Perl you are running, type

```
perl -v
```

This is perl, version 5.004_04 built for i386-linux (with 1 registered patch, see perl -V for more detail)

Copyright 1987-1998, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5.0 source kit.
Complete documentation for Perl, including FAQ lists, should be found on this system using 'man perl' or 'perldoc perl'. If you have access to the Internet, point your browser at http://www.perl.com/, the Perl Home Page.

Use the flag `-e` to execute Perl statements from the command line

```
perl -e 'foreach $i (1..20) { print "$i "; } print "\n"; '
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Don't worry about understanding how this command works. We'll cover it later in this course.

Command-line flags can also be placed inside programs, directly after `#!/usr/bin/perl`. The `-w` flag prints warnings intended for programmers about possible spelling errors and other error-prone constructs in the script.

```
#!/usr/bin/perl -w
$name = Nancy;
print "Hello $Name\n";
```

```
Name "main::Name" used only once:  possible typo at warning.pl line 3.
Name "main::name" used only once:  possible typo at warning.pl line 2.
Use of uninitialized value at warning.pl line 3.
Hello
```

The warnings should help you in debugging your programs.

Now you should be able to write a simple Perl program.

3.3 Quiz

Every so often I've included a quiz so that you can check whether you understood the material I've presented. Here's your first quiz.

1. Nearly every Perl statement ends in
   (a) `.`
   (b) `;`
   (c) `#`
   (d) `!`
2. Perl comments begin with
   (a) `%`
   (b) `;`
   (c) `#`
   (d) `/*`
   (e) `//`
   (f) None of the above.
3. Match up the command-line flag with what they do.
   (a) `-e` (d) Display Perl's version number.
   (b) `-v` (e) Warn you about potential errors.
   (c) `-w` (f) Execute Perl statements from the Unix command line.

3.4 Exercise

Only through practice will you become proficient at Perl. I strongly encourage you to work this exercise and some of the others in this text.

1. Write a program that prints what you want to do after this workshop.

3.5 Scalar Variables

In our first program we specified exactly what we wanted the program to print. Using a scalar variable, we can instruct the program to issue a greeting to someone. A scalar variable always consists of a `$` followed by a letter followed by any number of letters, numbers, or underscores. Both upper- and lower-case letters are legal but the variables `$name` and `$Name` cannot be used in place of each other unless you set one equal to the other. In other words, Perl is case sensitive.

```
#!/usr/bin/perl
$name = 'Clara';
print "Hello $name!\n";
```

Hello Clara!

Did you notice that I used two different types of quotes, single quotes (`'Clara'`) and double quotes (`"Hello $name!\n"`).

Perl replaces a variable's name with its value within double quotes. Backslashed characters within double quotes are expanded. For example, inside double quotes, a `\n` turns into a newline. Here is a table of some of the double-quoted string representations.

Construct	Meaning
`\n`	Newline
`\t`	Tab
`\r`	Carriage return
`\b`	Backspace
`\a`	Bell or beep
`\\`	Backslash
`\$`	$
`\%`	%
`\@`	@
`\"`	"
`\l`	Lowercase next letter
`\L`	Lowercase all following letters until `\E`
`\u`	Uppercase next letter
`\U`	Uppercase all following letters until `\E`
`\E`	Terminate `\L` or `\U`

Any character inside of single quotes is interpreted literally.[3]

3.6 Functions

There are functions built into Perl. You have already seen the `print()` function, which writes text to your screen. Unlike C, you can leave off parentheses. In Perl, parentheses are needed only to disambiguate between alternate interpretations. You could call `print` with parentheses, though they aren't necessary. Sometimes you can even omit quotes.

```
#!/usr/bin/perl
$greeting = 'Good morning';
print $greeting;
```

Good morning

[3]There are a couple of exceptions. If you want to put a quote (') in the string, precede it by a backslash and if you want to include a backslash, precede it by a backslash.

The `print` command can accept multiple arguments; just separate the arguments with commas.

```
#!/usr/bin/perl
$greeting = 'Good morning';
print $greeting, ' class!', "\n";
```

Good morning class!

The following program prompts you for your name.

```
#!/usr/bin/perl
print 'What is your name? ';
$name = <>;
chomp ($name);
print "Hello $name!\n";
```

Then it sets the scalar variable `$name` to what you enter. The command `chomp()` removes the trailing new line from your input. Then the program greets you by name, as you can see when I run this program.

What is your name? Nancy
Hello Nancy!

If you are at a loss as to what to write to someone, the Cyrano Server (`www.nando.net/toys/cyrano`) can help you to write valentines, love letters, and breakup notices. We can write our own Cyrano or madlibs program. Here we prompt the user for some information, manipulate it, and compose text based on the user-supplied information.

```
#!/usr/bin/perl
print 'Pick a number between 1 and 10:  ';
$number = <>;
chomp ($number);

print 'Give me a noun:  ';
$noun = <>;
chomp ($noun);
$noun = $noun . 's';  # Pluralize noun--append an 's'

print 'Give me a verb:  ';
$verb = <>;
chomp ($verb);
$verb = 'a-' . $verb . 'ing'; # Add prefix and suffix to $verb

print "\n$number $noun $verb\n";
```

Pick a number between 1 and 10: 8
Give me a noun: cow
Give me a verb: moo

8 cows a-mooing

When you enter a value, a newline is appended to the end of each of the variables (`$number`, `$noun`, and `$verb`). We use the `chomp` function to remove the last character only if it's a line separator. That's usually a newline, but you can define it to be something else with the `$/` special variable.

Here's what we would have gotten had we not called `chomp`.

```
8
 cow
s a-moo
ing
```

There are hundreds of functions built into Perl. The name of the function indicates the function's purpose.

Function	Description
`abs()`	Returns the absolute value.
`length()`	Returns the length of a string.
`print()`	Prints a string or a comma-separated list of strings.

Not all names will mean something if you, particularly if you are new to Perl.

Function	Description
`glob()`	Returns a value with file name expansions as a shell would do.
`rand()`	Returns a random fractional number.
`uc()`	Returns an uppercase version of its argument.

Lots of Perl functions have optional arguments, i.e., arguments that you can leave out when you call the function. You can find out about a function, variable, or operator in the Perl online manual pages or in a Perl reference book. The man page `perlfunc` describes the functions in Perl. Personally, I like the *Perl 5 Programmers Reference Manual* (Ventana) by R. Allen Wyle and Luke Duncan. Unlike the man page, it includes examples of nearly all the commands. It also lists related commands under *see also*. When I look up a command that doesn't do what I want it to do, I can check out the *see also* entries to see if any of them suit my needs.

3.7 Operators

Like a calculator, Perl can perform standard mathematical operations.

```
#!/usr/bin/perl
$a = 1 + 2 + 3;
$b = 1/3 + 2;
$c = 2 * 3 / -5;
print "a:  $a   b:  $b   c:  $c\n";
```

```
a:  6   b:  2.33333333333333   c:  -1.2
```

Perl employs the conventional order of precedence, i.e., exponentiation before multiplication and division before addition and subtraction. Use parentheses to change the order of precedence or to make it more obvious the order in which operations are performed.

```
#!/usr/bin/perl
$a = 2**3**2;
$b = 2**(3**2);
$c = (2**3)**2;
print "a:  $a   b:  $b   c:  $c\n";
```

```
a:  512   b:  512   c:  64
```

The string concatenation operator `.` glues two strings together, as you saw in our madlibs program.

```
$noun = $noun . 's';
$verb = 'a-' . $verb . 'ing';
```

Here we compute an employee's bonus, which is 1/7 of her salary.

```
#!/usr/bin/perl
$salary = 60000;
$bonus =  1/7 * $salary;
print "bonus:  $bonus\n";
```

bonus: 8571.42857142857

What if you don't like the way that Perl represents the bonus? You can use the function `printf()` to specify how you want the result displayed. The first argument tells how to format the result. The template field `%8.2f` displays `bonus` as a floating-point number, 8 characters wide with 2 places after the decimal point. Notice `printf()` rounds the number.

```
#!/usr/bin/perl
$salary = 60000;
$bonus =  1/7 * $salary;
printf "bonus:  \$%8.2f\n", $bonus;
```

bonus: $ 8571.43

See the backslash in front of the dollar sign in the line starting with `printf`? It's there to keep Perl from thinking that `$%` is a variable, which it is. It's a special variable that takes on the value of the current page number of a report prepared using `format`s, which in this example is 0, because we haven't used `format`. You'll learn about `format`s on page 34.

If you don't know the size of the result, you can use `%.2f` and Perl will print the result with two decimal places. The following table shows some of the `printf()` field types.

Field Types	**Description**
`%s`	String
`%c`	Character
`%d`	Integer
`%f`	Fixed-point floating-point number
`%e`	Exponential floating-point number (scientific notation)
`%u`	Unsigned integer
`%x`	Hexadecimal (lowercase)

If a number precedes the letter that specifies the type, e.g., `%5s`, the string or value is printed at least that many characters wide. If the string or value is shorter than the specified width, it is padded on the left with spaces.

You can put a flag between the percent sign and the field type.

```
#!/usr/bin/perl
printf "Number:  %+9d\n", -123456;
printf "Number:  %+9d\n",  123456;
printf "Number:  %-9d\n",  123456;
printf "Number:  %09d\n",  123456;
```

```
Number:    -123456
Number:    +123456
Number:  123456
Number:  000123456
```

Here's a description of the flags included in the last example.

Flag	Description
+	Always include a + or a - in the output.
-	Left-justify the output (this also works with strings).
0	Pad the number with zeros instead of spaces, as needed.

I find the string repetition operator, `x`, useful for underlining the titles of the financial reports. With the function `length()`, I obtain the number of characters in the title.

```
#!/usr/bin/perl
$title       = "HEDGED FINANCING REVERSES";
$titleLength = length($title);
print "$title", "\n", '-' x $titleLength, "\n";
```

```
HEDGED FINANCING REVERSES
-------------------------
```

Now you should be able to write a script using variables, functions, and operators.

3.8 Quiz

1. Which pair of lines produce the following output?

```
pie in the sky
pie in the sky
```

(a)
```
$food = "pie";
print '$food in the sky';
print '$food in the sky';
```

(b)
```
$food = 'pie';
print '$food in the sky\n';
print '$food in the sky\n';
```

(c)
```
$food = "pie";
print "$food in the sky";
print "$food in the sky";
```

(d)
```
$food = 'pie';
print "$food in the sky\n";
print "$food in the sky\n";
```

2. What does the following program print?

```
#!/usr/bin/perl
$word = "xyzzy\n";
chomp $word;
chomp $word;
$word = $word . 'ies';
print $word, "\n";
```

   (a) `ies`
   (b) `xyzzies`
   (c) `xyzzyies`
   (d) None of the above because there's an error in the program.

3. Which statement is a legal scalar assignment?

   (a) `april17 = $dayDay + 2;`
   (b) `$17april = $dayDay + 2;`
   (c) `$april.17 = $dayDay + 2;`
   (d) `$april___17 = $dayDay + 2;`

3.9 Exercises

1. Write a program that asks a user for a weight in pounds and returns the weight in kilograms.

   1 kilo = 2.2 lbs

2. Write a program that asks a user for a first name and last name and then prints them together on the same line.

3. *Extra credit (for whatever it's worth):* Print your first and last name together (with a space between them) enclosed in a box, e.g.,

```
------------------
| Nancy Blachman |
------------------
```

3.10 Conditional Expressions

Until now, every program you've seen in this course has executed all its statements in sequence. If you want to solve more complicated problems, you will probably want to write code that will, for instance, print some result when a particular condition is met. In this section, you will learn how to write conditional expressions using `if` and `else`.

3.10.1 Numeric Comparisons

As in mathematics, the symbols `<`, `>`, `=`, and `!` are used for testing numeric values.

Let's incorporate data checking into `product.pl`, a program from the last chapter that computed a product of two numbers supplied by a user. The following version tests that each number that the user enters are in between 1 and 12.

The conditional expression

```
(($number1 < 1) || ($number1 > 12))
```

tests whether the value of `$number1` is negative or greater than 12. Just like in the C programming language, the notation `||` is the logical OR operator. If either the first conditional expression (`$number1 < 1`) or the second conditional expression (`$number1 > 12`) evaluates to TRUE then the entire expression evaluates to TRUE.

I wrote the second conditional expression using different notation.

```
(($number2 >= 1) && ($number2 <= 12))
```

Here I test whether the value of `$number2` is greater or equal to 1 and less than or equal to 12. As you might have guessed, the notation `&&` is the logical AND operator. If bothr the first conditional expression (`$number2 >= 1`) and the second conditional expression (`$number2 <= 12`) evaluates to TRUE then the entire expression evaluates to TRUE.

```
#!/usr/bin/perl
# Filename: product.pl

print "This program computes the product of two numbers.\n";

# Prompt user for first number:
print "Please enter first number in the range [1, 12]: ";
$number1 = <STDIN>;
chomp $number1;          # remove "\n" from input
if (($number1 < 1) || ($number1 > 12)) {
    print "$number1 is out of the range [1, 12].\n";
    exit;
}

print "Please enter second number in the range [1, 12]: ";
$number2 = <STDIN>;
chomp $number2;          # remove "\n" from input
if (($number2 >= 1) && ($number2 <= 12)) {
    $product = $number1 * $number2;
    print "The product of $number1 and $number2 is $product.\n";
} else {
    print "$number2 is out of range [1, 12].\n";
}
```

Now let's run the program.

```
 Please enter first number:  6
Please enter second number:  14
14 is out of range [1, 12].
```

As in mathematics, the symbols `<`, `>`, `=`, and `!` are used for testing numeric values.

Expression	Evaluates to	When
x `==` y	`TRUE`	x is equal to y
x `>` y	`TRUE`	x is greater than y
x `<` y	`TRUE`	x is less than y
x `>=` y	`TRUE`	x is greater than or equal to y
x `<=` y	`TRUE`	x is less than or equal to y
x `!=` y	`TRUE`	x is not equal to y
`!`x	`TRUE`	not x, e.g., `! (1 > 2)` is TRUE
x `<=>` y	`-1, 0, 1`	x is less than (-1), equal to (0), or greater than (1) y

3.10.2 String Comparisons

Programming languages such as C and C++ don't make a distinction between numeric and string comparison operators. For instance, the "less than" operator (¡) is used to compare two numbers as well as to check if one string comes before another string in alphabetical order. Perl has two sets of comparison operators: one for comparing numerical values and the other for comparing strings.

Notice how the following program checks whether the user is interested in seeing the documentary BBC film 42 Up. If you type `yes`, the program asks you for money.

```
#!/usr/bin/perl
print 'Would you like to see the film 42 Up? ';
$answer = <>;
chomp $answer;
if ($answer eq 'yes') {
    print "That'll be \$9, please.\n";
} else {
    print "Enjoy your evening.\n";
}
```

```
Would you like to see the Film 42 Up?  yes
That'll be $9, please.
```

Notice that the `then` and `else` clauses are enclosed in curly braces (`{}`). You'll get a syntax error if you leave them out.

The `else` clause is optional. There are other ways to write an `if` statement. Pick the order that will make your code clearest to those who read or maintain it. It's often best to present the most important clause first.

```
print "Warning: Input out of range." if ($debug);
```

Here is a table showing the tests for comparing strings.

Expression	Evaluates to	When
string1 `eq` *string2*	`TRUE`	*string1* is equal to *string2*
string1 `gt` *string2*	`TRUE`	*string1* comes after *string2*
string1 `lt` *string2*	`TRUE`	*string1* comes before *string2*
string1 `ge` *string2*	`TRUE`	*string1* comes after or is equal to *string2*
string1 `le` *string2*	`TRUE`	*string1* comes before or is equal to *string2*
string1 `ne` *string2*	`TRUE`	*string1* is not equal to *string2*
`!(`*string1* `eq` *string2*`)`	`TRUE`	*string1* is not equal to *string2*
string1 `cmp` *string2*	`-1, 0, 1`	*string1* comes before (-1), after (1), or is equal to (0) *string2*

In addition to an `else` clause, you can have an `elsif` clause.

```
#!/usr/bin/perl
print 'Would you like to see the 7:30 pm showing of Shakespeare in Love? ';
$answer = <>;
chomp $answer;
if ($answer eq 'yes') {
    print "That'll be \$9, please.\n";
} elsif ($answer eq 'no') {
    print "Enjoy your evening.\n";
} else {
    print "I didn't understand your answer.\n";
}
```

3.11 Assignment Operators

Until now we have been assigning values to variables with the `=` sign, e.g., `$answer = <>`. You may want to change the value of the variable based on the previous value. There are shorthand notations for incrementing, decrementing, and performing other operations on variables.

Statement	Equivalent to	Meaning
`$num++`	`$num = $num + 1`	Add 1 to `$num` and return the previous value of `$num`
`$num--`	`$num = $num - 1`	Subtract 1 from `$num` and return the previous value of `$num`
`++$num`	`$num = $num + 1`	Add 1 to `$num`
`--$num`	`$num = $num - 1`	Subtract 1 from `$num`
`$num += 7`	`$num = $num + 7`	Add 7 to `$num`
`$num -= 7`	`$num = $num - 7`	Subtract 7 from `$num`
`$num *= 7`	`$num = $num * 7`	Multiply `$num` by 7
`$num /= 7`	`$num = $num / 7`	Divide `$num` by 7
`$num **= 7`	`$num = $num ** 7`	Raise `$num` to the power 7
`$noun .= 's'`	`$noun = $noun . 's'`	Append an 's' to `$noun`
`$string x= 3`	`$string = $string x 3`	Append two copies of `$string` to itself

3.12 Loops

Computers are great at doing repetitive tasks. Three commonly used looping constructs in Perl are `while`, `foreach`, and `for`.

In `flipCoin.pl`, I initialize the number of heads (`$heads`), the number of tails (`$tails`), and the number of coin flips (`$coinFlips`) to zero. If the total number of coin flips is less than 1000, I

simulate flipping a coin by generating a random number between 0 and 1. If the number is greater than .5, I add one to `$heads`, otherwise I add one to `$tails`. Then I add one to `$coinFlips`. I continue doing this until `$coinFlips` is no longer less than 1000.

```
#!/usr/bin/perl
$heads = 0;
$tails = 0;
for ($coinFlips = 0; $coinFlips < 1000; ++$coinFlips) {
    if (rand(1) > .5) {
        ++$heads;
     } else {
        ++$tails;
    }
}
$avgNumberOfHeads = $heads/1000;
print "Average number of times we got heads:  $avgNumberOfHeads\n";
```

Average number of times we got heads: 0.491

You can write that program using `while` in places of `for`.

```
#!/usr/bin/perl
while ($coinFlips++ < 1000) {
    if (rand(1) > .5) {
        ++$heads;
    } else {
        ++$tails;
    }
}
$avgNumberOfHeads = $heads/1000;
print "Average number of times we got heads:  $avgNumberOfHeads\n";
```

Average number of times we got heads: 0.502

Notice that I didn't set `$heads`, `$tails`, or `$coinFlips` to zero since variables are zero when initialized.

Here are some operators and functions you might find useful when programming.

Function or Operator	**Description**	**Example**
`%`	The `%` operator (pronounced "mod," short for "modulus") returns the remainder when the right side is divided by the left.	`17 % 5` → 2
`int(`*number*`)`	Returns the integer part or floor of its argument	`int(3.14159)` → 3
`rand(`*number*`)`	Returns a random floating-point number between 0 and *number*.	`rand(10)` → 5.060768

3.13 Quiz

Now for a short quiz.

1. Assume `$num` is 10. Which statement prints the following?

   ```
   100 81 64 49 36 25 16 9 4 1
   ```

   (a) `while($num > 0) { print $num * $num, ' '; }`
   (b) `while($num != 0) { print "$num * $num"; $num--; }`
   (c) `while($num >= 0) { print $num * $num, ' '; $num--; }`
   (d) `while($num >= 0) { $num--; print $num * $num, ' '; }`

3.14 Exercises

1. Write a program that asks for the temperature outside and prints "too hot" if the temperature is above 72 and "too cold" otherwise.

2. Modify the program from the previous exercises so that it prints "too hot" if the temperature is above 75, "too cold" if the temperature is below 68, and "just right" if it is between 68 and 75.

4 Scalars, Lists, Arrays, and Hashes

Perl has three kinds of data structures: scalars, arrays of scalars, and associative arrays of scalars, known as *hashes*.

4.1 Lists, and Arrays

Scalar variables begin with a `$` sign, e.g., `$a`. Array variables begin with an *at* sign (e.g., `@movies`), and their values can be represented by a comma-separated list of scalars enclosed in parentheses.

```
@languages = ('Perl', 'C', 'Java', 'HTML');
@days = (5,6,7,8,9,12,13,14,15,16);
```

If you don't feel like typing all those numbers, you can use the `..` operator to generate a list of sequential numbers.

```
@days = (5..9,12..16);
```

The range operator works with strings as well.

```
@letters = ('A'..'Z');
```

If you don't feel like typing the quotes and the commas, you can use `qw` to quote each word or string and interpret them as separate elements. But you can't use `qw` with the range operator.

```
@first_five_letters = qw(A B C D E);
@letters = qw(A .. Z);
```

You can use any character or bracket, e.g., `()`, `{}`, `[]`, to start and end the quoted expression.

```
@animals = qw#cat dog mouse#;
@animals = qw[cat dog mouse];
```

You can assign more than one value with a single assignment statement.

```
#!/usr/bin/perl
($a, $b) = (3, 4);
($c, @d) = (5..10);
print "a:  $a    b:  $b    c:  $c    d:  @d\n"
```

a: 3 b: 4 c: 5 d: 6 7 8 9 10

The loop construct `foreach` executes the statements in *body* for each *array* element in turn.

```
foreach $i (array) {
    body
}
```

Here's an example.

```
#!/usr/bin/perl
@days = (5..9,12..16);
print "Week days:  ";
foreach $i (@days) {
    print "$i ";
}
print "\n";
```

Week days: 5 6 7 8 9 12 13 14 15 16

But we didn't need to write a loop to print out each of the values in an array, because we can print the array itself.

```
#!/usr/bin/perl
@days = (5..9,12..16);
print "Week days:  @days\n";
```

Week days: 5 6 7 8 9 12 13 14 15 16

I hope you can now understand the one line Perl command that appeared on page 4.

```
perl -e 'foreach $i (1..20) { print "$i "; } print "\n"; '
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

4.2 Splitting Strings

Converting a string to an array is easy to do with the `split` function.

`split(`*pattern*`,` *string*`)`	Returns an array constructed by using the *pattern* as a delimiter to distinguish elements in *string*.

Here I use `split()` to construct an array from a record where colons (`:`) are used as delimiters.

```
#!/usr/bin/perl
$record  = "Nancy::June 18:::Palo Alto:A+::";
@record  = split(":", $record);    # Extract entries from record
foreach $entry (@record) {
        print "/$entry";
}
print "/\n";
```

/Nancy//June 18///Palo Alto/A+/

Space is a special delimiter. If you have more than one space in a row, it is interpreted as a single space. Here I split a description of the off-Broadway show Wit into a list of words using space as the delimiter.

```
#!/usr/bin/perl
$wit  = "warm brilliant and    moving ";
@wit  = split(" ", $wit);    # Extract the words
@wit  = sort @wit;           # Sort the words
foreach $word (@wit) {
        print "/$word";
}
print "/\n";
```

/and/brilliant/moving/warm/

4.3 Array Indexing

All arrays are ordered: You can refer to the 1st element, the 4th, or the 2nd to last. Like C, Perl uses *zero-based indexing*, which means the elements are numbered starting with 0. So the first element is at position 0 and the 5th element is at position 6.

There are several different ways to access the elements of an array. You can access individual elements with `$` and `[]`. Negative subscripts count from the end of a list. Subarrays are accessed with `@` and `[]`.

```
#!/usr/bin/perl
@languages = ('Perl', 'C', 'French', 'HTML');
print "The 2nd element:  $languages[1]\n";
print "The 2nd to last element:  $languages[-2]\n";
print "The 2nd through 4th element:  @languages[1..3]\n";
print "The index of the last element:  $#languages\n";
print "The last element:  @languages[$#languages]\n";
```

The 2nd element: C
The 2nd to last element: French
The 2nd through 4th element: C French HTML

```
The index of the last element:  3
The last element:  HTML
```

There are several different ways for obtaining the number of elements in an array. When you assign a scalar variable equal to an array, the scalar variable will be set to the number of elements in the array.

```
$length = @array;
```

There's an easier way to obtain the number of elements in an array. No need to create a variable, just call the function `scalar` on the array.

```
print "The array has ", scalar(@array), "elements\n";
```

You can add and remove elements from the end of an array using `push` and `pop` respectively. You can add and remove elements from the beginning of an array using `unshift` and `shift` respectively.

```
#!/usr/bin/perl
@languages = ('C', 'HTML');
pop(@languages);
push(@languages, ('Java', 'Pascal'));
unshift(@languages, 'Perl');
print "\@languages:  @languages\n";
```

```
@languages:  Perl C Java Pascal
```

4.4 Hashes or Associative Arrays

Regular arrays are indexed by numbers; *hashes*, also known as *associative arrays*, are indexed by strings. Hashes let you associate one scalar with another. You can think of a hash as being like an address book on a cellular phone that lets you assign whatever name you want to every number you store. So I could assign the name *home* to my home phone number.

There are several ways to store data in hashes. Here I create the `%daysPerMonth` hash, initialized with three key-value pairs.

```
%daysPerMonth = ('Jan' => 31, 'Feb' => 28, 'Mar' => 31);
```

Below I map the name of a month to the number of days in that month.

```
$daysPerMonth{'Jan'} = 31;
$daysPerMonth{'Feb'} = 28;
$daysPerMonth{'Mar'} = 31;
```

You can convert arrays to hashes. The even-numbered elements in the array become the keys and the odd-numbered elements become values.

```
#!/usr/bin/perl
%daysPerMonth = ('Jan', 31, 'Feb', 28, 'Mar', 31);
print "Jan has $daysPerMonth{Jan} days\n";
print "Feb has $daysPerMonth{Feb} days (usually)\n";
```

```
Jan has 31 days
Feb has 28 days (usually)
```

Keys can be any string. If the key has a space in it, make sure to enclose it in quotes.

```
$academyAwards{'Shakespeare in Love'} = 7;
```

4.5 Retrieving Values from Hashes

We are able to retrieve values using keys. What if you don't know the keys or you don't want to specify them explicitly? From a hash, you can extract an array of keys (with `keys()`) or of values (with `values()`).

```
#!/usr/bin/perl
%daysPerMonth = ('Jan' => 31, 'Feb' => 28, 'Mar' => 31);
@keys = keys %daysPerMonth;
print "Keys for \%daysPerMonth:  @keys\n";
```

Keys for %daysPerMonth: Mar Jan Feb

```
#!/usr/bin/perl
%daysPerMonth = ('Jan' => 31, 'Feb' => 28, 'Mar' => 31);
@values = values %daysPerMonth;
print "Values stored in \%daysPerMonth:  @values\n";
```

Values stored in %daysPerMonth: 31 31 28

We can loop through a hash using `keys()`.

```
#!/usr/bin/perl
%daysPerMonth = ('Jan' => 31, 'Feb' => 28, 'Mar' => 31);
foreach $key (keys %daysPerMonth) {
    print "$key is $daysPerMonth{$key} days long.\n";
}
```

Mar is 31 days long.
Jan is 31 days long.
Feb is 28 days long.

You'll often want to access both keys and values simultaneously, as we did in the last example. You can obtain both with `each()`.

```
#!/usr/bin/perl
%daysPerMonth = ('Jan' => 31, 'Feb' => 28, 'Mar' => 31);
while (($key, $value) = each %daysPerMonth) {
    print "$key is $value days long.\n";
}
```

Mar is 31 days long.
Jan is 31 days long.
Feb is 28 days long.

Unlike arrays, values in hashes are not stored in any particular order. Notice that the retrieval order of `keys()`, `values()`, and `each()` bears no resemblance to the assignment order. You can rearrange the order of the elements with the `sort()` function, which is described on page 42.

There are times when you want to check whether a variable is defined. You can use the function `defined()` for that purpose.

```
#!/usr/bin/perl
$a = 5;
if (defined($a)) {
    print "The variable \$a is $a.\n";
```

```
}
if (defined($b)) {
    print "The variable \$b is $b.\n";
} else {
    print "The variable \$b isn't defined.\n";
}
```

The variable $a is 5.
The variable $b isn't defined.

Here's an example that uses `defined()` to check whether the user supplied arguments (file names) when calling the program.

```
#!/usr/bin/perl
if (!defined(@ARGV)) {
    print "Usage:  $0 file1 file2 ...\n";
}
foreach $inputFile (@ARGV) {
   .
   .
   .
}
```

The special variable `$0` is set to the name of the program that is running. The array `@ARGV` contains all the command-line arguments provided to your program.

The following table lists the functions for checking whether variables, arrays, hashes, and subroutines are defined.

Function	**Description**
`defined(`*var*`)`	Returns `TRUE` (1) if the variable `var` is defined and `FALSE` otherwise. The argument to `defined()` can be an array, hash, or subroutine.
`exists($`*hash*`{`*key*`})`	Returns `TRUE` (1) if *key* exists in *hash*.
`undef(`*var*`)`	Undefines the value of *var*, which can be a scalar, an array, a hash, or a subroutine.

Here are the ways we covered on how to refer to a value in a scalar, array, and in a hash. You can determine the data structure by looking at the first character of the name.

Variable Name	**Description**
`$days`	The simple scalar value `$days`
`$days[28]`	The 29th element of array `@days`
`$#days`	The last index of array `@days`
`$days{'Feb'}`	The `Feb` value from the hash `%days`

Entire arrays or array slices are denoted by `@`.

Name	**Description**
`@days`	Same as `($days[0], $days[1], ..., $days[n])`
`@days[3,4,5]`	Same as `($days[3], $days[4], $days[5])`
`@days{'a','c'}`	Same as `($days{'a'}, $days{'c'})`

Every variable type has its own namespace. In other words, you can, use the same variable name for a scalar variable, an array, and a hash. This means that `$days` and `@days` are different variables. It means that `$days[28]` is part of `@days` and `$days{'Feb'}` is part of `%days`, but neither is part of `$days`.

4.6 Quiz

I hope you find this quiz more interesting than the last one.

1. What does `@letters` contain?

```
@letters = ();
$letters[0] = 'c';
$letters[1] = 'o';
$letters[2] = 'd';
$letters[0] = 'h';
$letters[3] = 'e';
@letters[1..2] = ('a', 'r');
push(@letters, 's');
```

   (a) `c o r e`
   (b) `s h a r e`
   (c) `c o d e s`
   (d) `h a r e s`

2. What is `$movie` after the following statements?

```
@movies = ('Life is Beautiful', 'Shakespeare in Love', 'Elizabeth', 'E.T.');
$movie = pop(@movies);
push(@movies, $movie);
$movie = pop(@movies);
$movie = pop(@movies);
push(@movies, $movie);
$movie = pop(@movies);
```

   (a) `Life is Beautiful`
   (b) `Shakespeare in Love`
   (c) `Elizabeth`
   (d) `E.T.`

3. How would you print the second and fourth values of the array `@movies` (as defined in the last problem) with a statement that only includes `@movies` once?
4. What does the following code print?

```
#!/usr/bin/perl
$hemispheres{Finland} = 'Northern';
%hemispheres = ('Peru' => 'Southern', 'Germany' => 'Northern');
foreach (keys %hemispheres) {
    $i++;
}
print "$i\n";
```

   (a) 1
   (b) 2
   (c) 3
   (d) 4

5. What does the following code print?

```
#!/usr/bin/perl
%hemispheres = ('Peru' => 'Southern', 'Germany' => 'Northern');
$hemispheres{Finland} = 'Northern';
foreach (keys %hemispheres) {
    $i++;
}
print "$i\n";
```

   (a) 1
   (b) 2
   (c) 3
   (d) 4

4.7 Exercise

1. Write a program that asks for 10 numbers and prints them out in reverse order. Your code should only contain one call to request input from the user. Invoke that call 10 times.

5 Pattern Matching: Regular Expressions

Wouldn't it be nice to recognize patterns in a line, extract a couple of columns of text, replace a string, such as "Version 2" with another, such as "Version 3", or select elements that meet some condition from an array. All these tasks can be performed using pattern matching.

5.1 The Substitution Operator

Here we use the substitution operator `s///` to replace the word "which" with the word "that" in the input string `$input`.

```
#!/usr/bin/perl
while ($input = <>) {
    $input =~ s/which/that/;
    print $input;
}
```

```
strunk.pl
which
that  the tea cup which was red
the tea cup that was red
```

When using the substitution operator, specify a pattern to search for and what to replace it with.

`$`*string* `=~ s/`*pattern*`/`*replacement*`/`	Searches `$string` for *pattern* and, if found, replace *pattern* with *replacement*

In English pronouns are used to shorten and simplify sentences.

> After Janet attends the Perl seminar, Janet will be able to find a lucrative job.

Instead of using Janet twice in the sentence, we can replace the second occurrence with "she" and most readers/listeners will know to whom the "she" refers.

> After Janet attends the Perl seminar, *she* will be able to find a lucrative job.

Perl uses `$_` somewhat like how we use pronouns. Some functions and operators can use `$_` in place of an argument. In such cases, you can omit that argument and Perl will use the contents of `$_` instead. When using `$_` we no longer need to name the input `$input`.

```
#!/usr/bin/perl
while (<>) {        # Assigns each input line to $_
    s/which/that/;  # In $_, replace "which" with "that"
    print;          # Print $_
}
```

Wouldn't it be nice if `strunk.pl` could read files? It can! Let's look at the file `text.txt`.

```
That which is, is not; that which is not, is.
```

Now let's call `strunk.pl` with the file `text.txt`.

```
strunk.pl text.txt
That that is, is not; that which is not, is.
```

Is that right? The substitution worked only for the first *which*. The second *which* remained unchanged. That's because `s///`, by default, replaces only the first matching string. To change all strings, use the `/g` modifier, which stands for *global*.

```
#!/usr/bin/perl
while (<>) {
    s/which/that/g;
    print;
}
```

```
strunk.pl text.txt
That that is, is not; that that is not, is.
```

I showed you how to substitute one string for another. But regular expressions don't have to be literal strings. The program `cleanse.pl` removes extraneous white space: All consecutive spaces, tabs, and newlines are replaced by a single space.

```
#!/usr/bin/perl
while (<>) {
    s/\s+/ /g;
    print $_, "\n";
}
```

Take a look at the file `swing.txt`. On a UNIX system, the command `cat` displays the contents of the file.

```
cat swing.txt
Up in the sky,   Up in the    air so blue
```

Now let's run our script on the file.

```
cleanse.pl swing.txt
Up in the sky, Up in the air so blue
```

5.2 Metacharacters

A *metacharacter* is a normal alphanumeric character preceded by a backslash, which gives it a special meaning. The metacharacter `\s` matches any white space, including spaces, tabs, and newlines.

So if `\s` stands for any white space character, what's `\S`? There's a simple rule: Capitalized metacharacters are the opposite of their lower-case counterparts. So `\S` represents anything that isn't white space, such as a letter, a digit, or some other printable character.

The `+` in `\s+` is a regular expression special character. The `+` will match one or more of whatever precedes the symbol. In `cleanse.pl`, `+` was preceded by `\s`, which means that it will match one or more white space characters. A `*` matches zero or more of whatever precedes the symbol, which I use in the next example.

The notation `[`*ab*`]` matches character or pattern *a* or the character or pattern *b*. You can have as many characters as you like inside the square brackets. But it's cumbersome to list a lot of characters, so Perl provides some shorthand notations.

Patterns	**Description**
`[djt]`	Matches either `d`, `j`, or `t`.
`[3-9]`	Matches any digit between 3 and 9.
`[a-zA-Z]`	Matches a letter of the alphabet.
`[^a-zA-Z]`	Matches any character except a letter of the alphabet.

The character ^ matches the beginning of a line. The character $ matches the end of a line. Use ^ and $ where ever you can; they increase the speed of regular expression matches.

```
/^Notice/
```

matches any string beginning with the word "Notice."

```
/-\s*$/
```

matches any string ending with a hyphen or dash, possibly followed by some white space.

If you ask users to type "Yes" or "No," I wouldn't be surprised if you get answers including "Y," "y," "YES," "yes," "yeah," "N," "n," and "no." Taking that into consideration, I wrote the following code, which looks for input starting with a "y" or a "Y."

```
#!/usr/bin/perl
print "Do you want to play tic-tac-toe (Yes or No)? ";
$userInput = <>;
$playGame = ($userInput =~ m/^\s*[yY]/);
print "Play tic-tac-toe? $playGame\n";
```

```
Do you want to play tic-tac-toe (Yes or No)?   YES
Play tic-tac-toe?  1

Do you want to play tic-tac-toe (Yes or No)?   Not now
Play tic-tac-toe?
```

Instead of using `[yY]`, which matches a lower case or an upper case Y, we can use the `/i` modifier to treat all letters as lower case. In the following statement, `$playGame` will be set to 1 if given input that starts with either a lower-case or upper-case Y.

```
$playGame = ($userInput =~ /^\s*y/i);
```

The expressions

```
$var =~ m/pattern/
$var =~ /pattern/
```

search for *pattern* in `$var` and return 1 if it is found.

Here are metacharacters I've shown you and some others.

Metacharacters	Description	Pattern String
\t	Tab	
\n	Newline	
\r	Return	
\f	Form feed	
\s	Matches any whitespace character	[\r\t\n\f]
\S	Matches anything that's not whitespace	[^ \r\t\n\f]
\d	Matches a digit	[0-9]
\D	Matches anything that's not a digit	[^0-9]
\w	Matches a word character (letters, digits, or the underscore)	[_0-9a-zA-Z]
\W	Matches a non-word character	[^_0-9a-zA-Z]
\b	Matches a word boundary	
\B	Matches a non-word boundary	
\Q	Implicitly quotes metacharacters that follow	
\E	Terminate \Q quoting	

The program `frenchize.pl` contains a small English-to-French dictionary and uses it to make substitutions.

```
#!/usr/bin/perl
%french = ('Hello'     => 'Bonjour',
           'drink'     => 'boisson',
           'the class' => 'la classe');
while (<>) {
    s/\b Hello      \b/$french{Hello}/gx;
    s/\b drink      \b/$french{drink}/gx;
    s/\b the\ class \b/$french{'the class'}/gx;
    print;
}

frenchize.pl \\
Hello! After the class, would you like to get a drink? \\

Bonjour!  After la classe, would you like to get a boisson?
```

In `frenchize.pl`, I used two different modifiers.

Modifier	Description
`/g`	Finds all occurrences, i.e., matches globally
`/x`	Ignore white space unless backslash or within brackets

Here are some of Perl's regular expression special characters or quantifiers.

Quantifiers	Description
`*`	Matches 0 or more of whatever precedes the symbol (as many as possible)
`+`	Matches 1 or more of whatever precedes the symbol (as many as possible)
`?`	Matches 0 or 1 of whatever precedes the symbol (1 if possible)
`{`n`}`	Matches exactly n of whatever precedes the quantifier
`{`n`,}`	Matches at least n of whatever precedes the quantifier (and as many as possible)
`{`n`, `m`}`	Matches at least n but no more than m of whatever precedes the quantifier

You've probably noticed that parentheses are used to isolate expressions in Perl, just as in mathematics. They do the same thing, and a little bit more inside regular expressions. When a regular expression contains parentheses, Perl remembers which substrings matched the part inside and returns a list of them when evaluated in a list context, e.g., `@array = /(\w+) (\d+) (\W+)/`). The regular expressions in parentheses are stored in the temporary variables `$1`, `$2`, and `$3`.

Here's a program that transforms dates in the form YYYYMMDD to MM/DD/YYYY. It uses the substitute operator to search for a number with 4 digits (the year), followed by 2 digits (the month), followed by another 2 digits (the day), and then rewrites the expression with the month followed by the day followed by the year.

```
#!/usr/bin/perl
$date = 19990417;
chomp $date;
$date =~ s| (\d{4})     # Year
            (\d{2})     # Month
            (\d{2})     # Day
          |$2/$3/$1|x;  # Ignore spaces in the pattern
print "Date:  $date\n";
```

Date: 04/17/1999

Notice that I used the delimiter | instead of slash / with the substitution operator. I didn't want to have to escape the slash in the result, e.g., `/$2\/$3\/$1/`. Fortunately, you can use any character you like with the substitution operator as long as you use the same one in a given expression, i.e., `s!which!that!` or `s~which~that~`. Avoid using characters that appear in the input pattern or your specification of the result.

Regular Expression Variables	Description
`$1,$2,$3,...`	Regular expression "backreferences."
`$+`	The string matched by the last pair of parentheses.
`$&`	The string matched by the last regular expression.

Here are some other special characters that you might find useful when using Perl as well as when doing the exercises below.

Special Characters	Description
`^`	Matches the beginning of a string
`$`	Matches the end of a string
`\`	Escapes the next character
`.`	Matches any character (except newline)
`...`	Matches any 3 characters (except newline)
`.*`	Matches any number (including 0) of characters (except newline)
`()`	Groups a series of pattern elements to assist in back referencing
`[]`	Matches if one of the choices in the set matches
`[^]`	Matches a character not in the set

5.3 Quoting Expressions

What if you want to replace the value of `$oldFile` with the value of `$newFile`? Why won't either of the following substitutions work?

```
s/$oldFile/$newFile/
s/\$oldFile/\$newFile/
```

So how would you do it? Use `\Q` to instruct Perl not to interpret `$` as well as other special character. Use `\E` to terminate the special quoting.

```
#!/usr/bin/perl
$oldFile = "tradersOrig.pl";
$newFile = "traders.pl";
$theString = "Filename:  tradersOrig.pl.";
```

```
$theString =~ s/\Q$oldFile\E/\Q$newFile\E/;
print "$theString\n";
```

Filename: traders\.pl.

The result isn't quite what I wanted, but I'll let you figure out how to remove the backslash that's in front of the period (.).

Here's a summary of quote and quote-like operators. The backquote operator I describe on page 35 in this section. The last column, with the heading *Interpolates*, indicates whether Perl performs *variable interpolation*, i.e., replaces a variable's name with its value.

Customary	Generic	Meaning	Interpolates
''	q{}	Literal	no
""	qq{}	Literal	yes
``	qx{}	Command	yes
	qw{}	Word list	no
//	m{}	Pattern match	yes
s///	s{}{}	Substitution	yes

5.4 Quiz

1. What will the following statements print?

```
$_ = "Bugs Bunny\n";
s/ Bunn//g;
print;
```

   (a) `Bugs Bunny`
   (b) `Bugsy`
   (c) `gsy`
   (d) `Bugsny`

2. What will the following program print?

```
#!/usr/bin/perl
$_ = "Sarnoff Corporation\n";
s/o/i/;
s/r/e/;
s/e/u/g;
print;
```

   (a) `Sauniff Ciupiuatiin`
   (b) `Sauniff Cuupuuatiun`
   (c) `Sauniff Curpuratiun`
   (d) `Sauniff Corporation`

3. What string doesn't match `/a+b*c?/` ?

   (a) `aabbc`
   (b) `a`
   (c) `abc`
   (d) `b`

4. What string doesn't match `/d{1,3}.o?.g+/` ? (Be careful!)

   (a) `dddg`
   (b) `ddddoogg`
   (c) `dog`
   (d) `dogg`

5.4.1 Exercises

1. Write a Perl script that reads in each line of a file and print the line number in front of each line.

```
1| This is the first line of the text.
2| This is the next line of the text.
3| this is the 3rd and final line.
```

2. *Difficulty:* Medium
   Given a number, use `while` to insert commas between sets of three digits that are to the left of the decimal point, i.e., your script should convert `123456789` to `123,456,789`.
3. Read an array of data. Change the order of the last two columns. For example, convert the matrix

```
1 2 3        1 3 2
4 5 6   to   4 6 5
7 8 9        7 9 8
```

4. Change the Perl program on page 27 so that the output doesn't include an escape character (backslash) in front of the period.

5. *Difficulty:* Medium to hard (depending on what your translator handles)
Write a TEX to HTML translator and a HTML to TEX translator that translates text that might include italic and bold type. The following table shows how to specify an *italic* and a **bold** face in TEX and HTML.

TEX	HTML
{\it *italic text* }	*italic text*
{\bf **bold text** }	**bold text**

6. *Difficulty:* Medium
Perl's pattern matcher is greedy. In other words, `+`, `*`, `?`, and `{}` all match as many characters as they can.

```
#!/usr/bin/perl
$_ = "I have 9 numbers.  The first is:  10072";
if ( /(.*)(\d*)/ ) {
    print "First pattern is <$1>, Number is <$2>.\n";
}
```

What part of the string do you think matches the first pattern and what part matches second pattern (the number)? Take a look at the result.

First pattern is <I have 9 numbers. The first is: 10072>, Number is <>.

There are times when you want those operators to match as few characters as possible. An extra `?` curbs their greed.

Quantifiers	Description
`+?`	Matches one or more times; as few as possible
`*?`	Matches zero or more times; as few as possible
`?`	Matches zero or once; zero, if possible
`{`*n*`,` *m*`}`	Between *n* and *m*; as few as possible

Here's a script that will let you see how greedy and non-greedy patters match, or rather don't match as you might hope or expect. Instead of putting quotes around each pattern, I used `qw`, which quotes each word (actually pattern).

```
#!/usr/bin/perl
$_ = "I have 9 numbers.  The first is:  10072";
@patterns = qw{
    (.*)(\d*)
    (.*)(\d+)
    (.*?)(\d*)
    (.*?)(\d+)
    (.*)(\d+)$
    (.*?)(\d+)$
    (.*)\b(\d+)$
    (.*\D)(\d+)$
```

```
};
for $pattern (@patterns) {
    printf "%-12s ", $pattern;
    if ( /$pattern/ ) {
       print "<$1> <$2>\n";
    } else {
       print "FAIL\n";
   }
}
(.*)(\d*)    <I have 9 numbers.  The first is:  10072> <>
(.*)(\d+)    <I have 9 numbers.  The first is:  1007> <2>
(.*?)(\d*)   <> <>
(.*?)(\d+)   <I have > <9>
(.*)(\d+)$   <I have 9 numbers.  The first is:  007> <2>
(.*?)(\d+)$  <I have 9 numbers.  The first is:  > <10072>
(.*)\b(\d+)$ <I have 9 numbers.  The first is:  > <10072>
(.*\D)(\d+)$ <I have 9 numbers.  The first is:  > <10072>
```

Explain why Perl obtained each of the results displayed here. For example, for the first, because Perl uses a greedy algorithm, `.*` matched the entire string and so `\d*` was matched with the empty string.

6 Reading from and Writing to Files

So far you've seen three data types: scalars, arrays, and hashes. A fourth data type, called a *filehandle*, acts as a bridge between your program and the outside world: files, directories, or other programs. In this section you will learn how to read from and write to files.

6.1 Reading from Files

Earlier in this course, we used empty angle brackets (`<>`) to read files supplied on the command line and to read the input a user types. In both cases, the angle brackets implicitly open a file handle. When your program grabs user input, it's using the `STDIN` (standard input) filehandle. When your program reads command-line files, it's using the `ARGV` filehandle. And whenever it `print()`s something, it's using the `STDOUT` filehandle. Built-in filehandles are all uppercase. I recommend you follow this convention so you can distinguish your filehandles from your variable and subroutine names. Perl fills in empty angle bracket with either `STDIN` or `ARGV` for you. If you want to use both in your program, supply the filehandle yourself:

```
$answer = <STDIN>;
```

sets `$answer` to the next line of standard input.

```
$line = <ARGV>;
```

sets `$line` to the next line of the command-line files. When your program reads command-line files in this manner, the special variable `$ARGV` is set to the current filename.

Empty angle brackets are interpreted as `<ARGV>` when there's data to be read from command-line files and interpreted as `<STDIN>` otherwise. The context of the angle brackets determines how much data is read:

```
$line = <ARGV>;
```

evaluates `<ARGV>` in a scalar context, retrieving the next line from the `ARGV` filehandle.

```
@line = <ARGV>;
```

evaluates `<ARGV>` in a list context, retrieving all the lines from `ARGV`.

There are three standard UNIX filehandles:

Standard Filehandles	Description
`STDIN`	Standard input: what the user types–program input
`STDOUT`	Standard output: what's printed on the user's screen
`STDERR`	Standard error: error messages printed to the user's screen

You can create your own filehandles with `open()`. The call `open(`*FILEHANDLE*`, `*string*`)` opens the filename given by *string* and associates it with *FILEHANDLE*. The file `rates.dat` contains pairs of numbers: an index and an interest rate.

```
1       4.347
2       4.347
3       4.349
4       4.350
```

Here's a program that reads data from the file `rates.dat`. This program is similar to one that I use at work to read interest rates. The line `open(DATA, "<rates.dat");` opens `rates.dat` for input and associates it with the filehandle `DATA`.

```
#!/usr/bin/perl
open(DATA, "<rates.dat");

# Read rate data, 2 number, an index and a rate, e.g., 1   4.237
# Just store the rate.  The index is the position of
# the item in the list. Note: $1 is the rate.
while($line = <DATA>) {
    if ($line =~ /^\s*\d*\s+(\d*\.\d*)/) {
        push (@rates, $1);
    }
}
close(DATA);
print "@rates\n";
```

4.347 4.347 4.349 4.350

As you might expect, you can close a file using `close(`*FILEHANDLE*`)`. What if you forget to close the files you opened? When a program exits, Perl closes all open filehandles.

6.2 Exiting from Your Program

In Unix, whenever a program ends, it exits with an exit status. If the program completes normally, that value is 0; if the program fails in some way, the exit status is some other integer that helps identify why the program failed. You can call `exit(`*number*`)` to have your program exit with status *number*.

```
exit(3);
```

What happens if `readRates.pl` tries to open `rates.dat` but can't because it doesn't exist or our program doesn't have permission to read the file? Then the call to `open()` will fail and return a `FALSE` value. I can instruct my program to exit if that value is returned and print an informative error message with the `die()` function.

`die(`*string*`)` Exits with the current value of the special variable `$!`, which contains the system error number, and prints *string*.

It's a good idea to use `die()` whenever there's a chance that an `open()` will fail.

```
open(DATA, "<$file") || die "Can't open file $file ($!)\n";
```

What's `||` mean? It stands for *or*. If the statement on the left of `||` is true, then the whole statement is true. The statement on its right is executed only if the statement on the left is false.

If your program detects an error that is serious enough to report to the user, but not so critical that the program should `die()`, consider `warn()`ing the user.

`warn()` Prints a string like `die()` but doesn't terminate the program.

Here I warn the user that my program wasn't able to read the data from the rates file.

```
if ($#rates == -1) {
    warn ("Unable to read rates in $file\n");
} else {
    print "@rates\n";
}
```

If you don't know the exact name of the file you want or you want to access several files, consider using the `glob()` function.

`glob(`*string*`)` Returns an array of the filenames matching the wildcards in *string*.

Here I obtain all the filenames ending in `.c`.

```
@files = glob('*.c');
```

6.3 Writing to a File

When you invoke `print()` (or `printf()`), by default, Perl prints to the filehandle `STDOUT` (standard output). Here I print to the file `hedge.out`.

```
#!/usr/bin/perl
$file = "hedge.out";
```

```
open(FILE, ">$file") || die "Can't write to $file ($!)\n";
$title =  "HEDGED FINANCING REVERSES";
$titleLength = length($title);
printf FILE "$title\n%s\n", ('-' x $titleLength);
```

HEDGED FINANCING REVERSES

Did you notice the difference in the second argument when I `open()`ed a file for reading and writing? The characters at the beginning of the second argument to `open()` affect how the filehandle is created.

Command	Description
`open(`*HANDLE*`, "`*filename*`")`	open *filename* for reading
`open(`*HANDLE*`, "<`*filename*`")`	open *filename* for reading
`open(`*HANDLE*`, ">`*filename*`")`	create *filename* for writing
`open(`*HANDLE*`, ">>`*filename*`")`	open or create *filename* for appending
`open(`*HANDLE*`, "+<`*filename*`")`	open *filename* with read and write access
`open(`*HANDLE*`, "+>`*filename*`")`	create *filename* with read and write access
`open(`*HANDLE*`, "+>>`*filename*`")`	open or create *filename* with read and write access

If you don't put any character in front of *filename*, Perl will open the file for reading, the same as it does if *filename* is preceded by `<`.

6.4 Formats

When Larry Wall created Perl, one of his motivations was to have a tool for generating reports. He created `format` so that you could easily control the appearance of text on pages and so you could let Perl deal with

- Page headers
- Document pagination
- Multicolumn text
- The number of lines on a page

Using `format`, you code up your output page to resemble how you want it to look when it's printed. Formats are declared as follows:

`format` *name* `=`
formList
variableList
`.`

The *formList* specifies the appearance of the output. The *variableList* specifies the variables to be printed. If *name* is not specified, `STDOUT` is used. Here are the ways you can display your data.

Form	Description
`@<<<`	Left justified
`@>>>`	Right justified
`@|||`	Centered
`@#.##`	Number with implied decimal point
`@*`	A Multi-line field

The length of the form indicates the amount of space the quantity is to occupy.

Like a subroutine declaration, a `format` declaration won't do anything until it is invoked with the `write()` function.

```
#!/usr/bin/perl
format EARNINGS_HEADER =
Earnings Distribution of Year-Round, Full-Time Workers by Sex, 1994
        Number (in 1000s) Distribution (%)
Earnings Group    Women      Men    Women     Men
.
format EARNINGS =
@<<<<<<<<<<<<<  @>>>>>> @>>>>>>>  @##.##% @##.##%
   $group,      $women,   $men,   $womenD, $menD
.

@groups  = ("\$7,500 or less", "\$7,501-\$12,499", "\$12,500-\$19,999");
@women   = (6233, 6552, 9729);
@men     = (6038, 5958, 9703);
@womenD  = (14.0, 14.7, 21.9);
@menD    = (9.5, 9.4, 15.2);

open(EARNINGS, ">earnings.out") || die "can't write earnings.out (!$)\n";
write EARNINGS_HEADER;
foreach $i (0..$#groups) {
    $group  = $groups[$i];
    $women  = $women[$i];
    $men    = $men[$i];
    $womenD = $womenD[$i];
    $menD   = $menD[$i];
    write EARNINGS;
}
```

This program produces the report `earnings.out`, which I've shown below.

```
Earnings Distribution of Year-Round, Full-Time Workers by Sex, 1994
        Number (in 1000s) Distribution (%)
Earnings Group    Women      Men    Women     Men
$7,500 or less     6233     6038   14.00%   9.50%
$7,501-$12,499     6552     5958   14.70%   9.40%
$12,500-$19,999    9729     9703   21.90%  15.20%
```

6.5 External Programs

What if the data you want are not in a file, but you can obtain them by executing an external program? Execute a command by enclosing it in backquotes.

```
$date    = `date`;
@listing = `ls -l`;
@dirList = `dir`;
```

The `system()` function is similar to backquotes.

Command or Function	Description
`` `command` ``	Executes *command* in a subshell (usually `/bin/sh`) and returns *command*'s output.
`system(`*program*`, `*args*`)`	Executes *program* and waits for it to return. The function `system` returns the exit status of *program* multiplied by 256.

The following command concatenates the files `results.dat` and `total.dat` to produce `report.dat`.

```
system("cat results.dat total.dat > report.dat");
```

The function `open()` is quite versatile. You can instruct your program to read data from the output of a command or to send the output to a command with `open`.

Command	Description
`open(`*HANDLE*`, "`*command*`|")`	read from the output of *command*
`open(`*HANDLE*`, "|`*command*`")`	write to the input of *command*

Using pipes, you can send and receive email from a Perl program. Here's how I can send email to myself when there is a problem reading the date in `$file`.

```
$USERLIST = "nancy\@VariableSymbols.com";
open MAIL, "| mail $USERLIST";
print MAIL "Error in $0:  Can't read date in $file\n";
close MAIL;
```

The special variable `$0` is set to the name of the program being run. Other special variables are listed in the table on page 46.

6.6 Inspecting Files

Perl provides a slew of file tests that let you inspect a file. File tests are not like other functions. Each is a hyphen followed by a single letter.

```
$file = "/data/risk/repo.19990417";
if (-e "$file") { print "$file exists;" }
```

This tests whether the file `/data/risk/repo.19990417` exists. The following table shows the most commonly used file tests.

File Test	Meaning
`-e`	File exists.
`-f`	File is a plain file.
`-d`	File is a directory.
`-T`	File seems to be a text file.
`-B`	File seems to be a binary file.
`-r`	File is readable.
`-w`	File is writable.
`-x`	File is executable.
`-s`	Size of the file (number of bytes).
`-z`	File is empty (has zero bytes).

6.7 Deleting Files and Creating and Deleting Directories

Now that I've shown you how you can call an external program, you could call `rm` or `delete` using backquotes or the `system()` command but I suggest using `unlink()` instead. Perl executes a built-in command much faster than an external program.

Command	Description
`unlink(`*files*`)`	delete *files* returning the number of files successfully deleted. E.g., `unlink "$OUTPUTDIR/$filename";`

In addition to being able to create and destroy files, you can create and destroy directories with `mkdir()` and `rmdir()`. Note, `rmdir()` will delete the directory only it it's empty.

Just because these two commands have the same name as their counterparts in UNIX, don't expect that you can call other UNIX commands in the same way. If you want to change directories, if you are a UNIX user, you might think that the command is `cd()`, but it's actually `chdir()`.

6.8 Exercise

Hasn't it been a while since I've given you an exercise?

1. The command `ls -l` on a UNIX computer and the command `dir` on DOS lists the contents of directories. Given the following output from `ls -l`,

```
-rwxr-xr-x   1 nb   nb    158 Mar 27 17:52 email.pl
-rwxr-xr-x   1 nb   nb    212 Mar 27 15:34 writeReport.pl
-rwxr-xr-x   1 nb   nb    523 Mar 27 15:32 readRates.pl
-rw-r--r--   1 nb   nb     41 Mar 27 13:05 rates.dat
```

   write a Perl program that reads this information or the result of running the equivalent command on your computer (`dir` on DOS) and creates two files with the column headings `FILES` and `SIZES` listing the file names and their sizes. Produce one file using `printf()` and the other file using `format`. At the end of the file put the current time and date.

   So your program should generate output that looks something like this:

```
FILES            SIZES
email.pl         158
writeReport.pl   212
readRates.pl     523
rates.dat         41

Date:  Sat Mar 27 18:12:33 EST 1999
```

   Hint: The command `printf()` with the template `%10s` will display a string using 10 spaces.

```
#!/usr/bin/perl
printf "File:  %10s\n", "test1";
printf "File:  %10s\n", "test2";
File:      test1
File:      test2
```

7 Subroutines

So far, I've shown you short programs that illustrate a specific aspect of the Perl language. If you want to do something complicated, your program will get large. You can make it easier to manage and debug by breaking it up into smaller pieces, each of which accomplishes some task and computes some result that is used to compute the end result. After grouping related statements together into a *subroutine*, you can execute those statements with a single expression. A subroutine can be placed anywhere in your program and executed by calling that subroutine.

7.1 Defining Your Own Subroutines

When do you use a subroutine? When a chunk of code makes sense by itself—in other words, if it performs a self-contained task—it probably merits a subroutine. Give your subroutine a name that indicates what it does.

Subroutines are defined with `sub`. Here is a subroutine that prints the name of a good book.

```
sub good_book{
    print "The Mind Body Problem\n";
}
```

Subroutine declarations aren't statements, so you don't need a semicolon at the end.

The subroutine `good_book()` doesn't do anything until it's invoked.

```
good_book();
```

The Mind Body Problem

Sometimes it won't be clear to Perl whether you're calling a subroutine or using a string:

```
$n = a_sub;
```

set `$n` to the value returned by the `a_sub()` subroutine, but only if the subroutine has been declared before that statement. Otherwise `$n` is set to the five-character string "`a_sub`". You can put an `&` before a name to identify it as a subroutine call. In Perl 5, the `&` is required only when the name might refer to something else (such as a variable) or when the subroutine definition appears after the subroutine call and you omit the parentheses. So you might see subroutines invoked as

`&a_sub;` or `a_sub();` or `&a_sub();` or just plain `a_sub;`

Use an `&` when you want to give a subroutine the name of a built-in Perl function. For instance, if you wanted to name your subroutines `open()` and `close()`, you would need to precede the calls to your subroutines with `&` otherwise, Perl would invoke the routines for opening and closing filehandles. But I encourage you to use names that are distinct from those built into Perl so that users will not get confused about which routine is being invoked.

A function is a subroutine that returns a value. Here is a function that computes the maximum of two or more values. You can pass arguments to a subroutine with the `@_` array. Perl places the arguments to a subroutine in the array `@_`, which is similar to a normal array (`@movies` or `@days`). When you want to return a value for a subroutine, use `return()`.

```
# max($n1, @n2);
# Returns the maximum of the arguments.
sub max {
    my ($n1, @n2) = @_;

    foreach $m (@n2) {
        if ($n1 < $m) {
            $n1 = $m;
        }
    }
    return ($n1);
}
```

Here we call `max()` with five arguments. Note that Perl combines scalar arguments and arrays into a single list.

```
$n = max(2, (1,2,3,4), 5);
```

The variable `$n` is set to the maximum, which is 5.

7.2 The Scope of Variables

Declaring the variables for `max()` with `my()` makes them visible only to the current block (in this case, within the function `max()`). In other words, `my()` performs lexical scoping. If you declare a variable using `local()` it is visible to the current block and any subroutines called from within that block. In other words, `local()` performs dynamic scoping. Let me show you the difference between local and dynamic scoping.

```
#!/usr/bin/perl
$a = 4;
$b = 9;
print "Initially:  a is $a and b is $b\n";
&outer;
print "Afterwards:  a is $a and b is $b\n";
sub outer {
    my ($a);        # Declare a new $a, lexically scoped
    local ($b);     # Declare a new $b, dynamically scoped
    $a = 44;
    $b = 99;
    print "outer subroutine:  a is $a and b is $b\n";
    &inner($a, $b);
}
sub inner {
    print "inner subroutine:  a is $a and b is $b\n";
}
```

```
Initially:  a is 4 and b is 9
outer subroutine:  a is 44 and b is 99
inner subroutine:  a is 4 and b is 99
Afterwards:  a is 4 and b is 9
```

Values in Perl exist until they are explicitly freed. They are freed by the Perl garbage collector

when the reference count (the number of variables that refer to the value) becomes zero, by a call to the `undef` function, or if they were declared `local` or `my` and the scope no longer exists. So the Perl interpreter will automatically destroy variables that are declared `my` in a subroutine when the subroutine returns.

7.3 Passing References

When you pass more than one argument to a subroutine, the boundaries of the arguments get lost. For example, if you passed the arrays `(1,2,3)` and `(4,5,6)` then `@_` is set to `(1,2,3,4,5,6)`. We can keep the array arguments intact by passing references to the arrays instead of the arrays themselves. Use `\` to create a reference. Here's an example.

```
#!/usr/bin/perl
@array1 = (1,2,3);
@array2 = (4,5,6);
passingArrays(\@array1,\@array2);
sub passingArrays{
    my($array1Ref, $array2Ref) = @_;
    my (@array1) = @$array1Ref;
    my (@array2) = @$array2Ref;

    print "Array 1 is @array1\n";
    print "Array 2 is @array2\n";
}
```

```
Array 1 is 1 2 3
Array 2 is 4 5 6
```

Notice in the function `intersection()`, I passed references to lists.

```
# intersection ($aRef, $bRef)
# Returns a sorted list of the elements common to the lists
# referenced by $aRef and $bRef.  Input lists must be sorted.
sub intersection {
    my ($aRef, $bRef) = @_;
    my ($aSize);    # size of a array
    my ($bSize);    # size of b array
    my @c = ();
    my ($i, $j);

    $aSize = $#$aRef;
    $bSize = $#$bRef;

    $i = $j = 0;
    while (($i <= $aSize) && ($j <= $bSize)) {
        if ($$aRef[$i] == $$bRef[$j]) {
            push (@c, $$aRef[$i]);
            $i++;
            $j++;
        } elsif ($$aRef[$i] > $$bRef[$j]) {
            $j++;
        } else {
            $i++;
        }
    }
    return (@c);
}
```

You would call `intersection()` as follows.

```
@a = (1,2,3,4,5,6);
@b = (1,3,5,7);
@c = intersection(\@a,\@b);
print "@c\n";
```

1 3 5

I hope that now you can write your own Perl subroutines and functions and that you will use local variables instead of making all your variables global.

7.4 Exercises

1. Write the function `union()`, which takes two arguments (references to sorted arrays), and returns a list of all the distinct elements that appear in the arrays referenced by the arguments. Test your definition of `union()` on the arrays `@a` and `@b`.

   ```
   @a = (1..6);
   @b = (1,3,5,7);
   union(\@a,\@b);
   ```

   Make sure your function doesn't list any element more than once.

2. The Perl `sort()` function is designed to rearrange an array in order according to the ASCII value of the characters.

```
#!/usr/bin/perl

@a = (3,2,9,23,12,54);
@b = sort @a;

print "Original \@a:       @a\n";
print "Simple sort \@b:    @b\n";
```

Original @a: 3 2 9 23 12 54
Simple sort @b: 12 2 23 3 54 9

Because 1 comes before 2 in ASCII, `sort()` placed 12 before 2. If a subroutine is provided as an argument to `sort()`, it will be called, but not in the conventional way. Instead of using `@_` to pass arguments, the elements to be compared appear in `$a` and `$b`. Here's code that sorts a list into numerical order, i.e., each element is equal to or larger than its predecessor.

```
#!/usr/bin/perl

@a = (3,2,9,23,12,54);
@c = sort numerically @a;

print "Original \@a:        @a\n";
print "Numerical sort \@c:  @c\n";

sub numerically { $a <=> $b }
```

Original @a: 3 2 9 23 12 54
Numerical sort @c: 2 3 9 12 23 54

(a) Write the subroutine `alphabetically` and verify that it works by sorting the list `@a`.

```
@a = (P,r,i,n,c,e,t,o,n);
```

(b) Set up a hash keyed on the names of cities in which you've lived. The hash should contain the years that you lived in each city. Then use `sort` to print a list of the cities and dates first sorted alphabetically by place and then numerically by date.

Test your program by running it on itself.

8 Using Library Modules

Not only can you break up your programs into subroutines, you can also break up your program into separate files. A file with a set of subroutines is called a Perl *library*.

For my current job, I wrote a library called `date.pl`, which contains the following date related functions.

```
countDays($date1, $date2);
  Return the number of days between the two dates.
  Always returns a positive number.

datePlus($date, $delta);
  Return the date that is $delta days after $date.

formatDate($date);
  Print $date in the format MM/DD/YYYY.

curveSettleDate($date);
  Returns the first day of the month following $date
  or the next business day, whichever comes first.
```

8.1 Loading a Library

Using the `require()` function, you can include a library and it will be loaded, if it hasn't already been.

```
require 'date.pl';
```

How does Perl know where to find the file `date.pl`? The array `@INC` contains a list of directories where Perl searches for library and module files. On my computer `@INC` contains:

```
perl -e 'print "@INC\n"'
```

/usr/lib/perl5/i386-linux/5.00404 /usr/lib/perl5
/usr/lib/perl5/site_perl/i386-linux /usr/lib/perl5/site_perl .

If your library is not in one of the directories in the array `@INC`, you could add the directory with `push()`

```
push(@INC, '/home/nb/perl/lib');
```

or you could specify the location to `require`.

```
require('/home/nb/perl/lib/date.pl');          # UNIX
require('C:\language\perl5:library\date.pl');  # Dos
require('MIKI: Perl 5:Library/date.pl');       # Macintosh
```

8.2 Libraries and Modules

There are attributes of library files that distinguish them from conventional Perl programs:

- They don't have a `#!/usr/bin/perl` at the beginning because they aren't stand alone programs.
- They must return a `True` value as their last statement so that `require()`, `do()`, or `use()` know that the code has been executed successfully.

If you look at a Perl archive, such the Comprehensive Perl Archive Network (CPAN), which can be found on the Web at `www.cpan.org`, you'll see files ending in `.pl` and `.pm`. A file ending in `.pl` is a Perl library. A file ending in `.pm` is a Perl module. Modules:

- Usually contain subroutines or methods for use by other programs (just like `.pl` files).
- Are packages and therefore contain their own symbol tables.

By convention, module names are capitalized. At the beginning of a package is a call to `package()`, which switches to another namespace. Symbols and variables in a module are maintained in their own name space, which means that the subroutines in the module can't access global variables that are outside the module. The module has its own set of variable, which may have the same name as variables in the program that called it.

Here are the first few lines of a package I wrote called `Sets.pm`

```
package Sets;

require Exporter;

@ISA    = qw(Exporter);
@EXPORT = qw(complement intersection union max min);
```

If I hadn't exported the functions, a user would have had to enter `Sets::`*function* to invoke *function* in the `Sets` module instead of simply *function*.

There are many modules that are included with the standard release of Perl. You can find hundreds of other publicly available modules on CPAN. Check it out. You'll probably find utilities that are useful to you. I'm just going to describe several of them.

8.3 Date

Here's what the `readme` file for the `Date::Manip` module says:

> This is a set of routines designed to make common date/time manipulation easy to do. Operations such as comparing two times, calculating a time a given amount of time from another, or parsing international times are all easily done. From the very beginning, the main focus of `Date::Manip` has been to be able to do *any* desired date/time operation easily, not necessarily quickly. There are other modules that can do a small subset of the operations available in `Date::Manip` much quicker than those presented here, so if speed is a primary issue, you should look elsewhere. Check out the CPAN listing of Time and Date modules. But for sheer flexibility, I believe that `Date::Manip` is your best bet.

I used the `Date::Manip` in my `date.pl` library.

```
use Date::Manip;    # Perl module available through CPAN

# datePlus($date, $delta)
# Return the date that is $delta days after $date,
# which is specified in the form YYYYMMDD.
sub datePlus {
    my($theDate, $delta) = @_;
    my ($day1, $day2, $date2);

    $day1   = &ParseDate($theDate);
    $day2   = &DateCalc($day1, $delta, \$err);
    $date2  = &UnixDate($day2,"%Y%m%d");

    return ($date2);
}
1;  # Return a TRUE value so that require(), use(), and do()
    # know that the code has been executed successfully.
```

8.4 Tk

If you want a graphical user interface to your program and your system supports X-windows, I encourage you to check out Perl/Tk. I was pleasantly surprised that I could put together an interface with buttons, list boxes, and scroll bars within a half an hour and most of that time was reading about Tk and typing in code.

```
#!/usr/bin/perl
use Tk;                         # include Tk package

$mw    = MainWindow->new();     # Create application window
$count = 0;                     # Initialize counter

# Define buttons
$mw->Button(-text => "Add 1",
            -command => sub { $count++ }) -> pack(-side => 'left');
$mw->Button(-textvariable => \$count)     -> pack(-side => 'left');
$mw->Button(-text => "Exit",
            -command => sub {exit })      -> pack(-side => 'left');

MainLoop;                       # Initiate event processing
                                # Process input from user
```

8.5 English

As you undoubtedly have noticed, Perl uses special characters for representing special variables, e.g., `$_` and `$0`. After you have loaded the English module, with `use English;`, you can replace some predefined English names, e.g., `$ARG` or `$PROGRAM_NAME`.

Here is a table of Perl special variables. The first column contains the "normal" variable names; the second column contains the long name (or names) imported if you `use` the Perl module `English`.

Special Variables	English Names	Description
`$_`	`$ARG`	The default variable.
`@_`		Subroutine arguments.
`$/`	`$RS`, `INPUT_RECORD_SEPARATOR`	The "line" identifier for input files (default: `\n`).
`$|`	`$OUTPUT_AUTOFLUSH`	When set, the current filehandle will be flushed after every write (default: 0).
`$0`	`$PROGRAM_NAME`	The name of the running program.
`$^X`	`$EXECUTABLE_NAME`	The name of your Perl.
`$^O`	`$OSNAME`	The name of your operating system.
`$]`	`$PERL_VERSION`	Perl's version number.

9 Programming with Perl on the Web

Though Perl was developed in 1987, it only became popular after the birth of the World Wide Web. You can easily create dynamic Web pages with Perl, such as pages that query you for information or present you with different links depending on your input, or the time of day, with a program that generates HTML automatically. The Web server executes the program when a user visits the page. Dynamic Web pages are not built from ordinary HTML text files; instead, small programs or scripts create the HTML source text that a Web browser then displays. Because scripts generate documents on the fly, they are capable of incorporating information that changes or that cannot be determined in advance. They can also be used to solicit and interpret user-supplied data, retrieve requested information, and produce content that is customized for a particular user.

Web scripts communicate with Web services, which in turn, relay information to a Web browser to display to a user. The World Wide Web is based on a client-server model. The server provides resources and a client requests them. These resources might include computing power, storage, or hardware devices. Servers are the computers that host Web sites and clients are users' browsers. The client and server communicate using the Hypertext Transfer Protocol (HTTP), conventions set up for exchanging data. With a CGI gateway, data passes from the Web server to the Perl program that generates HTML documents. These programs are often called CGI scripts.

Short Name	**Long Name**	**Description**
URL	Universal Resource Locator	A hyperlink to another Web resource
WWW	World Wide Web	A set of Internet services based on hypermedia
HTML	Hypertext Markup Language	A syntax for representing information as hypertext
HTTP	Hypertext Transfer Protocol	A protocol used by Web clients to communicate with servers
CGI	Common Gateway Interface	A standard for interfacing programs and the Web

On a UNIX machine, CGI scripts usually run under a particular account with limited permissions. Your script can't be run unless you make it world executable. Typically scripts are stored in particular directories on the server.

9.1 CGI.pm

The `CGI.pm` module, written by Lincoln Stein, is the most popular Perl utility for creating Web fill-out forms on the fly that parse their content. With it, you don't need to remember the syntax for HTML form elements. Instead you just call Perl functions. Here is a sample script to create a fill-out form that remembers its state each time it's invoke.

```
#!/usr/local/bin/perl
use CGI ':standard';

print header;
print start_html('A Simple Example'),
    h1('A Simple Example'),
    start_form,
    "What's your name? ", textfield('name'),
    p,
    "What's the combination?",
    p,
    checkbox_group(-name=>'words',
           -values=>['eenie', 'meenie', 'minie', 'moe'],
           -defaults=>['eenie', 'minie']),
    p,
```

```
        "What's your favorite color? ",
        popup_menu(-name=>'color',
               -values=>['red', 'green', 'blue', 'chartreuse']),
        p,
        submit,
        end_form,
        hr;

    if (param()) {
        print
        "Your name is: ", em(param('name')),
        p,
        "The keywords are: ", em(join(", ", param('words'))),
        p,
        "Your favorite color is: ", em(param('color')),
        hr;
    }
    print a({href=>'../cgi_docs.html'}, 'Go to the documentation');
```

`CGI.pm`, like many modules, provides you with *import tags*—labels that stand for groups of functions to import. Instead of loading all of `CGI.pm`, I loaded a subset of its functions with the command `use CGI ':standard';`.

I hope you are at least somewhat familiar with the HTML language. Below is an HTML file `gateway.html` that calls the Perl script `sample.cgi` when the user clicks on the link. Replace `your.web.server` with your host name's Internet address, e.g., `www.VariableSymbols.com`.

```
<H1> Gateway to Lincoln Stein's sample CGI Program</H1>

Click <A HREF="http://your.web.server/cgi-bin/tryIt.cgi">here</A> to
run Lincoln Stein's sample CGI program.
```

When you visit the page `http://your.web.server/gateway.html`, you'll see this page:

> ## Gateway to Lincoln Stein's sample CGI Program
>
> Click here to run Lincoln Stein's sample CGI program.

When you click on the word *here* on the gateway page, the Perl program `tryIt.pl` is launched and you'll see a simple query form.

10 Perl Style Guide

Here is a list of style guidelines put together by Larry Wall, the developer of Perl.

- Opening curly brace on same line as keyword, if possible, otherwise line up.
- Space before the opening curly brace of a multi-line block.
- No space before the semicolon.
- No space between function name and its opening parenthesis.

Here's an example using the above style recommendations.

```
$i = 10;
while($i > 0) {
    print "$i ";
    --$i;
}
```

- One-line blocks may be put on one line, including the curly braces.
- Semicolon omitted in "short" one-line block.
- Space around most operators.
- Space around a "complex" subscript (inside brackets).
- Blank lines between chunks that do different things.
- Space after each comma.
- Long lines broken after an operator (except `and` and `or`).
- Space after last parenthesis matching on current line.
- Line up corresponding items vertically.
- Omit redundant punctuation as long as clarity doesn't suffer.

10.1 Differences Between C and Perl

Here's a comparison of the data types supported in the two languages.

C	**Perl**	**Perl Example**
`char`	strings	`$name = "foo";`
`int`, `short`, `long`	integer literals	`$term = 42;`
`float`, `double`	float literals	`$principal = 1000000.00;`
array	array	`@movies ("E.T.", "Tootsie");`
`struct`	hash	`%daysPerMonth = ('Jan' => 31, 'Feb' => 28);`
`union`	nothing equivalent	
`typedef`	nothing equivalent	
file descriptor	file handle	`STDOUT` or `DATA`
objects in C++	objects in Perl 5	

Below is a table comparing capabilities, constructs, and function names in C and Perl.

Construct	**C**	**Perl**	**Comments**
Address-oriented operators	`&` (address of)	`\`*variable*, `$`*ref* to dereference	
Command-line Arguments	`argv[0]`	`$0`	Program name
	`argv[1]`	`$ARGV[0]`	
Comments	`/*` *comment* `*/`	`#` *comment*	
Comparison	`<`, `<=`, `==`, `>=`, `>`, `!=`, for numbers only	different operators for numbers and strings	
Conditionals	`if(`*expr*`)` *statement*	`if(`*expr*`){`*statement*`}`	Curly brackets required in Perl.
	`if/else if`	`if/elsif`	
	`switch`		Doesn't exist in Perl, but you can write one yourself.
File inclusion	`#include "`*file*`"`	`require '`*file*`'`	
		`use` *module*	
Invoking functions	need `()`	`()` optional	
Loops	`while(`*expr*`)` *statement*	`while(`*expr*`){`*statement*`}`	
	`break`	`last`	`last` doesn't work with `do{} while`.
	`continue`	`next`	`next` doesn't work with `do{} while`.
	nothing equivalent	`foreach`	
Memory management	`alloc`, `malloc`	reference-based garbage collection	
Pointers/reference	`*`*variable*	`$`*ref* to dereference `\`*variable*	In Perl, you can't take the address of anything. You can get a reference to a variable with backslash.
Storage classes	automatic, `extern`, `static`	global, `my`, `local`	

Subroutine	*name*`(`*argument list*`)`
	argument declarations
	`{`
	  body
	`}`
		`sub` *name* `{`
		  body
		`}`	

Type casting	`(`*type*`)`	nothing equivalent	
Variable declarations	required	optional	
Variable names	begin with a letter	begin with `$`, `@` or `%`	

11 Just What More?

I sincerely hope this tutorial has helped you to become (more) proficient in Perl. I have tried to anticipate your questions and problems. Please let me know if I have missed something. I welcome all comments. I look forward to hearing from you.

Nancy Blachman
Variable Symbols, Inc.
356 Bush Street
Mountain View, CA 94041
Email: `nancy@blachman.org`
650 966 8999

12 Answers to the Exercises

12.1 Getting Started, page 4

1b

2c

3a – 3f

3b – 3d

3c – 3e

12.2 Scalars, Functions, and Operators, page 9

1d

2c

3d

12.3 Control Statements, page 15

1c

12.4 Hashes or Associative Arrays, page 21

1d

2c

4b

5c

12.5 Pattern Matching: Regular Expressions, page 28

1b

2a

3d – The matching string must have at least one *a*.

4c – The matching expression must have at least four characters, at least one *d*, followed by two other characters, which can be anything, and followed by at least one *g*.

13 References

I referred to these books and documents when preparing these notes. I listed the most helpful ones first.

Perl 5 Interactive Course: Certified Edition by Jon Orwant (Waite Group Press, 1998)

Advanced Perl Programming by Sriram Srinivasan (O'Reilly, 1997)

Perl Power! by Michael Schilli (Addison-Wesley, 1999)

Introduction to CGI/PERL: Getting Started with Web Scripts by Steven E. Brenner and Edwin Aoki (M&T Books, 1996)

CGI.pm — a Perl 5 CGI Library by Lincoln Stein (Version 2.25, 9/10/96)

Perl 5 Unleashed by Kamran Hasain and Robert F. Breedlove (SAMS Publishing, 1996)

Learning Perl, 2nd Edition by Randal L. Schwartz and Tom Christiansen (O'Reilly, 1997)

Programming Perl, 2nd Edition by Larry Wall, Tom Christiansen and Randal L. Schwartz (O'Reilly, 1996)

Perl by Example, Second Edition by Ellie Quigley (Prentice Hall, 1998)

The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie